(60)

Certificate Discovery Using SPKI/SDSI 2.0 Certificates

by

Jean-Emile Elien

S.B. Massachusetts Institute of Technology (1997)

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

May 1998
[June 1998]

Author .....
Department of Electrical Engineering and Computer Science
May 22, 1998

Certified by .....
Ronald L. Rivest
Edwin Sibley Webster Professor of Computer Science and Engineering
Thesis Supervisor

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

Certificate Discovery Using SPKI/SDSI 2.0 Certificates

by

Jean-Emile Elien

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 1998, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

In this thesis, a framework was designed to implement certificate discovery in SPKI/-SDSI. The process of discovering a valid certificate chain is shown to be analogous to a string re-writing problem of finding certain string derivations. Given this framework, an algorithm was devloped to solve the problem and is shown to a time polynomial in the number of certificates.

Given the theoretical framework, two certificate discovery implementations were developed. The first, written in Perl, tooks a simple certificate format and generated the necessary sequence based upon the chain desired. The second implementation, in C, was written as part of a library that is designed for the SPKI/SDSI framework.

Thesis Supervisor: Ronald L. Rivest
Title: Edwin Sibley Webster Professor of Computer Science and Engineering

Acknowledgments

First and foremost, I would like to thank my wonderful fiancée, Malinda Kay Lutz. She has been a source of strength and motivation the many months that I have worked on developing the theory and implementations. She patiently listened to my thoughts, read my drafts, and baked me lemon bars (and other german bakery) when I was up late. After several years of stuggle at MIT, this thesis is a monument to what her love has done for me.

Not far behind in thanks is my mother, Solette Jean-Pierre. She has done more for me than she could ever know. I hope that this document helps to express my gratitude for her unwavering optimism, wisdom and patience.

I would also like to thank Prof. Rivest (Ron) for his insight and direction in this thesis. With his help, not only is the theory grounded and provable, my understanding of the concepts has been greatly deepened. I am truly in awe of his abilities — at times he just seems to pull insight out of nowhere, and I'd have to think for hours to see why (not if) he is right.

Many thanks to Matt Fredette and Alex Morcos, fellow implementors of the SPKI framework. Matt is simply one of the most impressive hackers I've seen, and Alex plugged a hole in the certificate discovery theory that made things much simpler.

Thanks to all of my friends, at MIT and abroad, who read my drafts and listened to me ramble about certificates, security, and lots of C code.

I wish to thank the writers of emacs and LaTeX — two tools without which I would probably still be fixing the cross-references for this thesis.

This research is supported by DARPA Grant DABT63-96-C-0018.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

With the growing popularity of the Internet for commercial and personal interaction, the question of trust is an expanding concern. How does one "trust" the credentials of a person over the Internet? For example, MIT allows its students to view their records over the World Wide Web. What credentials does a student need in order to gain access to his records?

One solution to this problem is borrowed from the physical world. To prove identity in the physical world, one presents a document issued by an authority (the state government, or a hospital) that binds the person to the name such as a valid driver's license or birth certificate. The analogy to this idea in cyberspace is the use of a *certificate*: an electronic document that is digitally signed by an authority and makes some statement. At MIT, a certificate from MIT is issued to each student. If a student wishes to review his grades over the Web, that certificate is presented to the Registrar's web server at the time of the request (along with his student ID and password). The server verifies that the signed certificate is valid (i.e. MIT states this person is a student with some ID), and responds with the grade information over an encrypted channel.

The dominant use of certificates in the digital domain is from within some *Public Key Infrastructure* (PKI). A PKI makes use of *private* and *public* keys that can sign

and verify the signature of any digital document. These keys have several properties:

- The public key is meant to be distributed. It is meant to be readily available, anyone can get it. On the contrary, the private key is kept confidential, no one but the keyholder should be allowed access to it.
- The private key can be used to sign a document. Since it is kept private, only the owner of the private key can use it.
- The public key can be used to verify a signature, so once a document is distributed, anyone can verify the signature.
- Given the public key, it is very difficult to determine the private key. This allows the public key to be distributed to the network without fear of discovery of the private key.

Given those properties of public and private keys, a certificate (typically abbreviated cert) is a digitally signed document that contains at the very least:

Issuer : The principal[1] signing the certificate. The certificate usually holds the public key of the issuer to allow verification of the signature.

Subject : The principal to which the issuer refers. In the MIT example, the subject of the certificate is the student.

Validity Period : The range of dates the certificate is valid.

The rest of the certificate details information about the subject, such as the location of the subject, authorization given to the subject by the issuer, and permission given to the subject to pass the authority to another principal. A certificate, therefore, becomes the basic document of trust — if the issuer is a known authority, and the issuer states something about the subject, then the subject can be trusted for those tasks.

[1]The term *principal* usually refers to the person who is represented by the key (as in PGP, or X.509). This is not the only definition; in fact, the definition changes with newer PKIs.

This certificate model can become unmanageable very quickly for the issuer. Assume the issuer has a group with one thousand members. Every time the issuer wants to change some authority for the group, the issuer generate one thousand new certificates. It would be desirable to just issue one certificate that all of the members can use.

This has motivated the idea of *certificate chains.* A certificate chain is a sequence of certificates, and uses a transitive property of trust: If Alice trusts Bob, and Bob trusts Charlie, then Alice "trusts" Charlie. The concluding trust is in quotes, because in the physical world, the result isn't necessarily true. I may trust my friend, and she trusts a convicted felon, but that says nothing of my trust of the felon.

Since "trust" is a vague and problematic term, some PKIs prefer to think about *delegation* instead of trust. Delegation is easier to think about because it can be more precisely defined. When "Alice delegates to Bob some authority ($A_{Alice \rightarrow Bob}$)" Alice is giving Bob the right to perform some task, in Alice's name. This makes the transitivity cleaner: If Alice *delegates* to Bob some authority $A_{Alice \rightarrow Bob}$, and Bob *delegates* to Charlie some authority $A_{Bob \rightarrow Charlie}$, then Alice has delegated to Charlie $A_{Alice \rightarrow Bob} \cap A_{Bob \rightarrow Charlie}$. Alice can also control the *propagation* of delegation, such that it would not be possible for Bob to transfer over to Charlie authority given by Alice. Using delegation and propagation, a more concrete notion of trust can be built. Trust is built by proper delegation of authority from key to key, with propagation control to limit the extent of delegation.

1.2 SPKI

The *Simple Public Key Infrastructure* (SPKI) is a movement to replace the specification for X.509[2] with something simpler. X.509 relied on several global structures that made it difficult to implement efficiently. One structure that SPKI tries to remove is the *global name space*, a feature intrinsic to X.509 and PGP[3] (See [3]).

[2]A popular PKI that is built upon global naming hierarchy. See [9].

[3]Pretty Good Privacy [10]. PGP uses a different name space than X.509, but the effect is identical.

A **global name space** is a set of names that is known worldwide. The names are intended to have some mapping the the "real names" of people. For example, my PGP name is "`Jean-Emile Elien ⟨jee_hbm@mit.edu⟩`". It is a string in the global name space that is the set of all PGP keys. The name is supposed to define me uniquely. A certificate only binds my name to my public key; it doesn't (at least, it didn't initially) say anything about authorizations I may have, or even that the name corresponds to me.

On the scale of the entire world, the criterion of a unique name presents a problem. What happens if the person changes his/her name? Several reasons may motivate this, such as marriage, change of name upon citizenship, loss of job, or even graduation. When I graduate, the username in my PGP name becomes invalid, thereby invalidating the whole name.

In PGP, the name is what identifies the person, and the public key is just meant for cryptographic communication. So if the name changes, what becomes of the name's identification to the person? If my real name changes, I can no longer use my PGP key, even though it hasn't been compromised. This is due to the distributed nature of digital certificates. In the physical world, if I change my name, all of my credentials are under my direct control, so there is one centralized place to change all references of my name. This is not possible in cyberspace, because the credentials are meant to be copied and distributed. The PKIs mentioned have the notion of *certificate revocation lists* (crls, See [5]) to revoke and issue new certificates, but that can easily become unmanageable as lists of revoked certificates grow.

Even if names don't change, uniqueness remains a problem. A global name space is, by definition, known to everyone. Even in a small realm, such as a company, this may be a problem. At MIT, there are 3 people with the name "William Smith". It could possibly be solved with employee identification, but a standard identification may not be agreed upon across nations — except for the public keys themselves.

In SPKI, the principal (the identity) is associated with the *public key*, not the person[4]. The motivation is that the public key is a more persistent entity than a

[4]Called the *keyholder* in SPKI.

```
(public-key
 (rsa-pkcs1-sha1
  (e #23#)
  (n
   |AImuUeR3N//ZTUducfR8M8gxMpkkhiqcZJDBnEnPqFd01bjgf1Dx2mZB
   EvZ+AlIanBmnk55dJsB5sCh1hR4KRwEJuVQHCW2AO04eDKsgGlgel2U9L
   4bncwfiEj0tbpSp5bDuzq6vcUHacXlL4xL63Mmc65RUF8rnPupHsdlwLB
   gt|)
 )
)
```

Figure 1-1: A SPKI/SDSI Public Key

name, and must be unique. Otherwise if two or more people had the same public key, they could impersonate each other. While it is theoretically possible to generate the same public key, given the sizes of the keys (1024 bits, about 309 digits), it is extremely unlikely. Additionally, a public key may not even refer to a person; some applications may have an automated principal that deals with credentials.

Because SPKI certificate chains are from public key to public key, delegating authority from one to the next avoids the problems associated with global names by disassociating the name from the identity. Unfortunately, this is very restrictive. Authorization to groups, for example, is cumbersome. Certificates giving authority need to be issued for each individual member, and any changes to those authorities given requires the addition and/or revocation of a potentially large number of certificates.

1.3 SDSI

The *Simple Distributed Security Infrastructure* (SDSI 1.0) [8] (pronounced sŭd-zee) was an independent effort to develop a simpler PKI. SDSI was similar to SPKI in that the principal was the public key, not some string associated with it. The main feature added by SDSI was the notion of *local name spaces* — name spaces that are defined relative to a particular key, which can later be "dereferenced" to a key or another *SDSI name*[5].

[5]Not names in the real-world sense, just string identifiers defined by the principal, such as `MOM`.

```
(name
 (public-key
  (rsa-pkcs1-sha1
   (e #23#)
   (n
    |AImuUeR3N//ZTUducfR8M8gxMpkkhiqcZJDBnEnPqFd01bjgf1Dx2mZB
    EvZ+AlIanBmnk55dJsB5sCh1hR4KRwEJuVQHCW2AO04eDKsgGlgel2U9L
    4bncwfiEj0tbpSp5bDuzq6vcUHacXlL4xL63Mmc65RUF8rnPupHsdlwLB
    gt|)
  )
 MOM HUSBAND)
```

Figure 1-2: A SPKI/SDSI name.

A **SDSI name** is a sequence of arbitrary length consisting of a public key followed by zero or more identifiers. An example of a SDSI name is: K_{Jean} `MOM HUSBAND`. It begins with the key K_{Jean}, that refers to a public key, such as the one in Figure 1-1[6]. The identifier `MOM` following the principal K_{Jean} is understood to be equivalent to another SDSI name (K_{Mom}) in the name space of K_{Jean}. Subsequently, `HUSBAND` is defined in the name space defined by the key that is bound to `MOM`. Figure 1-2 shows a complete name. If a SDSI name does not have any identifiers (e.g. K_{Jean}), it is known as a *trivial* SDSI name.

SDSI's local name space has the benefit of delaying to whom a particular certificate refers. Knowledge of the exact public key is not necessary in order to issue a certificate for a principal. For example, a certificate can have a subject that looks like: K_s `ALICE BOB`. This name means: "the principal bound to the name `BOB` in the name space of the principal bound to the name `ALICE` in the name space of the public key K_s", or more simply, "K_s's `ALICE`'s `BOB`".

This is very different from the global name space of X.509 and PGP. A principal (public key) is free to bind a name to whatever other principal (name or key) it wants. This allows, for example, membership certificates, which is not easily implementable in SPKI. A single name can bind to several keys within the name space of the principal, thereby having the effect of group membership.

[6]Figure 1-1 is an S-expression (see section 2.1.1), not exactly a true SDSI 1.0 public key, but similar.

1.4 SPKI/SDSI 2.0

The two efforts merged into a collaborative effort, SPKI/SDSI 2.0[2]. This effort has taken parts of both SPKI and SDSI 1.0 and eliminated unpopular features, in an effort to simplify the composite PKI even further. SPKI/SDSI combines SDSI names with the delegation of authority of SPKI.

The main new feature of this PKI is the separation of authorization from name definition. This eliminates several hairy problems that arise with the juxtaposition of authority (delegating rights to some subject) with naming (binding some subject to an identifier).

Since an authorization goes from public key to public key, certificates that give authorizations must only have a public key as the issuer — the issuer must be a trivial SDSI name.

Name definition works similarly. Since you really only know and control your own name space, you can, at will, issue certificates that bind subjects to names in your namespace. However, those certificates are intended for naming — they carry no authority. Only having control over your name space also means that you cannot define names in other name spaces. This restricts the number of identifiers in the issuer to at most one.

It is within the framework of SPKI/SDSI that this thesis will discuss the notion of *certificate discovery.*

1.5 Certificate Discovery

Consider a scenario with two parties, a Prover (Paula) and a Verifier (Vincent). Paula wants access to a resource that Vincent controls. In order to prove to Vincent that Paula is authorized, she must present a valid SPKI/SDSI certificate chain to Vincent. Vincent maintains an *access control list* (acl) which contains the set of valid SDSI names allowed access. For example, assume Paula is an MIT student who wants to read the online course catalog, and Vincent is the MIT Registrar's server. On

Vincent's machine there is an acl entry that has the SDSI name $\mathtt{K}_{MIT}$ `STUDENT`. Thus, any student of MIT can read the course catalog.

A simple protocol goes as follows:

1. Paula sends Vincent a request for access.
2. Vincent sends back the SDSI names allowed access to Paula.
3. Paula sends back a valid certificate chain to Vincent.

Certificate discovery is concerned with going from step 2 to step 3. Paula needs to produce a valid cert chain to present to Vincent, starting from an acl entry and ending with her key. Here is a example of a valid chain that Paula can send back to Vincent. The certificates shown are of the form ⟨issuer, subject⟩. The other fields are being ignored for clarity.

$\langle \mathtt{K}_{Vincent}, \mathtt{K}_{MIT}\ \mathtt{STUDENT} \rangle$
$\langle \mathtt{K}_{MIT}\ \mathtt{STUDENT}, \mathtt{K}_{MIT}\ \mathtt{EECS\ \ STUDENT} \rangle$
$\langle \mathtt{K}_{MIT}\ \mathtt{EECS}, \mathtt{K}_{EECS} \rangle$
$\langle \mathtt{K}_{EECS}\ \mathtt{STUDENT}, \mathtt{K}_{EECS}\ \mathtt{EECS\text{-}STUDENT} \rangle$
$\langle \mathtt{K}_{EECS}\ \mathtt{EECS\text{-}STUDENT}, \mathtt{K}_{Paula} \rangle$

The sequence above goes as follows:

- $\langle \mathtt{K}_{Vincent},\ \mathtt{K}_{MIT}\ \mathtt{STUDENT} \rangle$: $\mathtt{K}_{Vincent}$ has delegated some authority to the SDSI name $\mathtt{K}_{MIT}$ `STUDENT`.
- $\langle \mathtt{K}_{MIT}\ \mathtt{STUDENT},\ \mathtt{K}_{MIT}\ \mathtt{EECS\ \ STUDENT} \rangle$: To MIT, the name `STUDENT` is being defined as equivalent to `EECS STUDENT`, a student in the EECS[7] department at MIT. So, via MIT, Vincent is now giving authority to an EECS student.
- $\langle \mathtt{K}_{MIT}\ \mathtt{EECS},\ \mathtt{K}_{EECS} \rangle$: According to MIT, the name `EECS` is equivalent to the key $\mathtt{K}_{EECS}$. At this point in the sequence, Vincent is talking about a student as defined by the EECS key.

[7]Electrical Engineering and Computer Science

- $\langle K_{EECS}$ STUDENT, K_{EECS} EECS-STUDENT$\rangle$: K_{EECS} is doing a name binding of its own. This certificate means that the name STUDENT, according to the EECS department, is equivalent to EECS-STUDENT.

- $\langle K_{EECS}$ EECS-STUDENT, $K_{Paula}\rangle$: The final cert in the sequence, this defines Paula as a member of EECS-STUDENT. This is the final certificate, because via the above certificates, Vincent is now giving authority to anyone who matches the SDSI name K_{EECS} EECS-STUDENT.

If Paula maintains a database of certificates, how is she supposed to extract the correct certificates to send to Vincent? In a database of 20 certificates, it might be possible to find the certificates by hand, but perhaps not if the database has 100 or even 1000 certificates. It should be possible to automate the process of discovering and generate the sequence if a valid cert chain exists.

Given the naming mechanism of SPKI/SDSI, it is not obvious that the process can be automated efficiently. It is possible that the problem is intractable, or even undecidable. The AT&T's PathServer [7] and PolicyMaker [1] gives evidence that a similar problem within their respective frameworks is not solvable in polynomial time. However, an efficient algorithm does exist for SPKI/SDSI, making it possible for Paula to generate the necessary sequence without manual intervention.

1.6 Organization

Chapter 2 develops the theory and framework of the certificate discovery engine. It gives an algorithm for determining if Paula is authorized given her database of certificates, and proves a polynomial-time upper bound on the time it takes to determine the answer.

Chapter 3 discusses two different implementations of the certificate discovery engine. The first, `certgen.pl`, is written in Perl, and the second, `fclose.c`, is written in C.

Chapter 4 concludes with potential areas of future research.

Chapter 2

Theory

2.1 Definitions

In order to talk about the algorithms used to discover a certificate chain from a database of certificates, several notions need to be clarified.

2.1.1 Notation

S-expressions

In SPKI/SDSI, the certificates are encoded as *S-expressions*. S-expressions are Lisp-like structures that are enclosed by parenthesis whose elements are either strings or other S-expressions. In SPKI/SDSI, a naming certificate might look like:

```
(cert
 (issuer (name
   (public-key (rsa-pkcs1-sha1
     (e #23#)
     (n
      |AImuUeR3N//ZTUducfR8M8gxMpkkhiqcZJDBnEnPqFd01bjgf1Dx2mZBEvZ+
      AlIanBmnk55dJsB5sCh1hR4KRwEJuVQHCW2AO04eDKsgGlgel2U9L4bncwfiE
      j0tbpSp5bDuzq6vcUHacXlL4xL63Mmc65RUF8rnPupHsdlwLBgt|)))
   Masters-Alumnus))
 (subject (name Jean-Emile_Elien)))
 (not-before ``1998-06-05_15:00:00'')
)
```

The main S-expression `cert` has three elements (which are S-expressions themselves), an issuer, a subject, and a validity period. Notice, unlike the SPKI/SDSI name in Figure 1-2, the subject is missing a public key; if none is specified, the issuer's key is assumed.

Assume the public key in the S-expression above is MIT's public key. The above certificate is a naming certificate that states, in the eyes of MIT, the name `Masters-Alumnus` is equivalent to the name `Jean-Emile_Elien`.

Thesis Framework

Certificates The S-expression notation is too cumbersome to use throughout the thesis. Instead, the keys are distinguished by a small number or word for clarity (e.g. $\mathtt{K}_{MIT}$ instead of the key in Figure 1-1). For the process of certificate discovery, only the issuer and subjects are of interest. Because the validity and tag information can be used to select the correct certificates to pass to the discovery engine, they don't need to be represented. The "certificate format" used in this thesis therefore will simply be ⟨issuer, subject⟩, ignoring the other fields of the certificate. The above S-expression will be represented as:

$$\langle \mathtt{K}_{MIT}\ \texttt{Masters-Alumnus}, \mathtt{K}_{MIT}\ \texttt{Jean-Emile\_Elien} \rangle$$

Implications Similar to certificates are *implications*, defined in section 2.1.5. Since, within the framework, they can represent more than simply one certificate, they are displayed differently. If the above certificate were instead an implication, it would look like:

$$\mathtt{K}_{MIT}\ \texttt{Masters-Alumnus} \ \rightarrow\ \mathtt{K}_{MIT}\ \texttt{Jean-Emile\_Elien}$$

The issuer and subject of the implication are both SDSI names. The issuer has at most one name following the key. The subject has zero or more names.

2.1.2 Certificate Reduction

Certificate chains can alleviate the problem of an issuer having to maintain a large set of certificates for any authorization, given or revoked. For example, if an issuer wants to give some privileges to a group, all she needs to do is give each member of the group a name definition $\langle \mathtt{K}_i\ \mathtt{MEMBER},\ \mathtt{K}_{member} \rangle$, and issue one certificate that gives the group name some authority $\mathrm{A}_{i \rightarrow MEMBER}$. If she decides to add or revoke some privilege, she only need issue a new certificate with the proper authorization $\mathrm{A}'_{i \rightarrow MEMBER}$, it is not necessary to give a new certificate to each member.

Taken together, the two certificates — one giving authority to the group, and one defining membership — is equivalent to having issued one certificate giving the authority to the specific member; the two certificates *reduce* to one certificate that has the same meaning.

Certificate Reduction is the process of determining the implied statement made by a sequence of certificates. The end result is a "reduced" certificate, with all of the appropriate fields of a normal certificate. The reduced certificate can be derived by a pairwise reduction of two adjacent certificates in the sequence. If the certificates can merge using the combination rules, the two original certificates are replaced by the reduced certificate that has the same meaning. For example, given the following certificates:

$\langle \mathtt{K}_1\ \mathtt{STUDENT},\ \mathtt{K}_1\ \mathtt{Jean\ CLASSMATE} \rangle$

$\langle \mathtt{K}_1\ \mathtt{Jean},\ \mathtt{K}_2 \rangle$

Since the issuer of the second certificate is a proper prefix of the first certificate, it is possible to merge them. The reduced certificate has the issuer of the first certificate — $\mathtt{K}_1$ STUDENT, but the subject has changed. The subject of the reduced certificate is the subject of the first certificate, only the prefix that is the issuer of the second certificate is replaced by the subject of the second certificate yielding $\mathtt{K}_2$ CLASSMATE.

This follows from the local name space concept in SDSI. The first certificate's SDSI name is some key followed by a series of names, where each name is relative the name space of the SDSI name prefix up that point. So, CLASSMATE is relative to $\mathtt{K}_1$ Jean. The second certificate gives a definition for $\mathtt{K}_1$ Jean, so the prefix is replaced

with the new SDSI name, $\mathtt{K}_2$. A more complete presentation of the rules can be found in [2, 3].

2.1.3 Working Subjects

A certificate chain $(\langle i_1, s_1\rangle, \langle i_2, s_2\rangle, \ldots, \langle i_n, s_n\rangle)$ can be thought of as a rewriting of a sequence of *working subjects*[1] $(w_0, w_1, \ldots, w_n)$ where $w_0 = i_1$ and w_i $(0 < i \leq n)$ is the subject of a reduced certificate $\langle i_1, s_i'\rangle$, which is the pairwise reduction of the first i certificates in the chain. For every certificate chain, there is a unique sequence of working subjects that it generates.

For example, consider the following certificate chain:

$\langle \mathtt{K}_0, \mathtt{K}_0\ \mathtt{MIT}\rangle$

$\langle \mathtt{K}_0\ \mathtt{MIT}, \mathtt{K}_0\ \mathtt{EECS\ STUDENT}\rangle$

$\langle \mathtt{K}_0\ \mathtt{EECS}, \mathtt{K}_1\rangle$

$\langle \mathtt{K}_1\ \mathtt{STUDENT}, \mathtt{K}_1\ \mathtt{GRAD\text{-}STUDENT}\rangle$

$\langle \mathtt{K}_1\ \mathtt{GRAD\text{-}STUDENT}, \mathtt{K}_1\ \mathtt{Jean\text{-}Emile\_Elien}\rangle$

$\langle \mathtt{K}_1\ \mathtt{Jean\text{-}Emile\_Elien}, \mathtt{K}_2\rangle$

Here is the sequence of working subjects that the chain generates:

$\mathtt{K}_0$ (2.1)

$\mathtt{K}_0\ \mathtt{MIT}$ (2.2)

$\mathtt{K}_0\ \mathtt{EECS\ STUDENT}$ (2.3)

$\mathtt{K}_1\ \mathtt{STUDENT}$ (2.4)

$\mathtt{K}_1\ \mathtt{GRAD\text{-}STUDENT}$ (2.5)

$\mathtt{K}_1\ \mathtt{Jean\text{-}Emile\_Elien}$ (2.6)

$\mathtt{K}_2$ (2.7)

Starting with the beginning of the certificate chain, each working subject is the

[1]Note, this definition does not allow for threshold certs. They are discussed in section 2.4.1.

subject of a reduced certificate with the issuer $\texttt{K}_0$. To determine the certificate that took (2.3) to (2.4), find the longest common suffix of both subjects and remove them. The remaining prefix of each working subject defines the issuer and subject of the certificate. So, the certificate $\langle \texttt{K}_0\ \texttt{EECS},\ \texttt{K}_1 \rangle$ will take subject (2.3) to subject (2.4)[2].

Formally, each working subject is thought of as a "string" with the key and each of the following names is a "character" in the "alphabet" of possible names. So, (2.3) has three characters, $\texttt{K}_0$, EECS, and STUDENT. We define the length $|w_i|$ of a SPKI/SDSI name as the number of characters, so $|(2.3)| = 3$.

Key Nodes

If, in a sequence of working subjects, the length of some working subject w_i is 1, it is said to be a *key node*[3]. Key nodes are special points in the working subject sequence — the subject of the reduced certificate is a trivial SDSI name. For example, in the preceding working subject sequence, (2.1) and (2.7) are key nodes. Key nodes will be important in section 2.4.3, because certain things can only happen at key nodes.

Key Spaces

Similar to key nodes is the idea of a *key space.* A key space is a subset of a name space, where the elements are keys, not full SDSI names. An event is said to "occur in key space" when the two SDSI names involved are simply keys — there are no identifiers in the SDSI name. Given a sequence of working subjects, the key nodes comprise the key space for those certificates.

2.1.4 Prefix Rewrite Rules

By treating the working subjects as strings, each certificate can be thought of as a *rewrite rule* for some working subject. The certificate removes a substring of the

[2]This doesn't always work. If the two working subjects have identical names, and only the keys differ, two derivable certificates are possible. However, the def/auth restriction allows only one interpretation to be valid (see section 2.4.3).

[3]The point where $|w_i| = 1$ is also said to be at *top level.*

working subject, and replaces it with another substring. Notice that since the key is always the first character of every certificate, only the front of the working subject is rewritten. Thus, if each working subject is thought of as a string, then the certificates define a set of *prefix rewrite rules* on the set of possible working subjects. A particular sequence of working subjects then can be thought of as a *derivation* from the start of the sequence to the end.

2.1.5 Implications and Finite Closure

Given the set of rules (certificates), it is possible to generate new *implications* that have the same effect as the composition of multiple rules on a working subject string. For example, given the certificates:

$\langle \mathtt{K}_1\ \mathtt{STUDENT}, \mathtt{K}_1\ \mathtt{Jean} \rangle$

$\langle \mathtt{K}_1\ \mathtt{Jean}, \mathtt{K}_2 \rangle$

It is possible, using the reduction method explained in section 2.1.2, to generate an implication $\mathtt{K}_1\ \mathtt{STUDENT} \rightarrow \mathtt{K}_2$ that corresponds to applying both rules one after another. Since implications are really only reduced certificate chains, a single certificate is automatically an implication. New implications can also be generated by composing rules, as well as composing implications. So, in effect, the implications become new "rules" to use on working subjects.

Treating the certificates as rewrite rules, the question of determining a valid chain among the possible certs is the same as asking if there exists a derivation from any entry from the verifier's acl to the prover's key — whether the initial set of rules can generate an implication of the form: $\mathtt{<ACL\ ENTRY>} \rightarrow \mathtt{K}_{Prover}$

It is clear that a working subject sequence that starts from <ACL ENTRY> and ends with $\mathtt{K}_{Prover}$ can only exist if there exists the derivable implication $\mathtt{<ACL\ ENTRY>} \rightarrow \mathtt{K}_{Prover}$. The problem is that the existence of the implication may be difficult to determine given the way new implications are generated. It is possible that the

complete set of implications (the *closure* of the set of certificates) is infinite, given a finite starting rule set. For example, take this name definition:

$K_1\ A \rightarrow K_1\ A\ A$

This rule alone can generate the implications:

$K_1\ A \rightarrow K_1\ A\ A\ A$

$K_1\ A \rightarrow K_1\ A\ A\ A\ A$

$K_1\ A \rightarrow K_1\ A\ A\ A\ A\ A$

$\vdots$

The size of the closure of that lone rule is infinite, because the subjects of all of the implications grow in length. If it is possible to fix the maximum length of any given rule, then the closure of a set of rules is bounded. However, will this "finite closure" contain useful implications? If, given the set of rules and the access control list, will the finite closure of those rules generate the implication <ACL> $\rightarrow K_{Prover}$ if, and only if there exists a certificate chain from the access control list to the Prover's key[4]?

Assuming the implication can be found at all, it does not imply that it can be found efficiently. If the set of characters used in the rules is Σ, and the maximum length of the subject of any rule is l, then a rough upper bound is $O(|\Sigma|^l)$ implications (assuming no implication's subject is allowed to grow longer than l). If this exponential bound for discovery is tight, then the problem is inherently intractable. Does a polynomial bound exist?

The answer to both questions turns out to be yes, provided the mechanism is defined carefully. To generate the *finite closure*, the composition rules for creating new implications are exactly the same as before, with one exception — the second implication in the composition must have a subject no longer than its issuer. This gives a bound of $O(n^3 l)$, where n is the number of initial certificates.

> **Finite Closure** - *The finite closure* (Γ) *of a set of rules* $\mathcal{R}$*, is defined as a union* $\mathcal{R}$ *and the set of all implications that can be constructed as follows:*

[4]Note — this is the same as asking for the working subject sequence, given section 2.1.3.

Given two implications $\{\mathtt{I}_1 \rightarrow \mathtt{S}_1, \mathtt{I}_2 \rightarrow \mathtt{S}_2\} \in \Gamma$, *if* $\mathtt{I}_2$ *is a prefix of* $\mathtt{S}_1$, *then the two implications reduce as defined in section 2.1.2. The reduced implication* $\mathtt{I}_3 \rightarrow \mathtt{S}_3$ ($\mathtt{I}_3 = \mathtt{I}_1$, $\mathtt{S}_3 = (\mathtt{S}_1$, *where the prefix* $\mathtt{I}_2$ *is replaced by* $\mathtt{S}_2)) \in \Gamma$ *if* $|\mathtt{S}_3| \leq |\mathtt{S}_1|$.

Since $\mathtt{I}_2$ must replace a prefix of $\mathtt{S}_1$, in order to construct $\mathtt{S}_3$, $\mathtt{S}_2$ cannot add more names in than $\mathtt{I}_2$ removes. This leads to the equivalent requirement that $|\mathtt{S}_2| \leq |\mathtt{I}_2|$. So, $\mathtt{K}_1$ `STUDENT` $\rightarrow \mathtt{K}_2$ is a valid second implication, but $\mathtt{K}_1 \rightarrow \mathtt{K}_1$ `A A` is not. All that remains to show is that this restriction on the combination rules still allows the creation of the implication `<ACL ENTRY>` $\rightarrow \mathtt{K}_{Prover}$; equivalent to asking, "Can I prove that I am authorized?".

2.2 Proof of Finite Closure

Claim 1 *Given a set of SPKI/SDSI rules* $\mathcal{R}$ *and its finite closure* Γ, *if a certificate chain exists from* $\mathtt{K}_0$ *to* $\mathtt{K}_1$ *using* $\mathcal{R}$ *then there is an implication of the form* $\mathtt{K}_0 \rightarrow \mathtt{K}_1 \in \Gamma$.

Proof Sketch By contradiction.

Assume that Γ does not have the implication $\mathtt{K}_0 \rightarrow \mathtt{K}_1$. Since a certificate chain exists from $\mathtt{K}_0$ to $\mathtt{K}_1$, there must be a sequence of implications that reduces to $\mathtt{K}_0 \rightarrow \mathtt{K}_1$ (remember $\mathcal{R} \subseteq \Gamma$). Find a sequence that reduces to $\mathtt{K}_0 \rightarrow \mathtt{K}_1$ using the fewest number of implications. From this implication chain, generate a sequence of working subjects $\mathcal{S}$ using the method described in section 2.1.3. The goal is to show that $\mathcal{S}$ always has at least one derivable implication $\notin \Gamma$, showing that $\mathcal{S}$ is not the shortest possible sequence. If $\mathcal{S}$ is not the shortest possible sequence using Γ, then Γ could not be the finite closure.

Several facts about $\mathcal{S}$:

- At least one working subject must have a name following the key. If there were no names, then the sequence would look like: ($\mathtt{K}_0$, $\mathtt{K}_2$, $\mathtt{K}_3$, ..., $\mathtt{K}_1$). Take the first

three subjects, which have the implications $\mathtt{K}_0 \rightarrow \mathtt{K}_2$ and $\mathtt{K}_2 \rightarrow \mathtt{K}_3$. A new implication $\mathtt{K}_0 \rightarrow \mathtt{K}_3$ is derivable, but if Γ had it, it would have already been used to make a shorter sequence. Therefore, $\mathtt{K}_0 \rightarrow \mathtt{K}_3 \notin \Gamma$.

- $\mathcal{S}$ must have at least three subjects: If it only had two, then the sequence would be $(\mathtt{K}_0, \mathtt{K}_1)$ then the implication must have been $\mathtt{K}_0 \rightarrow \mathtt{K}_1$. But that would have been an element of Γ.

- In order to end with the subject $w_n = \mathtt{K}_1$, some subsequence of working subjects $(w_i, w_{i+1}, \ldots, w_{n-1})$ $(0 \leq i \leq n)$ must be in decreasing length:

  $|w_i| \geq |w_{i+1}| \geq ... \geq |w_n|$

Find the longest such sequence. That is, starting from $\mathtt{K}_1$, continue back until the length of the names stop increasing. Once w_i is found, take the implications (c_i, c_{i+1}) that took w_{i-1} to w_i and w_i to w_{i+1} respectively. In order to get from w_i to w_{i+1}, the length of the subject could not have been longer than the issuer, because $|w_i| \geq |w_{i+1}|$. Since $|w_{i-1}| < |w_i|$, the subject of c_i is strictly longer than the issuer, so the subject has at least one name. Since the issuer of c_{i+1} has at most one name, it must be a prefix of the subject of c_i. Therefore, the two implications must combine. If Γ was truly the finite closure, it should contain the above derived rule, but then the sequence wasn't the shortest possible using Γ. Thus, Γ could not have been the finite closure.

2.3 Running Time Analysis

Now that it is known that Γ will contain the correct implication, the question of the running time remains. In order to query Γ to see if the appropriate implication exists, the closure must first be generated. If Γ is too large, it might take an unreasonable amount of time (and space) to generate. The dominant factor in the time to response given initial rules is the actual size of Γ, so we take the size of Γ to be the best measure of how long it takes to wait for an answer. A crude analysis in section 2.1.5

gave an upper limit of $O(|\Sigma|^l)$, where Σ is the total number of characters and l is the maximum length of any subject. Since each rule contains at least one name, Σ is roughly the number of starting rules. Since the exponential-space bound for the number of rules is unacceptable, a better bound must be found. In order to determine a better bound, consider the worst-case bound for one rule.

2.3.1 Single Rule Worst Case Bound

Given $\mathtt{I}_1 \rightarrow \mathtt{S}_1$ ($|\mathtt{S}_1| = \mathcal{C}$) is the first implication and $\mathtt{I}_2 \rightarrow \mathtt{S}_2$ is a valid second implication:

1. Given that $|\mathcal{R}| = n$, there are at most n keys that can start $\mathtt{I}_2$. The set of keys in $\mathcal{R}$ is fixed; there can be at most $2n$ keys — two for each rule. Given the combination rules, any key that is in an issuer will remain an issuer's key for any implication in Γ, similarly for keys that start the subjects in $\mathcal{R}$. Half of the total number of keys are issuers, leaving at worst n keys to begin $\mathtt{I}_2$.

2. Since $|\mathtt{S}_2| \leq 2$, the combination will, at most, change the first name of $\mathtt{S}_1$ by either replacing it with another name or removing it entirely. This has a transitive effect:

   Any valid second implication places at most one name back on the subject. Thus, the subject of any implication that can be derived from $\mathtt{I}_1 \rightarrow \mathtt{S}_1$ differs from $\mathtt{S}_1$ in at most the first name. The remaining suffix is also a suffix of $\mathtt{S}_1$. This is equivalent to saying that the key was derived by taking some prefix of $\mathtt{S}_1$ and replacing it with at most one key and one name.

3. There are at most n possible names that can appear at the end of any implication. This is an effect of the fact that only the prefix of any implication is rewritten. Since only the front of the implications change, the names that started at the end (at most n), remain at the end for any implication generated.

Given these facts, it is possible to construct a complexity bound for the number of rules generated by one rule. Using (2), all implications that can be derived from

$$\begin{array}{lcl} \mathtt{K}_1\ \texttt{<name>} & \rightarrow & \underbrace{\mathtt{K}_i\ \mathtt{n}_1\ \mathtt{n}_2\ \ldots\ \mathtt{n}_i}_{\mathtt{K}_j\ \mathtt{n}_x}\ \ \mathtt{n}_{i+1}\ \mathtt{n}_{i+2}\ \ldots\ \mathtt{n}_C \\ \mathtt{K}_1\ \texttt{<name>} & \rightarrow & \mathtt{K}_j\ \mathtt{n}_x\ \ \mathtt{n}_{i+1}\ \mathtt{n}_{i+2}\ \ldots\ \mathtt{n}_C \end{array}$$

Figure 2-1: General Single Rule Combination Format

$\mathtt{I}_1 \rightarrow \mathtt{S}_1$, at worst, have to be of the form shown in Figure 2-1 — the first ith names of the subject is replaced with at most one name, $\mathtt{n}_x$. All that remains is to determine the number of possible implications that can make the new prefix $\mathtt{K}_j\ \mathtt{n}_x$.

Using (1), $\mathtt{K}_j$ can be at most n values. Similarly, using (3), n_x can have at most n values. This gives a total of n^2 possible prefixes. Since any prefix of $\mathtt{S}_1$ can be replaced, this makes a total of $n^2\mathcal{C}$ possible implications that can follow one implication.

2.3.2 Total Worst Case Bound

From section 2.3.1, the most number of implications that can be generated from a single rule is $O(n^2\mathcal{C})$. Since there are n total rules in $\mathcal{R}$, the total bound for the size of Γ is $O(n^3\mathcal{C})$. This is considerably better than the exponential bound given before. This shows that, unlike PolicyMaker, the SPKI/SDSI framework has the ability to efficiently determine whether a principal is authorized to perfom some task.

The size bound on Γ is actually the least upper bound possible. It is possible to construct a sequence that gives $O(n^3\mathcal{C})$ implications. Consider a set of $3n + 1$ rules in $\mathcal{R}$ divided into four parts: (In all cases $1 \leq i \leq n$)

1. 1 rule of the form:

$$\mathtt{K}_1\ \mathtt{A} \rightarrow \mathtt{K}_1$$

2. n rules of the form:

$$\mathtt{K}_0 \rightarrow \underbrace{\mathtt{K}_1\ \mathtt{A}\ \mathtt{A}\ \ldots \mathtt{A}\ \mathtt{X}_i}_{\mathcal{C}}$$

3. n rules of the form:

$$\mathtt{K}_1\ \mathtt{A} \rightarrow \mathtt{K}_1\ \mathtt{X}_i$$

4. n rules of the form:

$$\mathtt{K}_1 \rightarrow \mathtt{K}_i$$

Each $\mathtt{X}_i$ is a different character. Consider a single rule of the form (2). It has the longest subject in $\mathcal{R}$, so no derived impliation will be longer than it. The analysis of section 2.3.1 stated that with this rule, the number of derived implications is at worst $O(n^2\mathcal{C})$. This bound can be achieved as follows:

Given rule (1), names of any length from 1 to $\mathcal{C}$ are possible. For each name of a fixed length, using (3) is it possible to replace A with any $\mathtt{X}_i$, generating n implications. This gives a total of $n\mathcal{C}$ implications generated thus far of the form:

$$\mathtt{K}_0 \rightarrow \mathtt{K}_1\ \mathtt{X}_i\ \mathtt{A}\ \mathtt{A}\ \ldots\ \mathtt{X}_j \qquad (1 \leq i, j \leq n)$$

Given one of these implications, combining it with a rule of the form (4) will take $\mathtt{K}_1$ to some $\mathtt{K}_i$. Since there are n possible keys, an equal number of implications are generated as a result.

In total — $n\mathcal{C}$ possible implications $\star$ n choices of keys gives $O(n^2\mathcal{C})$ derived implications. Since there are n rules of type (2), the total number of implications derived (and thus, the size of Γ) is $O(n^3\mathcal{C})$.

2.4 SPKI/SDSI Constraints/Extensions

The previous running time analysis ignored several features of SPKI/SDSI certificates that would have complicated the rules framework given in section 2.2.

2.4.1 Threshold Subjects

Threshold subjects[5] is a feature in SPKI/SDSI that requires multiple principals to sign a request. There are situations that a request should require several signatures. For example, Mocha-Cola is very protective of the formula for its soda. Assume that one server holds the secret formula. It would not be a good idea to allow any single

[5]Since, in SPKI/SDSI, only subjects can contain threshold names, threshold certificates are equivalently referred to as threshold subjects.

person access, in case they are spying for Nasti-Cola. So in order to gain access, several persons must agree to release the secret. Say, two of the three vice presidents of Mocha-Cola must agree to release the formula. This is where threshold subjects come into play. All that is needed is one certificate that requires a minimum number of principals signing the request to be valid — in the case above, the certificate would require at least two vice-presidents to sign the request for the secret formula.

The ability to have threshold subjects is a desirable property of SPKI/SDSI, but it is also not clear how to cast these certificates in the theoretical framework of section 2.1.5. It turns out to be possible, but the combination rules must be adjusted to account for the nature of the threshold certificate.

Each threshold certificate is in essence a juxtaposition of several distinct certificate chains that share a common beginning sequence. So, if we break the threshold subject into distinct certificates, they can be added as initial rules into $\mathcal{R}$.

For example, consider the following threshold certificate:

$$\langle \mathtt{K}_{moche},\ \Theta_2(\mathtt{K}_{moche}\ \mathtt{vp1},\ \mathtt{K}_{moche}\ \mathtt{vp2},\ \mathtt{K}_{moche}\ \mathtt{vp3})\rangle$$

The Θ_2 means that two of the three subjects in the parentheses must lead to keys that sign the request. Thus, the above certificate means that Mocha-Cola requires two of three vice presidents to sign the request. This certificate currently does not fit into the implication framework at all. But, if this rule is replaced by:

$\mathtt{K}_{moche} \rightarrow \Theta_2(\mathtt{K}_{t_1},\ \mathtt{K}_{t_2},\ \mathtt{K}_{t_3})$

$\mathtt{K}_{t_1} \rightarrow \mathtt{K}_{moche}\ \mathtt{vp1}$

$\mathtt{K}_{t_2} \rightarrow \mathtt{K}_{moche}\ \mathtt{vp2}$

$\mathtt{K}_{t_3} \rightarrow \mathtt{K}_{moche}\ \mathtt{vp3}$

The original certificate has the subjects replaced by newly generated, dummy keys. Those keys form the issuers of new implications. The original subjects in the threshold become the subjects of these new implications. The reason for replacing the original subjects with dummy keys is to ensure proper closure. Each implication in Γ must have at most one name in the issuer. If the subjects of the threshold certificate had longer names, there would be no implication that could completely match the subjects, even if a valid implication can be derived.

Now, temporarily ignoring the threshold certificate, the rules look like any other rules in Γ. The dummy keys cannot be the prefix of any other implication in Γ, because no other implication has the key as a subject. Therefore, they will only appear as issuers in Γ. If there exists a chain from the threshold subjects to the signers of the request, it will be found.

Thus far, the modified threshold certificate has been ignored, so there is no way to complete the chain that might have a threshold certificate within it. The key is to maintain some extra state that will "keep track" of the threshold certificates, and generate new implications to if enough of the threshold subjects lead to a common subject. Continuing the above example, if the following implications existed in Γ:

$\mathtt{K}_{t_1} \rightarrow \mathtt{K}_*$

$\mathtt{K}_{t_3} \rightarrow \mathtt{K}_*$

Then the threshold of the certificate would be satisfied, allowing the creation of the implication:

$\mathtt{K}_{moche} \rightarrow \mathtt{K}_*$

This allows a certificate chain in which several authorities have to independently authorize a principal, in order to get the request. Anyone who wants the formula would have to get authority from at least two vice-presidents.

The resultant impact on the size complexity of Γ is turns out to be small. The definition of n must now include the number of rules added due to the transformation of the threshold certificates.

2.4.2 Multiple Signers

Given the ability to have threshold certificates, it is possible in SPKI/SDSI to have multiple keys sign a request. It is not always necessary for the names in a threshold to resolve to one key. In fact, it may be *desirable* that more than one key sign a specific request. In the Mocha-Cola scenario, instead of delegating authority to one key, all of the vice-presidents themselves can sign the request.

In the framework of section 2.1.5 however, it is not clear how to represent multiple signers if all that is in Γ are implications with a single issuer and a single subject.

One method is to find all of the implications that have a signer as its subject. Then, once these implications are found, search among them to find one that has the correct issuer. This is a possible, but cumbersome process. The ideal situation would be to look for exactly one implication in Γ that would state whether the chain is possible or not.

A better method is to have a single key as the subject. To accomplish this, we generate a "dummy" subject, $\mathtt{K}_*$. We then take each requested signer, generate an implication of the form $\mathtt{K}_{signer} \rightarrow \mathtt{K}_*$ and include them in $\mathcal{R}$. Now, compute Γ. If the implication $\mathtt{<ACL>} \rightarrow \mathtt{K}_*$ exists, then at least one of the signers is authorized to make the request.

Since the number of implications added by multiple signers does not depend on the other rules, the only change to the bound is to change the definition of n. Now $n = |\mathcal{R}| + |\mathcal{K}|$, where $\mathcal{K}$ is the set of keys signing the request.

2.4.3 Def/Auth Split

The *def/auth split* refers to the seperation of definitions and authorizations in SPKI/-SDSI. The certificate grammar is identical, but the interpretation of the certificate (depending on the issuer) is very different. As was stated in section 1.4, authorization certificates delegate some authority to a subject, but do not define any names. Definition certificates perform name bindings, but convey no authority. The distinction places restrictions on the ability to combine certificates — restrictions that are not represented in the above framework.

Working Subject Requirements

In SPKI/SDSI, the distinction between naming and authorization forced certain restrictions on the structure of a sequence of certificates— authorizations can only occur in key space. What this means is, when an authorization is reached in a certificate chain, the working subject at that point must be a key.

Consider the following certificates:

$$\langle \mathtt{K}_0, \mathtt{K}_0\ \mathtt{faculty\ secretary} \rangle \tag{2.8}$$
$$\langle \mathtt{K}_0\ \mathtt{faculty}, \mathtt{K}_0\ \mathtt{rivest} \rangle$$
$$\langle \mathtt{K}_0\ \mathtt{rivest}, \mathtt{K}_{rivest} \rangle$$

At this point in the reduction, the working subject is $\mathtt{K}_{rivest}$ `secretary`. It is not allowed at this point to enter as the next certificate:

$$\langle \mathtt{K}_{rivest}, \mathtt{K}_{elien} \rangle \tag{2.9}$$

It is possible, given (2.9), to use $\mathtt{K}_{rivest}$ to talk about my secretary. This ability might be desirable, but could be very damaging if unintentional. It is possible to get the same effect, by issuing $\langle \mathtt{K}_{rivest}$ `secretary`, $\mathtt{K}_{elien}$ `secretary`$\rangle$, but now the delegation is explicitly defined.

Notice that along with the issuer having at most one name, this restriction removes an ambiguity in the reconstruction of certificates from working subjects. For example, take these two working subjects:

$$\mathtt{K}_0\ \mathtt{Alice\ Bob\ Charlie} \tag{2.10}$$
$$\mathtt{K}_1\ \mathtt{Alice\ Bob\ Charlie} \tag{2.11}$$

Without the restrictions above, there are several different certificates that can lead (2.10) to (2.11). For example, two certificates, $\langle \mathtt{K}_0\ \mathtt{Alice}, \mathtt{K}_1\ \mathtt{Alice} \rangle$ and $\langle \mathtt{K}_0, \mathtt{K}_1 \rangle$ are both perfectly valid. However, with the restriction that keys can only give authorizations at key nodes, the ambiguity goes away. Since authorizations can only occur at key nodes, $\mathtt{K}_0$ cannot give any authority to $\mathtt{K}_1$ at that point in the chain, so $\langle \mathtt{K}_0, \mathtt{K}_1 \rangle$ cannot be the certificate. Also, since the issuer must have at most one name, the certificate $\langle \mathtt{K}_0\ \mathtt{Alice\ Bob}, \mathtt{K}_1\ \mathtt{Alice\ Bob} \rangle$ is not legal either. This leaves the certificate $\langle \mathtt{K}_0\ \mathtt{Alice}, \mathtt{K}_1\ \mathtt{Alice} \rangle$ as the only legal alternative.

Turnstiles

What the def/auth split means for certificate discovery is that Γ cannot contain any implications that violate this principle. However, the comparison rules do not preclude this from happening. This split must be encoded into the rules somehow so that they can be only combined legally. What is introduced into the rule is an "authorization marker" ('$\dashv$', also called a turnstile). The turnstile is treated as another character, but has the effect of eliminating combinations that violate the def/auth split. By adding the marker at the end of both the issuer and subject of authorizations, combinations of implications are only allowed if the markers match, i.e. when the subject of an authorization has resolved to a key. Using the above example, the combination would not be possible if (2.8) and (2.9) were replaced by:

$$\mathrm{K}_0\ \text{`}\dashv\text{'} \rightarrow \mathrm{K}_0\ \texttt{faculty secretary}\ \text{`}\dashv\text{'}$$
$$\mathrm{K}_{rivest}\ \text{`}\dashv\text{'} \rightarrow \mathrm{K}_{elien}\ \text{`}\dashv\text{'}$$

Notice that nothing of the algorithm has changed, just the encoding of the authorizations. The rule bound has not changed at all. In fact, fewer rules would be generated than without the restriction. All that remains in the certificate tuple that has an impact of certificate discovery is the control of propagation of authorization.

2.4.4 Propagation

Propagation control has the effect of disallowing chains that could be legal otherwise. The issuer of every authorization certificate has the ability to decide whether or not its subject can delegate any authority to another subject. So, even if the chain looks valid from the standpoint of the section 2.4.3, it might still be possible that the chain is not valid, because an issuer did not allow its subject to further delegate authority, but the subject attempts to anyway.

Let's revisit the "authorization marker" concept used to disallow authorization certificates that were not at key nodes within the chain. A '$\dashv$' is placed after the issuer and subject of every authorization certificate. This forces the issuer to combine

only if it is following a subject that is the same thing, i.e. the subject has collapsed to a key.

Now, placing the '⊣' after the issuer has a clear purpose. It prevents the issuer from combining out of turn — only when the working subject has reduced to a single key. At this point, we are in key space, and an authorization should follow. However, if the previous authorization certificate disallowed propagation, no certificate should follow at this point. In effect, the '⊣' following the subject controls whether another authorization may follow the current one — it controls propagation. Therefore, in order to add propagation control to the sequence, it suffices to restrict the use of the '⊣' character only to the subjects in which the propagation is allowed. Notice, this does not violate the requirement of key node only authorizations. The issuers of the authorization certificates can still only combine at key nodes, it is just some key nodes cannot be followed.

Again, the algorithm has not changed, no new rules have been added, so the current bound remains in effect. However, with these restrictions, the worst-case bound seems more an extreme bound than an average one, given the number of restrictions on which two rules can combine.

Chapter 3

Implementation

In order to validate the framework developed in chapter 2, several implementations of the certificate discovery engine have been built. The first version, written in Perl, implements all of the constraints in section 2.4. This is because of Perl's rapid prototyping ability. It is much easier to think about the concepts involved when one doesn't have to worry about memory management, complex data structures, and other elementary functions. The complete S-expression notation is also cumbersome, with potentially very long keys and names in the issuer, subject and signatures. Thus, a simpler notation was developed to capture the relevant parts of the S-expression — the issuer, subject, thresholds, and propagation, without having to deal with long keys and key hashes.

The second implementation is written in C. This implementation is intended to work with full SPKI/SDSI certificates. It is layered on top of the sdsi2 library, developed by Matthew Fredette working under Professor Rivest. The library is a re-implementation of his master's thesis, an implementation of a SDSI 1.0 server and object manager [4]. The C certificate discovery engine does not handle threshold certificates or multiple signers, but given the framework and the Perl implementation as a guide, the remaining features should be straightforward to complete.

3.1 Theory vs. Implementation

In the framework of section 2.1.5, there are several details that the theory relies upon:

1. The construction of Γ. It is assumed that Γ will just be created and contain the necessary implications. However, creating the closure requires an iterative generation of rules. In order to generate the new implications, extra state is necessary to keep track of potential combinations.

2. Γ is a set of implications, which means that every implication in the collection must be unique. The ability to verify that a new rule is unique could, at worst, have a running time of $O(|\Gamma|^2)$. Since the assumption is that the time to get information is approximately the size of Γ, an order of magnitude difference is unacceptable.

3. The generation of the final sequence. There is nothing in the framework that discusses generating the certificate chain once the desired implication is found. It is assumed that once the proper implication is found, enough information can be associated with it to construct the certificate chain.

3.1.1 Γ Construction

Since Γ is a set of implications, each potential implication that gets generated must be checked against Γ to ensure that duplicate entries do not exist. The reason is that it does not matter *how* a particular implication is derived, simply that it *exists*.

A naive method to implement Γ would be to use a linked list. However, the time required to search for an element is on the order of Γ. This would give a resultant running time of order $|\Gamma|^2$, which is extensive.

A better method would be the use of a hash table whose keys are the implications themselves. This allows for near constant-time searching and insertion of new implications. This eliminates the order of magnitude difference in total execution time.

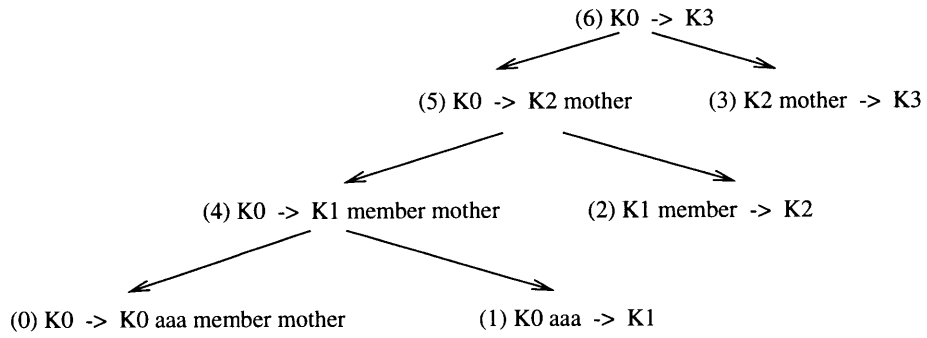

Figure 3-1: Backpointer Directed Acyclic Graph for Simple Certificate Chain.

3.1.2 Certificate Chain Generation

It turns out that the generation of the certificate chain once the proper implication is found takes very little extra information. With most implications, all that is required are two *backpointers* — references to the implications that generated it. Backpointers for threshold generated implications are handled slightly differently, but the effect is identical. Recursively following the backpointers, one can build a *dag* (directed acyclic graph) of combinations necessary to show how the implication was constructed.

In the SPKI/SDSI sequence structure, there is the ability to define *named subsequences* of certificates. A named subsequence looks exactly like a full certificate sequence, but it can be given a label. This allows for the use of a subsequence in several places without the necessity of specifying all of the certificates repeatedly.

This maps well to the backpointer dag. Each implication (node) is a named subsequence of all that was needed to generate it. Therefore, a straightforward mapping from the dag to a certificate chain exists; each node is a named subsequence of two elements — the labels of the two nodes the backpointers reference (a similar rule holds for thresholds). Figure 3-1 show this. It is a backpointer dag for a simple certificate chain. The numbers in parenthesis preceding them are unique identifiers.

The corresponding sequence for Figure 3-1 is:

```
(sequence
 (def rule0 cert0)
 (def rule1 cert1)
 (def rule4 (sequence rule0 rule1))
```

```
 (def rule2 cert2)
 (def rule5 (sequence rule4 rule2))
 (def rule3 cert3)
 (def rule6 (sequence rule5 rule3))
rule6)
```

The numbers after the `rule` labels correspond to the identifiers in the dag. Starting from the bottom, `rule6` is composed of `rule 5` and `rule 3`, just like implication 6, the top node, is composed of implications 5 and 3. Following recursively, the entire structure maps exactly to the sequence. The leaves of the dag correspond to the actual certificates that are needed for the sequence. Section 3.2.3 describes the process that generates the above sequence.

3.2 Perl - `certgen.pl`

The following describes the execution of the `certgen.pl` — the perl code is included in appendix A.

3.2.1 Input Specification

The input to `certgen.pl` is split into two parts:

Desired Rule - what rule should the code look for after it has completed generating all of the implications. It has the form of:

$$\mathtt{K}_{Issuer}\ \mathtt{[<name>]}\ \mathtt{::}\ \mathtt{K}_{Signer1}\mathtt{[,K}_{Signer2}\mathtt{,K}_{Signer3}\mathtt{,...]}$$

The name after the issuer is optional. Since it is possible to have multiple signers sign the request, the subject can be a comma separated list of all of the signers' keys. If the issuer is just a key, then the name '`-blank-`' is added after the issuer and all of the signers' keys. The name '`-blank-`' is intended to be the '$\dashv$' symbol, the turnstile character that marks authorizations.

Certificate Database - the certificates that Γ is generated from. These rules can take one of two forms:

K_{Issuer} [<name>] -> [P] $K_{Subject}$ [<name1> <name2> ...]

This is the simple certificate form. It, like the the desired rule, has an optional issuer name, and the subject can have multiple names. Also like in the desired rule, '-blank-' is added if the issuer name is missing.

K_{Issuer} [<name>] -> [P] T<val> <Subject1>:<Subject2>:...:<SubjectN>

This is the threshold certificate. <val> is the threshold value — the minimum number of subjects necessary to satisfy the certificate. <Subject*i*> is a subject as in the simple certificate case (no nested thresholds), and N $\geq$ <val>. The threshold subjects are separated by the ':' character, and are used to construct a fake rule with a dummy key issuer.

In order to ensure that the key node authorization is upheld, if the threshold certificate's issuer does not have a name after the key, '-blank-' is added to all of the issuers and subjects of the fake rules. Once the fake rules have been made, the threshold certificate is reconstructed with the dummy keys as the subjects. This is needed later for threshold combination.

The [P] in both certificate formats is the optional propagation flag. This flag affects the placement of the '-blank-' character on the subjects — if it is set, then the '-blank-' is appended to the suffix.

Each certificate is given a unique certificate id number. This number is the value that is used in the backpointer structure to refer to the certificate. The simple rules are processed as they are being read; the thresholds are held until after the rules have all been input. Once all of the rules have been entered, the dummy keys can be generated and the threshold certificates and multiple signers are processed. The rules are then ready for closure.

3.2.2 Finite Closure Generation

Finite closure begins after all of the rules have been entered and processed. The process is a while loop that has two parts, the generation of the simple implications

followed by the generation of the threshold implications. When there are no more rules generated, the while loop ends and the closure is completed.

Simple Rule Closure

The first thing that happens is the simple rule closure. This is another while loop that continues to try to generate simple implications until it cannot any longer. To aid in the process, several data structures are used:

Rules Hashtable - this hashtable is keyed by the implications — it is Γ. Any implication generated is entered here if it does not already exist. The value associated with each of the implications is its unique certificate id number. This allows for easy reference by other implications.

Mergable Rules - This is a list of the certificates (and later, the implications) that can be valid second rules in a combination — i.e. the certificates whose subject is no longer than its issuer. This is to avoid having to search the rules hashtable every time in order to determine which implications are mergable.

Subjects Hashtable - This hashtable, generated as the rules are being read, contains the information necessary to complete the combination. It is keyed by SDSI name, and contains the certificates whose subject is equal to the SDSI name or longer. For example, if $\texttt{K}_1$ `Alice Bob Charlie` had certificate id 7, the value 7 would be added to two entries into the hashtable: '`1`' and '`1:Alice`'.

The reason is, again, efficiency. The information is available as the certificates are being read, so there is no point in having to search Γ for all of the possible matches to a given name. That is why the names `Bob` and `Charlie` do not actually appear as entries in the table — no issuer can ever have more than one name following it.

Given these parameters, the rule closure begins. It iterates through the list of mergable implications, trying to find an implication that will match it. It finds the issuer from the certificate id, and if that issuer has an entry in the subject hashtable, there

are implications that can merge with it. Given those ids from the subject hashtable, the program then proceeds to construct the composite implication, following the rules of section 2.1.5. If the new implication does not exist in the rules hashtable (Γ), it is added. Once the rule is added, the backpointer array (indexed by certificate id) contains the two implications needed for the combination. If, after going through the list of mergable implications, at least one new implication has been added, the loop starts again. This continues until no more new implications are added.

Threshold Rule Combination

Once the simple rule combination is complete, if there are any threshold certificates, the threshold combination begins. For each original threshold certificate, the dummy subjects are collected. Since the dummy keys only appeared in Γ as the issuers of the fake rules, they can only be the issuers of any implications that appear if Γ.

All of these rules are collected and placed in a hashtable keyed by their subjects. The values of these entries are the issuers that created them. The idea is this — if a subject has enough of the dummy keys as issuers to it to satisfy the threshold value, a new implication can be formed. This new implication has the issuer of the original threshold certificate, and the subject that had enough dummy keys as issues to it. This is an exact implementation of the method that is described in section 2.4.1.

If a new rule can be generated, then it is added — as a simple rule, because it has only one issuer and subject. All that remains is the backpointer structure for this new implication. The value is a string that has two parts, the certificate id of the original threshold certificate, and the ids of the implications that were used to generate the rule. The two pieces are separated by a ':' character, and entered into the array.

Both combinations, the simple and threshold, are run in a loop until there are no more implications generated. Once no more implications are generated, Γ is complete and all that remains is the generation of the certificate chain, if the desired rule is in Γ.

3.2.3 Certificate Chain Construction

Since the SPKI/SDSI sequence structure allows for the construction of named subsequences, each implication that was needed to generate the desired rule becomes a named sequence. This makes the traversal of the backpointers very easy — if the backpointer references an implication that was already made into a defined subsequence, just use the defined name.

So, given the id of the implication that represents the desired rule, the backpointer array entry can contain one of four possible values:

1. **Nothing** - this means that the implication is an actual certificate, and should be printed as such.

2. **-1** - This means that the implication was a fabricated rule — one that either came from the dummy keys of a threshold certificate or a signer to K_*. In either case, this implication should NOT be printed, it only exists to allow the closure to complete.

3. **Two Ids** - This is the common case, an implication was constructed from two other simple implications. For this implication, recursively generate the two sub-chains, then build a named subsequence using the names they would have generated.

4. **Several Ids separated by ':'** - this is a threshold implication. Each value corresponds to the implication needed to take its respective subject to the common subject of the implication. If the subject was not used, it has the word "`kill`" instead of a number.

The process of generating the certificate chain is basically a modified topological sort of the certificate ids. Starting with the desired rule's id, the code follows the backpointers to the combining implications. It recursively generates the sequences for those rules first, then generates the sequence for the desired implication. If both children have generated sequences, then the rule is a named sequence of the two. If

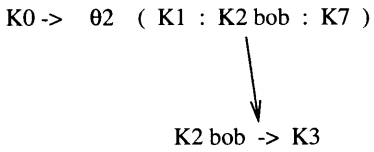

Figure 3-2: Threshold Backpointer Dag - K_1 and K_3 signed the request.

either backpointer is -1, then it should just be ignored in the generation of the desired implication. If `rule5` from the generated sequence in section 3.1.2 was a fabricated rule (its backpointer value was -1), the definition for `rule6` would instead look like:

```
(def rule6 (sequence rule3))
```

Since `rule5` was not a real certificate, it is just ignored in the generation of anything depending on it.

Threshold backpointers are simply an extension of the basic backpointer notation — the only difference is that there a number of implications needed to generate the subsequence. The code takes each backpointer, and recursively generates the rules for it, or nothing if its backpointer is -1. Once all of the dependencies are generated, the rule itself is printed.

```
(sequence
 (def rule1 cert1)
 (def rule7 (sequence rule1))
 (def rule9 (sequence rule7))
 (def rule10 (sequence
  (cert 0)
  (process-threshold
        done,
        (sequence rule9),
        kill)
 )
 rule10)
```

The preceding sequence is what `certgen.pl` generates given the backpointer structure in Figure 3-2. In the definition of `rule10`, `(cert 0)` the threshold certificate. It is followed by a directive that tells the verifier how to process the individual subjects.

The first subject (K_1) signed the request, so it satisfies the first subject — hence the "`done`" directive. The third subject is not used by the chain; this is denoted by the directive "`kill`". The second subject has a chain that follows it, and that is specified by specifying the sequence that follows.

3.3 C version - `fclose.c`

The C implementation, built on top of the sdsi2 library of Matthew Fredette, implements the same basic framework, with the enforcement of key node authorizations. Once `certgen.pl` was completed, the concepts and issues necessary to implement Γ had been fully worked out. However, the one feature `certgen.pl` lacked was the ability to work with true SPKI/SDSI certificates. This, along with the fact that the majority of SPKI/SDSI implementations will be in C, motivated a re-implementation of the discovery engine. The remaining sections talk about the issues in porting the engine to C. The new implementation of the engine relied heavily on the underlying sdsi2 library, not only for the ability to manage a database of implications, but the ability to search the database for possible matches. The library will be available under the SDSI web page: <http://theory.lcs.mit.edu/ cis/sdsi.html>.

3.3.1 Input Specification

The input to `fclose.c` was a database of certificates in the form of a sdsi2 library certificate cache. Since the engine is concerned with determining possible chains, and the sdsi2 library provided a rich mechanism for managing certificates, a cache is assumed to have already been created and is taken as input. This cache is copied to a new cache where the closure is performed.

Since the sdsi2 library caches are a fixed size, the ability to place a maximum size using the bound in section 2.3.2 was invaluable. Although the actual size of the closure was typically much smaller than the capacity, given a small database, the assumed size was not unreasonable (approx. 30 kilobytes for 800 certificates).

3.3.2 Finite Closure Generation

Since `fclose.c` was written using the sdsi2 library, the data structures used to implement Γ changed significantly. Revisiting the pieces needed in computing Γ in `certgen.pl`:

Rules Hashtable - This was implemented by the sdsi2 library cache. The sdsi2 library had the ability to, given certain criteria, perform searches on the cache. This enabled the ability to verify uniqueness by querying the cache for a particular certificate. The cache also referenced the certificates contained within it by a unique identifier, enabling efficient referencing ability once a certificate was found.

Mergable Rules - This was a linked list of implications that was maintained in much the same way as the array of mergable implications in `certgen.pl`. Each new implication that was generated was checked to see if was a valid second implication. If it was, it was added to the list.

Subject Hashtable - This role was also served by the cache. The cache allowed for the ability to perform searches on subjects that matched a name and anything longer. This is not as efficient as a dedicated table for the purpose, but the trade-off was the decreased complexity of the system, since the functionality already existed in the sdsi2 library.

Backpointer Storage

In order to store the backpointer information, when new implications are added to the cache, they contain extra information besides the issuer and subject. Within a SPKI/-SDSI certificate exists the ability to define an S-expression that gives information about the issuer. It is called `issuer-info`, and it looks like this:

```
(issuer-info <bytestring1> <bytestring2> ...)
```

The `bytestring`'s are strings that can have any value. It is here that the values of the backpointers are stored. So, if an implication was derived from two other implications, with the IDs 3 and 5, its `issuer-info` field would look like:

```
(issuer-info "3" "5")
```

'⊣' Storage

Like the basic combination rules, there is no implicit mechanism in the sdsi2 library that enforces key node authorizations. So, like the backpointer structure, the certificate needs to be able to carry the information inside the S-expression. Like the `issuer-info` there also exists a `subject-info` that has the same format:

```
(subject-info <bytestring1> <bytestring2> ...)
```

Here is where the propagation information for the subject is kept. The issuer doesn't need any extra information — if it is just a key, then the certificate is an authorization. If the certificate is an authorization, the issuer has an implicit '⊣' attached. Only the subject needs to maintain the information of whether a '⊣' trails the subject. If the resultant implication needs to have a '⊣' follow it, the following `subject-info` S-expression is added:

```
(subject-info "propagate")
```

This tells the code that performs the certificate merging that the implication has a trailing '⊣'.

Notice that the `issuer-info` and `subject-info` S-expressions are being used even on input certificates. When the certificates are copied to a new cache to perform closure, they are replaced by implications that reference the original cache. Thus, there is no mangling of the original certificates— only the relevent information is copied.

3.3.3 Sequence Generation

Once the closure was completed, and the desired implication was found, a depth-first search identical to the search in `certgen.pl` is performed. Since there were no multiple signers, if the certificate has an `issuer-loc`, it must be a derived implication. The generated sequence looks exactly like the sequence that gets generated by `certgen.pl`, the only difference is that instead of `(cert` i`)`, the actual certificate is printed.

Chapter 4

Conclusions

4.1 Discovery in SPKI/SDSI

As Chapter 2 has shown, the SPKI/SDSI infrastructure has, given a database of certificates, the ability to efficiently determine whether a certificate chain is possible. As stated before, there is evidence that similar problems in other PKIs cannot be solved as efficiently. The remaining section talks about possible directions the research can take.

4.2 Future Work

The following are currently open issues with the theory of certificate discovery:

- What happens if the engine cannot find the required chain? There is nothing in the theory currently that talks about the possibility of the implication not being found. Perhaps it is possible to search a common database of certificates to gain more information. Or the server that holds the acl might give hints as to where to search for the missing certificates. It might even be possible to determine the exact implication necessary from Γ itself. Future work could possibly develop a framework that would be automated enough to search for credentials away from the machine.

- Is it possible to perform a distributed finite closure? This problem is a little different from trying to find the chain using the network. This is more oriented towards discovery services that might not want to keep thousands of certificates in one place. It might be possible to partition the certificates and create closures using little network bandwidth.

Appendix A

certgen.pl

```
#!/usr/athena/bin/perl

# Closure 2.0 - redone to implement the new style of rule maintainance
# - Jean-Emile Elien {elien@theory.lcs.mit.edu, hbm@alum.mit.edu}
# May 1998

# Warning - this code is pretty well documented, and (I hope) pretty
# self-explanitory.  However, it assumes you know Perl, SDSI naming and
# combination rules, and understand the overall picture of closure of a
# set of certificates.  With that said --- have fun.

# Declarations of global variables used

# Unique certificate id assigned to each rule read
$cert_id = 0;

# String corresponding to the Implication to be searched for
$desired_rule = "";

# Array [by cert_id] of strings pointing to rules need to compose this rule
# if the value is -1, then the rule is a ficticious rule, and should not
# appear in the final printed sequence.
@backpointer_certs = ();

# List of mergable certs (ones where |subject| <= |issuer|
@meragble_certs = ();

# Table of strings (keyed by key and key:name, value is a string of cert_ids)
# who's subject is the hash key or longer.  So, if 'k0 bob' was the key, the
# rule that had the subject 'k0 bob alice fred' would be in the value.
```

```
%subject_equal_or_longer;

# Arrays of the respective parts of eeach rule [index'd by cert_id]
@issuer_key = ();
@issuer_name = ();
@subject_key = ();
@subject_names = ();

## Special arrays which hold threshold certificates, handled differently than
## other rules.
# The string of the threshold rules
@threshold_certs = ();
# the corresponding cert_id in the sequence.
@threshold_ids = ();
# the value of the threshold needed to pass.
@thresholds = ();

# used to make dummy keys - maximum value so far found of keys
$max_key = 0;

# read in the first line, for the desired rule
chop ($_ = <>);

# Basically, looking for line of the form:
# K<val> [<name>] :: K<val> [,K<val2>,K<val3>,...]

/^[Kk](\d+)(\s+(\w+))?\s+::\s*(.*)/ ||
    die "Error: Unable to parse desired rule:\n\t<$_>";

local ($ik, $in, $sk) = ($1, $3, $4);

$max_key = $ik+1 if ($ik >= $max_key);

# see if there are multiple signers in the requested line and parse them
@signers = split(/,/, $sk);

for ($i = 0; $i < @signers; $i++) {
    $signers[$i] =~ /^[Kk](\d+)\s*$/ ||
die "Error: unable to parse signer:\n\t<$signers[$i]>";
    $signers[$i] = $1;
    $max_key = $1+1 if ($1 >= $max_key);
}

# see if -blank- characters are needed for the issuer and subjects (if it is
# an auth)
```

```
unless ($in) {
    $in = "-blank-";
    for ($i = 0; $i < @signers; $i++) {
$signers[$i] .= " -blank-";
    }
    $sk = "* -blank-";
} else {
    $sk = "*";
}


# Construct the desired rule to look for, and print it.
$desired_rule = "$ik $in:";
$desired_rule .= (@signers > 1) ? $sk : "$signers[0]";

print "Desire Rule => 'K$ik $in' :: 'K",join("', 'K",@signers),"'\n\n";

# read in the rest of the rules, holding off on adding the threshold certs
while (<>) {

    # skip commented out lines
    next if (/^\#/ || /^\s+$/);

    chop if (/\n$/);


    if (/^[Kk](\d+)(\s(\w+))?\s+->\s*([Pp])?\s*[Kk](\d+)(\s(.*))?/) {
# matched single rule regex

# parse the rule for the necessary pieces, let's say:
# $1 - issuer key
# $3 - optional issuer name
# $5 - subject key
# #7 - optional subject names (space seperated)

local ($ikey, $iname, $skey, $snames) = ($1, $3, $5, $7);

# get the maximum key value to generate unique keys
$max_key = $ikey+1 if ($ikey >= $max_key);
$max_key = $skey+1 if ($skey >= $max_key);

# add blank characters necessary for syntax correctness
unless ($iname) {
    $iname = "-blank-";
    $snames = $7 ? "$7 -blank-" : "-blank-";
```

```
}

# if duplicates, signal error.
die "repeated certificate:  <$_>\n"
    if ($rules{"$ikey $iname:$skey $snames"});

# add rule to the list of rules
&add_rule($cert_id, $ikey, $iname, $skey, $snames);

    } elsif (/^[Kk](\d+)(\s(\w+))?\s+->\s*([Pp])?\s*[Tt](\d+)\s+(.*)/) {
# matched the threshold regex

# $1 = issuer key
# $3 = optional issuer name
# $5 = threshold value
# $6 = threshold subjects (':' seperated)

local ($ik, $in, $thval, $ths) = ($1, $3, $5, $6);

# add to the list of threshold subjects currently acquired
# assign them to the same cert_id as normal, to keep the certificate
# order correct.

$max_key = $1+1 if ($1 >= $max_key);

# validate the threshold subjects:

local (@subjects) = split(/:/, $ths);

# If the number of subjects is less than the threshold, fail.
die "Invalid rule (threshold too high) '$input'\n"
    if (@subjects < $thval);

# make sure that they all at least have a key
# grep for valid rules and check to make sure the length
# is identical to the original list.
die "Invalid rule (invalid subject) '$input'\n"
    unless (@subjects == grep(/^[Kk]\d+(\s+(.*))?$/, @subjects));

# go through subjects and get the maximum value, in order to
# be able to generate dummy keys.
foreach (@subjects) {
    /^[Kk](\d+)/;
    $max_key = $1+1 if ($1 >= $max_key);
    $_ .= " -blank-" unless ($in);
```

```
}

local ($iname) = ($in) ? $in : "-blank-";

# hold onto the threshold certs - will be dealt with later.

# re-join the thresholds into an easily breakable format.
push(@threshold_certs,"$1|$iname|".join(':',@subjects));
push(@threshold_ids, $cert_id);
push(@thresholds, $thval);
    } else {
die "Unable to parse line:\n\t<$_>";
    }

    $cert_id++;
}

# done with input file
close(ARGV);

## Add the special case rules into the rule list: thresholds and
## multiple signers.  Thresholds are handled by taking each of the
## subjects and adding them into the list with a dummy key issuer,
## then performing the closure on that.  Multiple signers are handled
## by having them all delegate to a special key, K*, that will be the
## single key searched for.

# Add the threshold subjects in: one by one, assigning them dummy key issuers
# and placing them into the rule list.
if (@threshold_certs) {

    for ($i = 0; $i < @threshold_certs; $i++) {

print "\n";
# split the previously reconstructed subjects
local($ik, $in, $sks) = split(/\|/,$threshold_certs[$i]);
local(@sks) = split(/:/, $sks);

# create a new dummy key to be the issuer to each subject
foreach (@sks) {
    /^[Kk](\d+)(\s+(.*))?$/;
    # preserve the -blank- name for the issuer, if the subject has it.
    # FIXME - when propogation is added, need a better way to denote -blank-
    local ($newsubject) = ($in eq "-blank-") ? $in : "";
```

```
    &add_rule($cert_id, $max_key, $newsubject, $1, $3, "{-1}");

    # this rule never will make it into the actual sequence,
    # so has no backpointer.
    $backpointer_certs[$cert_id] = -1;

    # replace subject with the dummy key generated
    $_ = "$max_key $newsubject";

    $cert_id++;
    $max_key++;
}
# put back in the array the threshold with the new dummy subjects.
$threshold_certs[$i] = join('|', $ik, $in, join(':',@sks));
    }
}

# If there are multiple signers, then add a dummy key K* to the mix, and have
# all of the signers point to it.
if (@signers > 1) {

    print "\n";
    foreach (@signers) {
local($key, $name) = split(' ');
local($useblank) = $name eq "-blank-";

&add_rule($cert_id, $key, $name, "*", ((useblank) ? $name : ""), "{-1}");
$backpointer_certs[$cert_id] = -1;
$cert_id++;
    }
}

## Now compute the generated rules.  These rules are derived from the
## initial set by transitive rules, and a subject length maintaining
## rule.  Once they are generated, if there are threshold subjects,
## see if any implications have the right issuers and enough subjects
## to satisfy the threshold value.  Add them as rules, and loop
## until no new rules are generated.

print "\n";

# boolean flag to denote if new threshold rules are generated.
$threshold_rules_generated = 1;

# loop while threshold rules AND closure generates rules.  This should
```

```
# not be an OR, because I only want to go through the trouble of
# trying to find more threshold rules if there were BOTH threshold
# rules and closure rules the last iteration.  If no threshold rules
# get generated, then there is no need to call the closure again, no
# new rules have been added.  If that is true, there is no need to go
# into the loop because once all of the threshold rules have been
# added if no combinations have taken place in the closure.  No more
# combinationscan occur w/o more rules generated by the closure.
while ($threshold_rules_generated &&
       &closure(*rules, *subject_equal_or_longer, *mergable_certs) ) {

    $threshold_rules_generated = 0;

    # Now, iterate through all of the threshold rules, and collect the
    # dummy keys that they point to.  Then take the list of rules, and
    # find all of the rules that have issuers that match the dummy
    # keys.  If they do, try to match common subjects.  If enough of
    # the dummy issuers mathc the same subject, then a new rule can be
    # created with the issuer of the real threshold cert pointing to
    # that subject.

    for ($i = 0; $i < @threshold_certs; $i++) {
local ($ik, $in, $sks) = split(/\|/, $threshold_certs[$i]);
local (@sks) = split(/:/, $sks);
local (%mark);

# Foreach dummy key, grep through the list of rules that match
# that issuer.  Once that issuer is found, build a hash table,
# keyed by subject, and add that issuer to the list.
foreach $key (@sks) {
    foreach (grep(/^$key/, keys %rules)) {
split(/:/);
$mark{$_[1]} .= "$key|$rules{$_}:";
    }
}

# Now go through each subject.  If the number of issuers is
# greater than the threshold value, then it is possible to add
# this cert.  If the rule with the threshold cert's issuer and
# this subject don't already exist, enter them into the rule
# list, and create the proper backpointer structure.
foreach (keys %mark) {
    if (split(/:/,$mark{$_}) >= $thresholds[$i]) {

/^(\d+|\*)\s+(.*)$/;
```

```
next if ($rules{"$ik $in:$1 $2"});
&add_rule($cert_id, $ik, $in, $1, $2);

# this uses a bit of trickery.  @_ refers to the array
# created by the above split() on the issuers for this
# subject.  Go through the list of subjects of the the
# threshold cert, and see if this dummy key was used
# to make the threshold.  If it was, then use it,
# otherwise 'kill' this subject.
local (@trules);
foreach $elem (@sks) {
    local (@elems) = grep(/$elem\|/, @_);
    if (@elems) {
$elems[0] =~ /\|(\d+)$/;
push(@trules, $1);
    } else {push (@trules, "kill");}
}

$backpointer_certs[$cert_id] = "$threshold_ids[$i]:" .
    join(' ', @trules);
$cert_id++;
$threshold_rules_generated = 1;
    }
}
    }
    print "\n";
}

print "Total number of rules: $cert_id\n";
print "\nDesired rule found: " . (($_ = $rules{$desired_rule}) ? "yes ($_)" : "no

exit unless ($_);

print "(sequence\n";
&print_chain($_);
print "rule$rules{$desired_rule})\n";

## Subroutines
# ------------

## print out the chain: takes as an argument, the rule that should be
# printed.  Basically, _print_chain is a depth-first search that
# follows backpointers until they can't go back any farther (either
# they are original certs, or they are 'created rules'.  In the
# former, print a rule that is just the cert, and in the latter, print
```

```
# nothing.  Mark reaching this node, and return.  Once all the
# children of a node has printed, print a new rule that is your
# cert_id, and it was composed of the cert_ids of the backpointers.
# Keep going until done printing the whole tree.
sub print_chain {
    sub _print_chain {
local ($node, *explored) = @_;

# already visited this node, go back, but tell caller to refer to this rule.
return 1 if ($explored[$node]);
# not a real rule, fabricated, so return, but tell caller not to refer to this rule
return -1 if ($backpointer_certs[$node] == -1);

if ($_ = $backpointer_certs[$node]) {
    if (index($_, ':') == -1) {
# simple backpointer - no thresholds.  Has two numbers: "$first $second"
local ($left, $right) = split;
local (@printable);
# only push onto printable stack if child doesn't return -1.
push (@printable, $left)
    if (&_print_chain($left,*explored) >= 0);
push (@printable, $right)
    if (&_print_chain($right, *explored) >= 0);

# note, if @printable is empty, then no need to refer to this rule either.
if (@printable) {
    print " (def rule$node (sequence rule", join(' rule',@printable), "))\n";
} else {return -1;}
    } else {
# this is threshold backpointer - a little
# trickier. Split by ':' into 2 pieces: the original
# cert_id of the threshold, and the rules needed to
# complete this threshold (those are seperated by ' ')
split(/:/);
local (@process_rules) = split(' ', @_[1]);
foreach $subrule (@process_rules) {
    # skip the subject that was killed.
    next if ($subrule eq 'kill');

    # if the subject has nothing following it, then it is done.
    $subrule = "done"
unless (&_print_chain($subrule, *explored) >= 0);
}
print " (def rule$node (sequence\n  (cert $_[0])\n";
```

```
# basically, take each rule, wrap them in appropriate sequence, and
# print them all together.
foreach (@process_rules) {
    s/^(\d+)$/\(sequence rule$1\)/;
}
print "  (process-threshold\n\t",join(",\n\t",@process_rules),")\n";
print " )\n";
    }
} else { # no backpointer define at all - simple certificate, make a simple rule.
    print " (def rule$node cert$node)\n";
}
$explored[$node] = 1;
$node;
    }

local ($node) = @_;
local (@explored);
&_print_chain($node, *explored);
}

## compute closure of given set of rules.  basically, while there has
# been rules generated, keep trying to combine the rules to generate
# new rules.  Once no more rules have been generated in an iteration,
# return whether or not any rules at all have been generated.
sub closure {
    local (*rules, *subject_table, *mergable) = @_;
    local ($rules_generated, $first_rule, $second_rule) = (1, "", "");
    local (@certs_merged);

    local ($any_rules_generated) = 0;

    while ($rules_generated) {

$rules_generated = 0;
print "\n";
local ($isname, @certs, @new_mergable);
foreach $second_rule (@mergable) {

    # construct issuer and subject of the second rule.
    $issuer = $issuer_key[$second_rule];
    $issuer .= ':'. $issuer_name[$second_rule]
        if ($isname = $issuer_name[$second_rule]);
    $subject = $subject_names[$second_rule];

    # remove the rules that this rule has already combined with
```

```
    # .. gotten from perl4 manual - finds the difference of two arrays
    local(%mark);
    grep($mark{$_}++, split(' ', $certs_merged[$second_rule]));
    local(@certs) = grep(!$mark{$_}, split(' ',$subject_table{$issuer}));

    foreach $first_rule (@certs) {

# first, must construct new subject name.
local(@new_subject) = split(' ', $subject_names[$first_rule]);

# If there is a name in the issuer, it must have matched the first name
# in the subject.  Remove it.
shift(@new_subject) if ($isname);

# if there is a name in the subject, add it in front.
unshift(@new_subject, $subject) if ($subject);

# contruct the new cert, and see if it hasn't been used
local ($new_rule) = "$issuer_key[$first_rule] $issuer_name[$first_rule]";
$new_rule .= ":$subject_key[$second_rule] ".join(' ', @new_subject);

# if rule already exists, then skip.
next if ($rules{$new_rule});

# add rule to the list of rules
&add_rule($cert_id,
  $issuer_key[$first_rule], $issuer_name[$first_rule],
  $subject_key[$second_rule], join(' ',@new_subject),
  "{$first_rule,$second_rule}");

# create a simple backpointer reference.
$backpointer_certs[$cert_id] = "$first_rule $second_rule";

$cert_id++;
$any_rules_generated = $rules_generated = 1;
    }

    # add these rules to the list of rules already merged.. so
    # the code doesn't have to try and merge them again.  This
    # is only a time-saver, by filtering out rules already
    # looked at.
    $certs_merged[$second_rule] .= " " . join(' ', @certs);
    undef @certs;
}
```

```
# add new mergable rules -- rules with subject no longer than the issuer.
@mergable = (@mergable, @new_mergable);
    }

    $any_rules_generated;
}

# Adds a rule to the rule list.. common piece used several times in the code
# assumes that the rule is going to be new, as different pieces of code
# handles duplicates differently.

sub add_rule {
    local ($cid, $ik, $in, $sk, $sns, $trailer) = @_;

    # give rule a certificate id to be used by the rest of the code.
    $issuer_key[$cid] = $ik;
    $issuer_name[$cid] = $in;
    $subject_key[$cid] = $sk;
    $subject_names[$cid] = $sns;

    # Add unique rule to the hash table of rules, to ensure no future
    # duplicates
    local ($rule) = "$ik $in:$sk $sns";
    $rules{$rule} = $cid;

    # quick lookup table for rules: which ids is the subject of this rule a
    # prefix of?
    $subject_equal_or_longer{$sk} .= "$cid ";

    # only need to do this for the first name, because issuers are guaranteed
    # to have at most one name.
    local(@subjects) = split(' ',$sns);
    $subject_equal_or_longer{"$sk:$subjects[0]"} .= "$cid ";

    # if this rule doesn't grow the subject, then it is a candidate to merge
    # with some other rule.
    push(@mergable_certs, $cid)
if ( ($in && (@subjects <= 1)) || (@subjects == 0) );

    print "($cid) 'K$ik $in' -> 'K$sk $sns' $trailer\n";
}
```

Bibliography

[1] Matt Blaze, Joan Feigenbaum, and Martin Strauss. Compliance Checking in the PolicyMaker Trust Management System. In *Proceedings of the 2nd Financial Crypto Conference, Lecture Notes in Computer Science*, Berlin, 1998. Springer. Available in preprint form as AT&T Technical Report 98.3.2.

[2] Ellison, Frantz, Lampson, Rivest, Thomas, and Ylonon. Simple Public Key Certificate. Technical report, IETF - Internet Draft, March 1998. Text available at: <ftp://ds.internic.net/internet-drafts/draft-ietf-spki-cert-structure-04.txt>.

[3] Ellison, Frantz, Lampson, Rivest, Thomas, and Ylonon. SPKI Certificate Theory. Technical report, IETF - Internet Draft, March 1999. Text available at: <ftp://ds.internic.net/internet-drafts/draft-ietf-spki-cert-theory-02.txt>.

[4] Matthew H. Fredette. An Implementation of SDSI - the Simple Distributed Security Infrastructure. Master's thesis, Massachusetts Institute of Techology, Cambridge MA, May 1997.

[5] Alfred J. Menezes, Paul C van Oorschot, and Scott A Vanstone. *Handbook of Applied Cryptography.* Discrete Mathematics and Its Applications. CRC Press, Boca Raton New York London Tokyo, 1997.

[6] Alexander Morcos. A Java Implementation of Simple Distributed Security Infrastructure. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1998.

[7] Michael K. Reiter and Stuart G. Stubblebine. Path Independence for Authentication in Large-Scale Systems. In *Proceedings of the 4th Conference on Computer and Communications Security*, April 1997.

[8] Ronald L. Rivest and Butler Lampson. *SDSI - A Simple Distributed Security Infrastructre.* <http://theory.lcs.mit.edu/~rivest/sdsi10.html>, April 1996.

[9] Solo et al. Internet Public Key Infrastructure X.509 certificate and CRL profile. Technical report, IETF - Internet Draft, March 1998. Text available at: <http://www.ietf.org/internet-drafts/draft-ietf-pkix-ipki-part1-07.txt>.

[10] Philip R. Zimmermann. *The Offical PGP User's Guide.* MIT Press, Boston, 1995.