Storage and retrieval of SPKI certificates using the DNS

Master's Thesis

Tero Hasu

Helsinki University of Technology	Teknillinen korkeakoulu
Department of Computer Science and Engineering	Tietotekniikan osasto
Telecommunications Software and Multimedia Laboratory	Tietoliikenneohjelmistojen ja multimedian laboratorio
Espoo 1999	

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF MASTER'S THESIS

Author: Tero Hasu

Name of thesis:
Storage and retrieval of SPKI certificates using the DNS

Date:	April 14, 1999	**Pages:**	7 + 67 + 1
Department:	Department of Computer Science and Engineering	**Chair:**	Tik-110
Supervisor:	Professor Arto Karila		
Instructor:	Pekka Nikander Lic.Tech.		

Simple Public Key Infrastructure (SPKI) certificates can be used to represent trust and policy information in a manner that allows the authenticity and the integrity of the information to be verified. These properties make SPKI certificates useful when maintaining information and system security. The reliability of a distributed system whose security management is based on certificates may entirely depend on the availability of certificates. Clearly, such a system could benefit from having a distributed, fault-tolerant certificate repository that supports flexible administration of certificate data. The SPKI specification does not define one, however.

The Domain Name System (DNS) is a distributed database that is mainly used to provide a naming service for the Internet. It can, however, be adopted for other uses as well, by adding support for new data types to name servers and resolvers.

This thesis presents a detailed description of how the DNS can be used as an SPKI certificate repository. Existing knowledge is utilized when available, and new solutions are suggested as necessary. Among other things, the naming practice of SPKI certificates is addressed, and a scheme that offers support for two-way graph search algorithms is described. Such algorithms have previously been found to be efficient when acquiring proof of authorization from distributed databases in the form of certificate chains. Evaluation of the suitability of the DNS as a certificate repository is also given in this work.

Some of the certificate storage theory was refined and applied to practice as the author implemented a DNS resolver with certificate support, and used it to retrieve SPKI certificates from the DNS. The resolver was implemented using JaCoB, a framework for cryptographic protocols. The interface and the high-level structure of the implementation are described in this thesis.

Keywords: DNS, SPKI, JaCoB, resolver, certificate, framework
Language: English

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Tero Hasu		
Työn nimi:	SPKI-sertifikaattien tallennus ja haku käyttäen DNS:ää		
Päivämäärä:	14. huhtikuuta 1999	**Sivuja:**	7 + 67 + 1
Osasto:	Tietotekniikan osasto	**Professuuri:**	Tik-110
Työn valvoja:	Professori Arto Karila		
Työn ohjaaja:	TkL Pekka Nikander		

Simple Public Key Infrastructure (SPKI) -sertifikaatteja voidaan käyttää luottamusta ja turvapolitiikkaa koskevan tiedon esittämiseen siten, että tiedon aitous ja eheys voidaan tarkistaa. Nämä ominaisuudet tekevät SPKI-sertifikaateista hyödyllisiä ylläpidettäessä tietojärjestelmien turvallisuutta. Hajautetun järjestelmän, jonka turvallisuuden hallinta perustuu sertifikaatteihin, luotettavuus voi riippua täysin sertifikaattien saatavuudesta. Hajautetusta ja vikasietoisesta sertifikaattien talletuspaikasta, joka mahdollistaa sertifikaattiaineiston joustavan hallinnan, voisi selvästi olla hyötyä kyseisenlaisessa järjestelmässä. SPKI ei sellaista kuitenkaan määrittele.

Domain Name System (DNS) on hajautettu tietokanta, jota käytetään pääasiassa nimipalvelun tarjoamiseen Internetissä. Se voidaan kuitenkin ottaa muuhunkin käyttöön lisäämällä nimipalvelimiin ja resolvereihin tuki uusille tietotyypeille.

Tämä diplomityö esittää yksityiskohtaisen kuvauksen siitä, miten DNS:ää voidaan käyttää SPKI-sertifikaattien talletuspaikkana. Olemassaolevaa tietoa on hyödynnetty mahdollisuuksien mukaan, ja uusia ratkaisuja on ehdotettu tarpeen vaatiessa. Muun muassa SPKI-sertifikaattien nimeämiskäytäntöön on otettu kantaa, ja kaksisuuntaisia graafihakualgoritmeja tukeva käytäntö on kuvattu. Kyseisenlaisten algoritmien on aiemmin todettu olevan tehokkaita valtuutuksen todistamiseen vaadittavien sertifikaattiketjujen hankinnassa hajautetuista tietokannoista. Arviointia DNS:n soveltuvuudesta sertifikaattien säilytykseen on myös esitetty tässä työssä.

Osa esitetystä teoriasta hioutui ja tuli sovellettua käytäntöön kirjoittajan toteuttaessa sertifikaatteja tukevan DNS-resolverin, ja käyttäessä sitä sertifikaattien hakemiseen DNS:stä. Resolveri toteutettiin käyttäen JaCoB:ia, kryptografisten protokollien tekemiseen tarkoitettua sovelluskehystä. Toteutuksen rajapinta ja korkean tason rakenne on kuvattu tässä diplomityössä.

Avainsanat:	DNS, SPKI, JaCoB, resolveri, sertifikaatti, sovelluskehys
Kieli:	englanti

Contents

Abstract **ii**

Tiivistelmä **iii**

Table of Contents **iv**

Preface **vii**

1 Introduction **1**

1.1 Organization of This Thesis 3

2 Expressing Trust **4**

2.1 Trust Models 4

2.2 Digital Certificates 5

2.3 Storage of Certificates 6

2.4 Public Key Infrastructures 7

3 SPKI **8**

3.1 Certificate Types 10

3.2 Certificate Format 11

3.2.1 Authorization Certificates 13

3.2.2 Name Certificates 14

3.2.3 Tuples 15

3.3 Use of Certificates 15

3.3.1 Certificate Chains 16

3.3.2 Preserving Privacy 17

3.3.3 Verifying Authority 18

4 The Domain Name System **21**
4.1 Domain Name Space 22
4.2 Resource Records 23
4.3 DNS Messages 25
4.4 Name Servers 26
4.5 Resolvers 27
4.6 DNS Security Extensions 28
4.7 Updates in the DNS 29

5 Distribution of SPKI Certificates Using the DNS **31**
5.1 `CERT` Resource Record 31
5.2 Administration of Certificates 33
5.2.1 Choosing a Storage Location 33
5.2.2 Choosing a Domain Name 35
5.2.3 Storage Considerations When Creating a Certificate . . 39
5.2.4 Updating a Zone 42
5.3 Retrieval of Certificates 43
5.3.1 Caching 44
5.3.2 Search Algorithm 45

6 Resolver Implementation **48**
6.1 Functional Requirements 48
6.2 Design Goals 49
6.2.1 Design Patterns 49
6.2.2 The JaCoB Framework 50
6.3 Functional Description 50
6.3.1 Instantiation and Initialization 50
6.3.2 Interfaces 51
6.4 Internal Structure and Implementation Details 53
6.5 Testing 55

7 Evaluation and Consideration **57**
7.1 Evaluation of the Resolver Implementation 57
7.2 The DNS as a Certificate Repository 60

7.3 Future Work 62

8 Conclusions **63**

Bibliography **65**

A Terms and Abbreviations **68**

Preface

This work was done as a part of the Telecommunications Software Security Architecture (TeSSA) project. I started writing this thesis in the fall of 1998, once I felt I had learned enough about the project and its goals. I wish to thank all the people in the project, as well as everyone else who somehow contributed to this work.

I am especially indebted to Pekka Nikander, who gave me the opportunity to join the project, and offered to act as my instructor. He was also the person who thought of the concept of storing SPKI certificates in the DNS in the first place. I thank him for giving me encouragement and advice when I needed it.

I also thank Professor Arto Karila, my thesis supervisor, for advice and help in keeping things running smoothly, which made it possible for me to follow a schedule that was somewhat tight at times. I am likewise grateful to Yki Kortesniemi for constructive criticism about the structure and language of this thesis, Jonna Partanen for her suggestions regarding SPKI certificate storage in the DNS, and Sanna Suoranta for help with administrative matters and the layout.

The deep conversations about SPKI that I shared with Yulian Wang are appreciated, as well as the suggestions regarding the language that were given by Kristiina Volmari-Mäkinen.

My gratitude also goes to my parents and Jie Huang for their support, and more. My apologies to Kanako Yashiro, and all of my friends for whom I did not have as much time as I would have liked during this work.

Finally, I thank David Mack and Miki Kuwaki for inspiration.

In Espoo, on April 14 1999,

Tero Hasu

Chapter 1

Introduction

The advances in network technology seen in recent years have made it possible to access vast amounts of information even with small and simple devices, regardless of location. To ensure the integrity and confidentiality of that information, access to it needs to be controlled. Traditionally, access control has required a way to somehow identify clients, as well as a trusted data storage in the server end containing information about the clients. Such a solution is not well suited for systems in which information is distributed between numerous small and simple network entities.

The information about entities' access rights can also be distributed if its originator can later verify the integrity and origin of the information. This is possible if the data is stored in certificates that are signed by the verifier. Now, in order for an entity to gain access to a restricted resource, the correct certificates need to be presented to the verifier, who has issued at least some of the certificates and controls the access to the resource.

This brings up the question of how to best store the certificates. If every entity were to store all the certificates it may ever need, it would add a lot of management duties for all entities involved in the certificate system. All entities might not even have enough storage space to permanently store all the necessary certificates. Again, a lot of administration would be required if all entities not having the storage space needed to make deals with other entities to arrange for certificate storage elsewhere. Clearly, the system could greatly benefit from a shared certificate repository accessible to everyone.

The work done in this thesis is a part of the ongoing project to create an architecture called Telecommunications Software Security Architecture (TeSSA). The project aims for presenting a complete solution for securing communications in a non-trusted internet-like network environment, and for allowing distributed management of authorization and access control information in that solution. In order for the latter goal to be met, the architecture requires a distributed certificate repository, as well as a suitable component through which the repository can be accessed.

One of the rules of TeSSA is that when looking for solutions, open standards are considered first. Luckily, there already exists a commonly used standard that was created for a highly similar purpose, although probably with different kind of data in mind. The Domain Name System defines a way to arrange storage and on-demand retrieval of information from a public, distributed repository of data. Considering the sheer size of the Internet, it would be infeasible for every host connected to the net to have its own name-to-IP-address translation table, for every host it ever needs to communicate with. The DNS is a working and existing solution for the name translation problem. It is therefore worth looking into its suitability for the certificate acquisition problem as well.

The DNS was designed to be a generic database, capable of storing many kinds of data. Therefore it can also be used for storing certificates. One of the definite advantages of a certificate repository implemented within the DNS would be that it could be adopted quickly on a world-wide scale, as the DNS is already in use in the global Internet. While new data types are readily supported by the DNS, the existing name servers and resolvers would need to be updated to support records suitable for storing certificates. If the necessary modifications were made, entities that would not want to take care of certificate storage by themselves could arrange for a name server to take care of the task. A DNS resolver would then be used as the interface for requesting certificates on demand, in the same manner as it is used to make name-to-address translations.

In this thesis, I cover the details of storing and retrieving certificates from a DNS database, and present my evaluation of the suitability of such a database for distributed certificate storage. I limit my examination to a particular type of certificate, the SPKI certificate, but the results may still be more or less directly applicable to other certificate types.

To test the theory presented I applied it to practice by implementing a DNS resolver which supports certificate entries. In doing so I also provided the TeSSA architecture with a component it was lacking by designing the resolver in such a way that it can be added to the existing architecture as is. On top of the resolver I implemented a search algorithm capable of retrieving entire certificate chains from the DNS. The resolver implementation is described in this thesis, and results based on experiences with it are also presented.

1.1 Organization of This Thesis

The rest of this thesis is organized as follows. Chapter 2 presents background information by shortly discussing trust and digital certificates in general. Chapters 3 and 4 continue by covering SPKI and the Domain Name System, respectively. Chapter 5 describes how the DNS may be used as a repository to store and retrieve SPKI certificates. Chapter 6 describes a prototype of a DNS resolver implementation which supports retrieval of certificates. Chapter 7 provides some analysis and evaluation of both the prototype and the suitability of the DNS for SPKI certificate storage. Finally, in Chapter 8, conclusions are presented.

Chapter 2

Expressing Trust

2.1 Trust Models

As Lehti and Nikander state in [18], trust is a belief that an entity behaves in a certain way. The believer knows in what way it trusts the entity. Trust is rarely absolute. Therefore, if Bob says "I trust Alice", it isn't really all that informative. Such a statement really only tells the listener that Bob trusts Alice in some matters, which is even likely, but what any of those matters are, that remains to be established. Statements such as "I trust Alice to do her part of the job well" or "I trust Alice to be kind to me when she is in a good mood" are much more informative, even though there may still be other ways in which Bob trusts Alice, which are not revealed by the statements.

It is important for trust to have attributes that make it clear what kind of trust is in question. Sometimes such attributes may be obvious from the context, however. Suppose your friend tells you that someone used a security hole in his mail software to send sensitive information from his hard disk to a Usenet newsgroup. If you then say that you trust your mail software, it can be guessed from the context what kind of trust you mean.

The source of the trust, the target of the trust, and the kind of trust together form a trust relationship. If there is more than one target, the trust relationship can be split into as many relationships as there are trustees. In some cases both parties can be the same. For example, the statement "I trust myself to get this thesis written before the deadline" describes one such an autonomous trust relationship.

A trust model of a system is defined by the set of all trust relationships in the system. "Trust no one", an empty set of relationships, would be an ideal trust model. However, in practice it is difficult and often impossible to accomplish. In many cases there is no way to get the task done without trusting any external entities. In Java 1.0, all code loaded from the open network is untrusted. Execution of such code is done inside a sandbox, a very restricted environment in which direct access to critical resources should be impossible. No matter how essential a task is, the applet simply cannot perform it if the limited resources are not enough. This was found to be too restrictive, and in Java 2 it is possible to choose which access rights are granted to which programs.

Generally, when speaking about software systems, the number of trust relationships in which the user trusts another entity should be minimized to achieve better security for the user. This is an oversimplification, however, because it does not account for different types of trust, or the degrees of importance that the other party indeed is trustworthy. Categorizing and analyzing trust, called trust modeling, can be done when a more accurate idea of the security of a system is desired.

2.2 Digital Certificates

Certificates were originally viewed as being signed documents which bind names to keys. However, since then, it has been thought best to expand the definition, as it does not allow for the need to state that some entity is authorized to get a certain service, for example. Therefore, in this thesis, the definition of a certificate is expanded to mean a signed statement, in which the signer's belief about the properties of some entity is expressed. The property can be a name and the entity can be a key, but that does not need to be the case. The belief may not be justified, but that doesn't make the certificate invalid. Sometimes the signer may not even believe the statement, but despite that he can express belief in it. Doing so may be useful if the signer wants to delegate rights he expects to obtain in the future.

It is assumed that the cryptographic methods used are such that it is, in practice, impossible to modify a digital certificate, without invalidating the binding of the signature to the statement, and thus the whole certificate as well. As this is the case, the signature is tightly bound to the statement, but unfortunately not necessarily to the signer. However, if the signer is thought to be the key used for signing, or the corresponding public key, the signer also becomes tightly bound to the document.

The digital signature would typically consist of a number first computed from the statement and then encrypted using the private key of the signer. Some information about the signer is usually also included, as the effort to identify the signer could otherwise require that the number be decrypted using different public keys until a match is found. A good choice for the information is the key that forms a key pair together with the key used to sign, as it then needn't first be acquired before the signature can be checked.

Certificates are often categorized into different types according to the kind of statement in the certificate. If it says that an entity is allowed to do something, then the certificate is called an authorization certificate. If it specifies the name of an entity in some name space, then the certificate is often called a name certificate, or an identity certificate. Here, the term name certificate is preferred, for the reason that a name does not necessarily imply identity. Many different entities could be given the same name, even in the same name space, unless measures are takes to avoid collisions.

It should be noted that certificates are closely related to trust relationships. In fact, they are representations of trust relationships. To find a good trust model for a system, one may want to consider using sandboxes to minimize the amount of trust needed, and the harm that malicious code can cause, and digital certificates to express the trust that is required. This scheme immediately implies one trust relationship, namely the need for the user to trust the sandbox to work according to specification. If the user trusts the sandbox, she can issue a certificate that expresses the trust, and proves that the sandbox is authorized to use critical resources in the system it is running on.

When considering which entities to assign certificates to, it may be useful if the security policies of those entities are known. A security policy is the set of rules and practices that regulate how sensitive information and other resources are managed. Definitions of such policies, especially formal ones, may aid in the decision of whether or not to trust an entity. Languages for defining policy rules exist. One example of these is the Security Policy Specification Language (SPSL) [6], which is meant for specifying communication security policies.

2.3 Storage of Certificates

One certificate may be needed by multiple entities. If each entity stored their own copy of the certificate, it would perhaps speed up the process of proving the existence of a trust relationship. However, this kind of duplication of data would make it hard to avoid inconsistencies between the data possessed by different entities. Also, making an update to a piece of information would require that each concerned entity be notified. This would probably lead to the generation of unnecessary traffic, as every one of the entities might not need the information between updates. If all entities instead used a shared repository, there would be very little or no unnecessary traffic. Entities could acquire updated information only if and when they need it, an arrangement sometimes called “lazy evaluation”.

A repository shared by a large number of entities is likely to be huge. Centralized solutions would probably prove inadequate, at least because of congestion caused by all entities needing a certificate accessing the same network node. Distributed databases do not have this problem; they scale well, because more servers can be added if the amount of data stored in the database gets too large for the existing servers to handle. Distributedness also increases fault-tolerance, as failure in one network node does not make the whole database inaccessible. This reduces the chance of denial of service situations.

2.4 Public Key Infrastructures

A public key infrastructure (PKI) is a system in which public keys and information pertaining to the keys are represented in a certain, defined way. Digital certificates can be used to bind the information to the keys, regardless of what kind of information is in question. A PKI also defines the process of checking whether a given certificate is valid. That is necessary for certificate revocation and expiration to be possible.

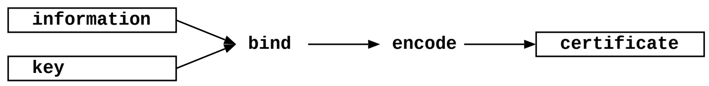

Figure 2.1: Certification.

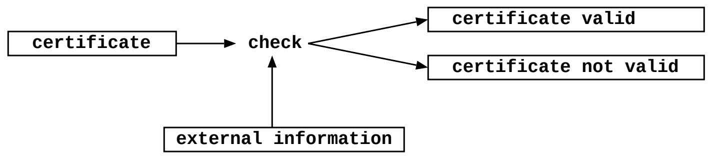

Figure 2.2: Validation.

The kinds of procedures used to bind information to a key and to encode the result, and to verify the validity of a claim given in the encoded form are the defining characteristics of any a PKI. In [5], these two operations are called certification and validation.

A PKI and a suitable certificate repository can together provide a foundation for managing trust and policy information in the form of digital certificates, even for a large distributed system with numerous interacting entities. A PKI defining a common representation for information offers support for making interoperable applications. The same is true for a shared repository that is accessible to all entities in the distributed system.

Chapter 3

SPKI

Public key technology is widely employed for security purposes. To ensure the interoperability of applications which require the use of public key certificates, different parties have to agree on common certificate formats and semantics. Detailed and complete specifications of such things would promote the development of interoperable applications.

The Internet Engineering Task Force (IETF) SPKI Working Group is attempting to develop an Internet standard which defines an infrastructure for managing public keys. The standard is to be called Simple Public Key Infrastructure (SPKI), and is to define a public key certificate format, associated signature and other formats, and key acquisition protocols. The overview of SPKI presented in this chapter is based on the latest draft documents released by the working group [13, 14].

SPKI, being a PKI, defines certification and validation mechanisms. The structures and encodings used when creating a certificate are covered in Section 3.2, and validation issues are touched in Section 3.3.

As the name implies, unnecessary complexity was avoided in the design of SPKI. That makes it faster to get a grasp of the infrastructure, and to make implementations based on it. Simplicity of the implementations should allow them to be used even in constrained environments, such as smart cards.

Lack of complexity does not imply that SPKI could only be used for a very limited number of different purposes. SPKI certificates are expressive, and can be used to describe just about any a trust relationship. Also, any entity that has a private key can generate certificates; the right to do so is not reserved just to some high-and-mighty certification authorities (CAs). Indeed, support for various different trust models is important for an infrastructure that is intended to become an Internet-wide standard.

One of the fundamental ideas behind SPKI is that there is no need to have a unique name other than a key used in public key cryptography. A public key can act as an identifier for the set of all entities which can prove possession of the counterpart of the public key. If there is only one such entity, then it is uniquely identified by the public key. If the key generation process is flawless and there are enough many different possible keys, it should be safe to assume that no two keys generated will ever be the same. The creator of a key pair just needs to keep the private key as a secret and no one else can impersonate as her, assuming that it is infeasible for anyone to acquire the key through cryptanalysis. An entity possessing multiple private keys effectively also has multiple identities, and the freedom to choose as whom to appear when interacting with other entities.

Although public keys are ultimately used as the identifiers in SPKI when verifying the existence of a trust relationship, sometimes indirection may be desirable. One such reason could be the desire to have shorter identifiers. A hash of a public key can be used as an identifier instead of the key itself. The hash function should be such that collisions do not occur with any significant probability and that it is computationally infeasible to find any two distinct inputs which hash to the same output. That should guarantee that a hash of a public key is just as unique an identifier as the public key itself.

Names, in turn, can be used to achieve delayed binding or increased convenience. One can express belief that the entities with a certain name have a certain property, even if no entity yet has the name. Also, people tend to find it easier to remember names rather than long, seemingly random numbers. It should be noted that in SPKI, there are no universal names which would refer to the same keys in every name space; hence it is necessary to know to which key a name is relative to. Neither does SPKI attempt to enforce that there be no more than one key per name. Names can therefore be thought to refer to groups of entities.

One of the main requirements in the design of SPKI was to create an infrastructure which would offer support for making authorization decisions. SPKI certificates allow not only names, but also authorizations to be bound to keys or other objects. Thus policy rules can be expressed and permissions can be granted in the form of digital certificates. Authorization decisions can be supported by or be completely based on a set of SPKI certificates. A mechanism that can be used in deriving such decisions is described in Section 3.3.

The SPKI specification doesn't provide a list of authorizations that can be included in an SPKI certificate. The decision of which authorizations to support in an application is left to the developer, which adds to the simplicity and generality of SPKI. Usually it is the entity that originated a certificate that must make decisions based on it; it should be safe to assume that the creator of a certificate can also interpret the authorizations encoded into it.

3.1 Certificate Types

An SPKI certificate is a tool that can be used to bind together pieces of information. The SPKI specification categorizes such pieces of information to three classes: key, name and authorization. Anything that can be used as a unique identifier fits into the key class. Authorization definitions belong to the authorization class. Everything else must be categorized as a member of the name class.

One SPKI certificate can be used to tie together two pieces of information which belong to different classes. Therefore the are $\binom{3}{2} = 3$ general kinds of certificates. I use the following terms to refer to SPKI certificates when I want to also express the category they belong to:

authorization certificate Binds together a key and an authorization.

name certificate Binds together a key and a name.

attribute certificate Binds together a name and an authorization.

By considering the ordering of the items in a certificate one gets the following $2!\binom{3}{2} = 6$ different mappings:

1. key $\Rightarrow$ authorization
2. key $\Rightarrow$ name
3. name $\Rightarrow$ authorization
4. name $\not\Rightarrow$ key
5. authorization $\not\Rightarrow$ key
6. authorization $\not\Rightarrow$ name

The mappings 1. and 2. have been marked as valid because a key can be assigned any names or authorizations; therefore possession of a key and an authorization certificate which has the key as the subject implies authority, i.e. $\frac{key \rightarrow authorization \quad key}{authorization}$ (M.P.)[1]. Similarly, possession of a key and a certificate which binds the key to some name implies possession of the name, i.e. $\frac{key \rightarrow name \quad key}{name}$. The mappings 4. and 5. have been marked as invalid; even if an entity proves possession of a name and presents a certificate which ties

[1]Modus Ponens. If (one possesses a certificate stating) *key* implies *authorization* and (one possesses) *key*, then (one possesses proof of) *authorization*.

the name to some key, there is still no way to be sure that the identifier in the certificate belongs to the entity, as more than one entity can have the same name. The same applies for authorizations. In SPKI, having a certain name implies belonging the a group of entities having the name. Any defined authorization can be assigned to any group; thus the mapping 3. is valid. However, the same authority can be possessed by some other entity that does not belong to the group, and the mapping 6. must be considered invalid.

Based on the above results it can be noticed that each category of SPKI certificates has only one possible direction for the mapping, and a more accurate description for the categories can be given:

authorization certificate Binds an authorization to a key.

name certificate Binds a name to a key.

attribute certificate Binds an authorization to a name.

Given the possible mappings, there are two different ways an entity could prove authorization using a mixture of SPKI certificate types. These two ways are given below; they show that each category of certificates has its uses in making access control decisions.

1. $\frac{key \rightarrow authorization \quad key}{authorization}$. (M.P.)
2. $\frac{name \rightarrow authorization \quad \frac{key \rightarrow name \quad key}{name}}{authorization}$. (M.P.)

3.2 Certificate Format

In SPKI, a clear distinction is made between name certificates and the other two kinds of SPKI certificates. The structure of name certificates is different from that of authorization and attribute certificates, and is shortly described in Section 3.2.2. Authorization certificate structure is presented in Section 3.2.1 to the extent that is relevant to this thesis; attribute certificates share the same structure. A more complete specification of both certificate structures can be found in [14].

SPKI certificates are represented using S-expressions. An S-expression is a data structure which can be used to represent complex data. Its specification was first developed for Simple Distributed Security Infrastructure (SDSI). The development was influenced by the LISP programming language, in which a particular form of S-expressions is used.

In short, an S-expression is either a string or a finite list of elements. The strings in the expressions consist of a concatenation of zero or more octet strings. The lists have zero or more elements, which can be either strings or lists.

While the S-expressions are always structurally the same, their encodings may differ depending on their use. A compact format is efficient when sending data over a network, whereas user interaction requires a readable representation; the two attributes rarely coincide. Typically, no single encoding can be found which is most efficient for every imaginable situation. For this reason, the S-expression specification defines several different formats for both strings and lists. For details on them, refer to [30].

In order for a checksum or message digest calculation to always give the same result for the same expression, regardless of differences in the encoding, the expressions need to be translated into some common format. The S-expression specification defines such a format, and calls it the "canonical" format. It is uniquely defined for each S-expression, and intended to be easy to parse [30].

In the canonical format strings are written as-is, except that they are prefixed with a string representation of a decimal number, followed by a colon. The number states the number of octets in the string, excluding the prefix. Elements in a list are written back to back, with no separators. All lists are surrounded with parentheses to make it possible to tell which element belongs to which list.

```
(3:ssh(4:host10:tcm.hut.fi)(4:user4:root)(9:max-times1:ö))
```

The above expression is an example of the canonical format. For purposes of readability, I prefer to write it as:

```
(ssh (host tcm.hut.fi) (user root) (max-times #F6#))
```

Another reason for using the latter format is that the canonical encoding of keys or hashes is extremely likely to contain some unprintable characters.

Both SDSI and SPKI use S-expressions to represent certificates. Compared to the description above, SPKI imposes a further requirement to the S-expression structure. Any lists in an expression must always have a string as the first element. The string often identifies the type of data contained in the list. In the above example `ssh` could indicate the kind of permission described by the expression, in this case perhaps being the permission to create an SSH connection to the given host as the given user. The meaning is for the issuer of the statement to decide, however.

S-expressions may contain octets which are not printable in the character set used; sometimes some of the octets may not even belong to the character set. For example, most SPKI certificates encoded using canonical S-expressions could not be expressed using 7-bit ASCII, which can be a problem if certificates are sent by e-mail or managed by editing text files. For this reason the SPKI specification allows the use of the base64 encoding, as defined in [15]. The base64-encoded result is often surrounded with braces. For example, `(ssh (host camphor.tcm.hut.fi) (user root))`, after being converted to the canonical form, encodes into:

{KDM6c3NoKDQ6aG9zdDE4OmNhbXBob3IudGNtLmh1dC5maSkoNDp1c2VyNDpyb2
90KSk=}

It is, however, more efficient if the canonical format can be directly used for both storage and transmission, as that format anyway needs to be used in the hash calculations.

The following two sections give the high-level syntax for SPKI certificates, but do not mention signatures. All SPKI certificates have a signature, which usually follows right after one of the `cert` structures described below. The signature contains a hash of the `cert` element being signed, the signer's public key or its hash, and the hash value encrypted using the signer's private key.

3.2.1 Authorization Certificates

An SPKI authorization certificate consists of several fields, as can be seen from the following format specification.

```
"(" "cert" <version>? <cert-display>? <issuer> <issuer-loc>?
<subject> <subject-loc>? <deleg>? <tag> <valid>? <comment>? ")"
```

Five of the fields are relevant when making authorization decisions. These five are:

<issuer> **Issuer** The entity that issued the certificate, expressed as a public key or its hash.

<subject> **Subject** The entity that the certificate was issued to, commonly expressed as a public key or its hash. May also be a name, or a chain of names in which every name that follows another is relative to the other name. If the keyholder indicating the name space is not explicitly given, it is taken to be the issuer. Other options for a subject include a whole object (e.g. a Java class file), an object hash, or a threshold of subjects. By using a threshold subject the issuer can specify any number of subjects and insist that at least a certain number of them must cooperate in order to exercise the authorization given in the certificate.

<deleg> **Delegation** The optional delegation field, if present, signifies that the subject is permitted to delegate the permissions specified in the certificate to others, fully or partially.

`<tag>` **Authorization** The authority field specifies the permissions granted by the certificate. SPKI does not give a set of all the allowed permissions, so application developers are free to invent their own. However, the grammar given in SPKI must still be followed when deciding on expressions for new permissions.

`<valid>` **Validity** The optional validity field specifies the period of time during which the subject possesses the permissions described in the authority field. Either the lower bound, or the upper bound, or both can be given, up to the accuracy of seconds. The validity field may also contain, in addition to the time period or alone, a set of online tests, which can be used to set further limitations to when the certificate is valid. If the validity field is not present at all, then the certificate is always valid.

Other fields, all of which optional, include:

`<version>` Version number of the certificate format. Certificates containing an unrecognized version number are to be ignored.

`<cert-display>` Gives a display hint for the certificate. Means to aid user interface software in deciding how to display the certificate. Implementors are free to define their own hints, none are specified in SPKI.

`<issuer-loc>` Can be used to specify URIs that provide information regarding the issuer.

`<subject-loc>` Can be used to specify URIs that provide information regarding the subject.

`<comment>` Allows comments to be included in the certificate.

3.2.2 Name Certificates

As SPKI name certificates are used to bind names to keys, they naturally contain a name field and a subject field. An identifier for the name space is also included.

The format of name certificates is the following:

```
"(" "cert" <version>? <cert-display>? "(" "issuer" "(" "name"
<principal> <byte-string> ")" ")" <subject> <valid>? <comment>?
")"
```

Only the three compulsory fields and the validity field are relevant when making access control decisions. They are described below. The rest of the fields are the same as for authorization certificates.

`<principal>` **Issuer** The entity that issued the certificate, expressed as a public key or its hash.

`<byte-string>` **Name** A name for the subject in the issuer's name space. The name may be any octet string.

`<subject>` **Subject** This field is the same as for authorization certificates.

`<valid>` **Validity** This field is the same as for authorization certificates.

3.2.3 Tuples

When analyzing certificates in order to make a decision based on them, it is simpler to only concentrate on that information which is relevant to the particular decision. As mentioned above, only certain fields of SPKI certificates are needed for authorization decisions. The SPKI specification defines two structures called 5-tuple and 4-tuple, which only contain those fields.

SPKI authorization and attribute certificates can be mapped to 5-tuples, for which I will use the notation (i, s, d, a, v). The elements correspond to the Issuer, Subject, Delegation, Authorization, and Validity fields of the certificates, respectively. SPKI name certificates in turn can be mapped to 4-tuples which I denote as (i, s, n, v), where the elements correspond to the Issuer, Subject, Name, and Validity fields. If non-SPKI certificates can be translated to tuples they can also be included in an analysis, and processed in the same manner as SPKI certificates.

After the signature of a certificate has been checked, the certificate no longer needs to be kept in the canonical S-expression format, and can thus be translated into another form. If the memory used can be trusted to protect the integrity of the data, the signature need not be preserved. Tuples do not contain a signature; therefore a conversion from certificates to tuples is a lossy operation which cannot be reversed. The original certificates must not be thrown away if information contained in them is still needed after the processing of the tuples.

As 5-tuples and 4-tuples are usually not kept in mass storage or given to entities other than the one that created them, there is no format defined for them. Any representation which allows efficient processing in memory should be suitable.

3.3 Use of Certificates

As mentioned before, the main reason for the existence of SPKI is to offer support for making authorization decisions. Other uses, such as creating a database containing information about attributes and group memberships of different entities, could be devised. However, for simplicity I'll restrict my examination of the uses of SPKI certificates to the use of authorization certificates in access control situations. As described in Section 3.1, authority can be expressed with a name certificate and an attribute certificate instead of an authorization certificate, but I do not cover that option as it is a trivial matter to replace a name with a key if the correct name certificate is available.

By using the different kinds of SPKI certificates together, just about any a statement expressing trust can be made. Typically, it is not the making of a statement that is difficult, but the decision of whether to make it or not. When someone asks for access to a resource, the entity controlling access should consider the information available about the requester and the risks involved. The bigger the potential damage caused by misuse, the more likely should the answer "no" be. In some cases it is simple to make a decision. Suppose there is no possibility of misuse, and a contract that states that those and only those who have paid for a service are to be granted access to it. Then all the information required for a decision is a list of entities who have paid.

One of the major benefits of using SPKI certificates is that one can often do away with using access control lists (ACLs). If service provider ρ decides to grant entity ϵ the right to use a service, it can issue a certificate stating that ϵ has the right. ρ may not even need to store the certificate anywhere, if and when the certificate has been given to ϵ. It can be left as the responsibility of clients to store the certificates. ρ need not memorize who it has decided to trust, about which matters, nor for how long; in other words ρ need not maintain an ACL. Thus ρ could possibly be very lightweight and reside on a device with very little storage space, even if it has thousands of clients.

Most clients do not access thousands of different servers, so it is usually not a great burden for them to store the certificates they need to access all the servers whose services they do use. Those clients that need to have access to a large number of certificates can be kept light by using an external database. The same database could possibly serve more than one client, but to scale well the database would need to be distributed.

3.3.1 Certificate Chains

In order to keep the tasks of different entities simple, it is necessary for entities to delegate responsibilities to others. ρ might want to assign another server the responsibility of issuing permissions to clients. The other server in turn may need to further delegate those permissions to servers which have more information about would-be clients.

To empower entity δ_1 to act on its behalf, ρ must issue a delegation certificate to δ_1; the certificate should contain at least those rights which δ_1 is to pass on to others. Suppose δ_1 passes some permission to entity ϵ, and ϵ has not received the same permission directly from ρ. Then one certificate is not enough to prove that ϵ has the permission. A certificate chain is required.

A chain of certificates consists of a set of certificates $C = \{c_1, c_2, \ldots, c_n\}$ such that $\forall c_j = (i_j, s_j, d_j, a_j, v_j)$, $2 \leq j \leq n$, $s_{j-1} = i_j$ and $\forall c_k = (i_k, s_k, d_k, a_k, v_k)$, $1 \leq k \leq n-1$, $d_k = true$. (Note that here I am using 5-tuples to represent entire certificates, and that each certificate must be signed by the entity marked into the issuer field.) C can be used to prove that s_n has been given authorization $a_1 \cap a_2 \cap \ldots \cap a_n$ by i_1 for the validity period $v_1 \cap v_2 \cap \ldots \cap v_n$. In addition to C, there could be other, different chains that are proof of the same or overlapping rights for the same or overlapping validity period.

3.3.2 Preserving Privacy

Given the expressiveness of SPKI certificates, a certificate could contain, alone or in combination with others, some information which is considered sensitive by the issuer or the subject of the chain, or by one of the delegators. In such a case measures need to be taken to keep the information from falling into the hands of those who do not need to know it when creating the chain or when proving the right of the subject.

Storage Considerations

If all of the parties involved in the chain can be trusted to keep the sensitive information confidential, it should be enough to use a storage method other than a public, global repository for the certificates in the chain, and to encrypt all communication between the parties when certificates are being transmitted. Such trust is a lot to expect, however.

Certificate Reduction Certificates

Suppose a delegator, δ_l, wants to remain anonymous. It possesses a certificate $c_l = (\delta_{l-1}, \delta_l, true, a, v)$, and wants to delegate the right $\hat{a} \subseteq a$ to δ_{l+1}. δ_{l-1} already knows about δ_l, but δ_l would like to remain anonymous to δ_{l+1}. What it can do is to issue the certificate $c_{l+1} = (\delta_l, \delta_{l+1}, d, \hat{a}, v)$ as it would do in any case, but instead of sending it to δ_{l+1} send both c_l and c_{l+1} to δ_{l-1} and ask δ_{l-1} to issue a Certificate Reduction Certificate (CRC). δ_{l-1} should then create the certificate $c_r = (\delta_{l-1}, \delta_{l+1}, d, \hat{a}, v)$ which δ_l can have delivered to δ_{l+1} without revealing its identity.

Encryption

Use of encryption is a common way to maintain confidentiality. Certificates can be wholly or partially encrypted (along with some random data) using some key which is only known to parties that need to know the information that is encrypted.

Suppose an entity ρ creates the certificate $c = (\rho, \gamma, d, a, v)$ and encrypts it with the key k. ρ then sends the result $e_k(c)$ to γ, telling γ in what kind of access control situations it should present $e_k(c)$ to ρ. It is not essential for γ to know exactly what is written in c to be able to prove its access rights to ρ, who can verify the validity of the proof by using the key k. No other entity can gain any information from $e_k(c)$ just by examining it (assuming they do not possess the key k), which is appropriate if c does not allow delegation and thus only concerns ρ and γ. If delegation is allowed and γ wants to delegate its rights to some other entity ϵ, γ can use $e_k(c)$ to prove its delegation right to ρ, and ask ρ to issue the appropriate certificate c_ϵ. γ can then give c_ϵ – which could be thought to be a CRC – to ϵ.

Delegation of rights stated in partially encrypted certificates can be more straightforward. If only some of the fields of a certificate, for example the authority and validity fields, are encrypted, it is still possible for the subject of the certificate to create a certificate for delegating the rights. The fields can be copied to a new certificate even if the entity copying them does not know their meaning. Only the originator of a certificate chain needs to be able to understand what rights are proven by the chain; it may even have decided by itself how to describe those rights using S-expressions. Such descriptions could possibly be incomprehensible to others even without being encrypted. It is also enough for the verifier of a chain to be able to check what the intersection of the chain's certificates' validity dates is.

3.3.3 Verifying Authority

In a typical access control situation there is an entity ρ that provides a service and will let any entity to which it has, directly or indirectly, issued authorization $\hat{a}$ use the service. To prove the authorization the relevant certificates need to be acquired. The source could be ρ itself, the prover, or some third party. I will not try to cover all possibilities. Instead I assume that ρ does not keep track of the certificates it has issued and that the required certificates are always sent along with the requests, in such an order which allows easy verification.

In some cases, for example when some of ρ's clients are running on smart cards with no storage space for certificates, the above requirement may be too strict. Even if the clients can acquire the certificates from a database maintained by some other entity, the clients would still need to contain enough logic to acquire the correct certificates from the database, which might be counterproductive. In such cases it would be better for the service provider to acquire the required certificates, which can be seen to be a part of the service.

I'll cover the access control situation through an example. If an entity ϵ wants to use the service provided by ρ, it will sign a request with its private key and send it to ρ. Upon receipt ρ checks the signature in the request. It then proceeds to check the certificates sent as proof, perhaps first individually, but then by checking if they form a valid chain.

Validity of a certificate chain is checked using a method called tuple reduction. Below is an example of a reduction of a certificate chain containing three certificates, but the process is essentially the same regardless of the length. If the input is a valid chain of certificates (as defined in 3.3.1), the process of reduction will produce one certificate; otherwise the process will fail.

$$\begin{aligned}(\rho, \delta_1, true, a_1, v_1) + (\delta_1, \delta_2, true, a_2, v_2) + (\delta_2, \epsilon, d, a_3, v_3) \\ = (\rho, \epsilon, d, a_1 \cap a_2 \cap a_3, v_1 \cap v_2 \cap v_3)\end{aligned} \tag{3.1}$$

If the validity period encompasses the moment when the chain is checked and authorization $a_1 \cap a_2 \cap a_3$ is enough to access the service, i.e. $a_1 \cap a_2 \cap a_3 \supseteq \hat{a}$, the chain has been verified to prove that ϵ has access rights to the service.

Servers may have local policy rules, however. Especially in the case of delegated permissions, the server may not accept all valid certificate chains. For example, the server could have a list of entities which it does not trust as an issuer of certificates, and if a certificate contains such an issuer it will not be accepted, and the chain is broken. The verifier will usually check the individual certificates before performing tuple reduction, as there is little point in trying to verify an incomplete chain. Online checks can in some cases be so slow, however, that it is best to only perform them once the chain has otherwise been found to be valid.

Once the chain has been found to prove the access rights of ϵ, what remains is to verify that the entity with whom ρ is negotiating with really is ϵ. Although the request was signed by ϵ, there still is the possibility of a replay attack. Authentication is usually done through some kind of a challenge-response protocol, in which the challenger sends random data to the other party. The recipient uses its private key to encrypt the data, and then sends the data back to the challenger who can decrypt it using the client's public key and verify that the data is the same that was originally sent to the client. The process of authentication effectively closes the certificate chain into a loop which proves that the entity requesting access to a resource can be allowed to access it. The loop for the example chain of certificates is depicted in Figure 3.1.

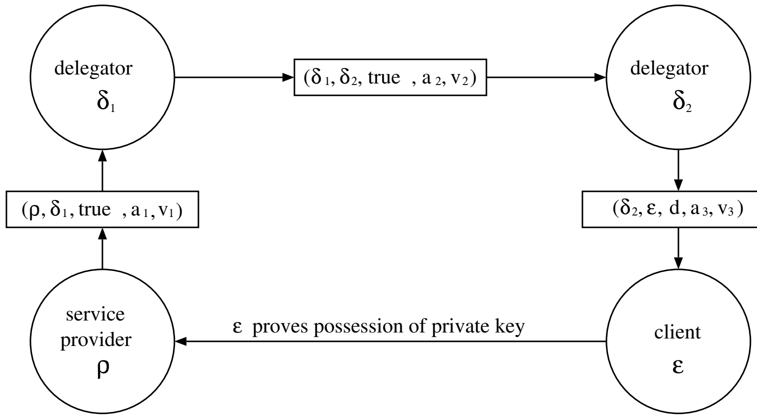

Figure 3.1: Client authorization certificate loop.

Often the client will also want to authenticate the service provider and verify that it is trusted to provide the service the client wants. This is especially important if the service provider needs some of the client's resources to perform the service. Again, a loop of certificates is needed as proof. It is shown in Figure 3.2. The delegating entities γ_1 and γ_2 shown in the figure are some entities that ϵ trusts as introducers. They could even be owned by ϵ and have been assigned the task of collecting data, based on which they could make decisions regarding the trustworthiness of various service providers.

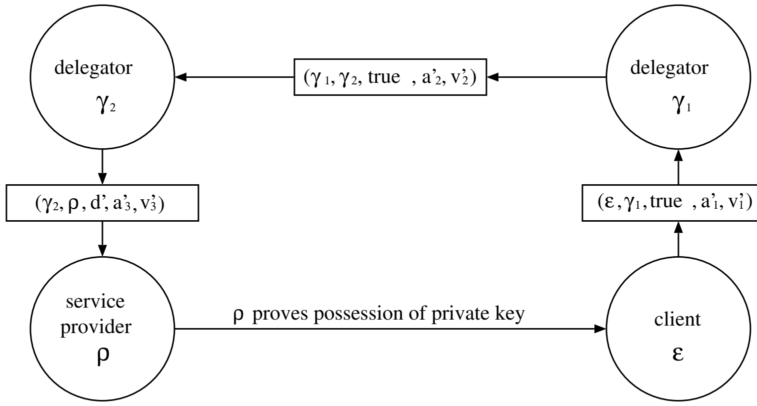

Figure 3.2: Service provider authorization certificate loop.

Chapter 4

The Domain Name System

In early networks, low-level hardware addresses were used to identify machines. Internetworking introduced the concept of mapping universal addresses into low-level addresses. Still, for humans the universal addresses – being just seemingly arbitrary numbers – were impractical and difficult to remember. To alleviate these problems, a new naming concept based on host names was adopted. Host names are usually meaningful and pronounceable text strings. Their purpose is to act as high level, unique identifiers for hosts, in addition to Internet Protocol (IP) addresses. Still, the host names, which could be thought of as mnemonics for the IP addresses, could not be used within the IP packets as the actual addressing information. Therefore, a way to translate the host names to the IP addresses had to be arranged.

The mappings from host names to IP addresses used to be maintained in a single file on a single host. Other hosts could keep their mapping information up to date by fetching the file with FTP. Network administrators would typically e-mail the maintainer of the file about any changes in their local networks, to get the changes made visible to the rest of the Internet. Along with the growth of the net the bandwidth consumed by such administrative tasks grew, and it became apparent that a new solution was required.

The Domain Name System (DNS) is that solution; it is now the member of the TCP/IP protocol suite that is responsible for name resolution. An overview of the DNS is given here, and a more detailed description can be found in [19, 21, 22].

While the major goal of the DNS was to handle the host name to IP address mappings, general usefulness was also sought for in the design. Since an Internet-wide system takes a lot of work to set up, it is reasonable to expect significant benefits in return.

As it stands, the DNS is usable for more than one purpose. In fact, it is a distributed database that can be used to manage any kind of data, as long as the data is not too sensitive to be stored in a database accessible by anyone.

The records in the database are indexed by the domain names associated with them, which are typically host names, but can also be other kind of entities' names.

4.1 Domain Name Space

The set of all different values by which data is indexed into a DNS database forms a domain name space. While the values could be chosen in almost any way, to achieve a distributed database they need to be chosen so that they indicate a position in a hierarchical structure. A domain name space is therefore always structured like a tree. As the tree acts as an index for the database, it is essential that it be kept in a consistent state. Otherwise some of the branches could be "cut off" from the tree, and queries regarding any nodes in those branches would fail. In this thesis, the term *node* is used to refer to both leaf nodes and internal nodes.

In the DNS, a domain can be thought to be a subtree of a tree that represents a domain name space. Figure 4.1 shows a a small name space. The domain `hut.fi` has been circled.

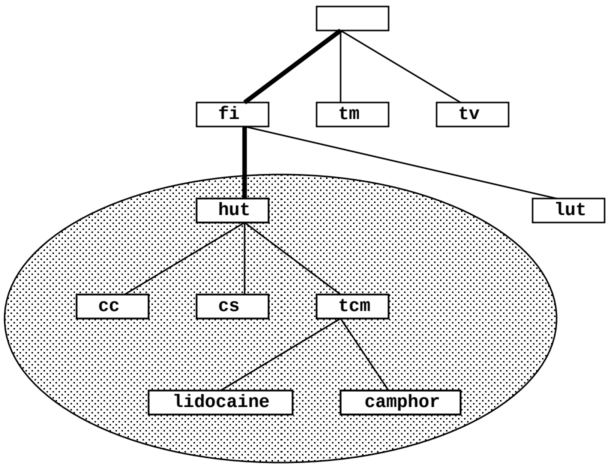

Figure 4.1: The domain `hut.fi`.

The name of a domain consists of a concatenation of the labels of each node on the path from the root of the subtree to the root of the whole tree. In Figure 4.1, the domain name `hut.fi` is acquired by following the emphasized path towards the root. The maximum length of a label is 63 bytes. The empty string is reserved for the root of the tree, and nodes that have the same parent may not have the same label. The DNS standard itself allows labels to contain any octets. Unfortunately, a lot of the flexibility that this freedom would otherwise offer is done away with the fact that all comparisons in the

DNS are case insensitive. Because of this, case cannot always be preserved, even though the standard recommends that it be done whenever possible. The labels used thus cannot be just any binary objects less than 64 octets in size, not even if encoded with the commonly used base64 encoding. Some systems that use the DNS have their own rules for the kind of domain names that are allowed. For instance, the rules defined in [7] should be satisfied when naming a mail domain.

The DNS specification also defines how domain names should be represented as text. Dots are used as separators for the labels in the text representation. Some characters, such as dots and non-printable characters of the labels need to be escaped; i.e. they need to be represented using more than one character.

A zone is a group of domain names and data that are governed by the same authority. Suppose a name server (see Section 4.4) is authoritative for the entire domain `hut.fi`, except for the subdomain `tcm.hut.fi`, which is managed by another name server. The whole domain `hut.fi` then is not ruled by one and the same authority, and is thus divided into at least two zones. The authority managing `tcm.hut.fi` could have further delegated the authority for possible subordinate domains of `tcm.hut.fi`. To name a zone, the name of the subtree within the zone that is the closest one to the root (of the whole tree) is used.

4.2 Resource Records

A domain name space and the information associated with the names together form a DNS database. The information is composed of so-called resource records (RRs). As different kinds of information usually require a different storage format, each kind has its own resource record type. Various different types have been defined in [22], and more can be defined as the need arises.

The RRs have the same high-level structure, regardless of the type. The structure is shown in Figure 4.2. Among other things the record contains the name of the node to which the record pertains, and the time-to-live (TTL) value of the record. The latter value indicates the number of seconds for which the RR may be cached.

The last of the fields, RDATA, has a type-dependent internal structure. The type of an RR can be determined by looking at the type value contained within it. The RDATA field also consists of one or more fields. As fields with the same semantics may have different sets of possible values in different environments, it may be desirable for the format of RDATA fields of the same type to vary depending on the kind of protocol family or instance of a protocol that the RDATA information concerns. For example, an IP address

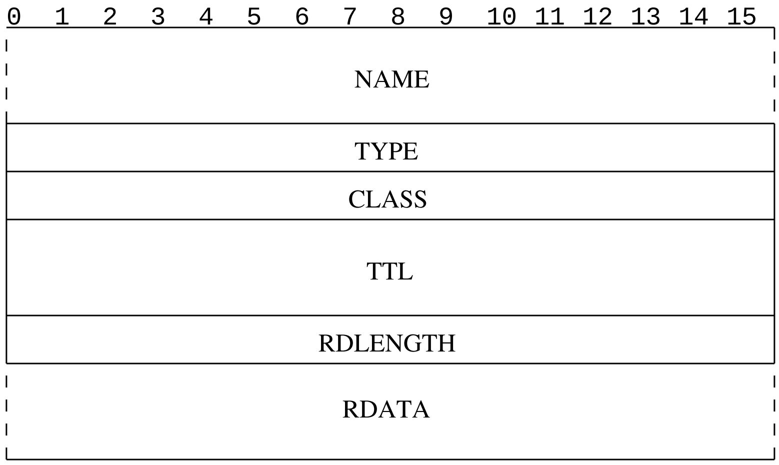

Figure 4.2: Resource record structure.

is 32 bits in size; longer addresses could be used by some other protocol. For the reason given above, RRs also have a class value, which specifies the protocol suite that the information in RDATA applies to. The format of RDATA also depends on this value. Each type and class are assigned their own unique numbers. The class value 1, which has the mnemonic `IN` for Internet, is currently by far the most commonly used one. Thus, for brevity, I often neglect to mention the RR class altogether when discussing record types and other selection criteria.

A query operation is an attempt to retrieve a set of all the RRs in a certain database which have the same domain name, type, and class. Such a set is sometimes referred to as an *RRset*. Currently, requests for RRsets of the type `A` are most common, as an `A` RR is needed when one wishes to know the IP address corresponding to a domain name. The RDATA fields of `A` RRs have the simple structure shown in Figure 4.3.

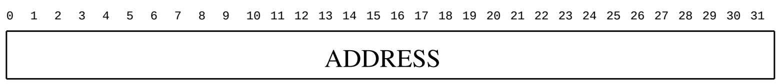

Figure 4.3: `A` RDATA structure.

Another important type of RR is `CNAME`. Records of that type indicate that the domain name to which the record pertains is actually only an alias, and that the primary name of the alias owner is contained within the RDATA field of the RR. The length of the field varies depending on the length of the name. `NS` RRs have the same structure, but the domain name specifies a host which is authoritative for the zone starting at the domain of the RR.

Within DNS messages, RRs are represented in the binary form described in this chapter. When stored in a name server's configuration file, the RRs are typically represented in a printable and readable form. The format used when caching RRs completely depends on the implementation.

4.3 DNS Messages

Resource records are carried from a name server to another within DNS messages. These messages can be transmitted either as User Datagram Protocol (UDP) [28] datagrams or in a byte stream formed with Transmission Control Protocol (TCP) [29]. UDP packets are preferable, as they offer lower overhead and better performance. However, the use of UDP brings limitations to the size of messages that can be sent. The DNS specification defines the maximum size of messages that have a UDP datagram as the transport to be 512 bytes (excluding the UDP header). Queries are typically short and will fit in 512 bytes, but replies containing many entries may well exceed the limit. Even a single RR containing an SPKI certificate, for instance, could easily be too large. In such a case the response should be truncated and the truncation flag in the message header (see below) should be set in order to conform to the DNS standard. When a resolver finds that the flag has been set it should resubmit the query using TCP. An intelligent resolver may first check to see if the response contains the entire answer section; if so, there is no need to use TCP.

The UDP standard itself does not define such a severe 512 byte limitation. Most of the current Internet can handle packets of up to 1500 bytes in size without fragmentation [8, 23], and it can be expected that the average MTU (Maximum Transfer Unit) of the network hardware in use will continue to grow. Even now, some networks are only limited by the maximum size that can be specified in an IP datagram header, which is 65535 bytes. In practice, packets with significantly larger DNS payload than 512 bytes could be transmitted between many network nodes if the DNS software accepted larger UDP packets. A backward compatible modification to the DNS protocol which would allow larger responses has been suggested [8].

The format of DNS messages is the same for both queries and replies. All messages have a header of the same size and structure. The structure is shown in Figure 4.4.

Only brief descriptions of the fields in the header are given here. For more information, refer to [22].

ID A 16-bit identifier used by the client to match replies to outstanding queries, or by the server to detect duplicated requests. The value can be freely chosen by the program that generates a query.

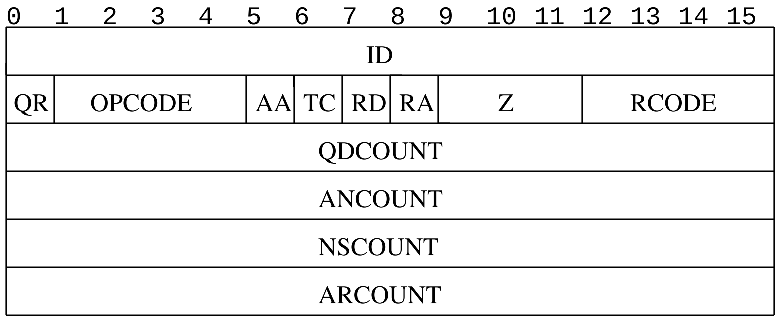

Figure 4.4: The structure of a DNS message header.

QR A flag which is set if the message is a response.

OPCODE A value that states the purpose of the message. Typically 0 (`QUERY`), which indicates that the message is a request for an RRset.

AA A flag which is set if the name server that sent the response is authoritative for the domain that the query concerned.

TC A flag which is set if the message was truncated.

RD A flag which is set if the client wants the name server to pursue the query recursively.

RA A flag which is set if the sender of the response supports recursive queries.

Z Reserved. Must be zero.

RCODE A value that specifies the kind of error, if any, that the sender of the response encountered when attempting to find an answer to a query.

The header is immediately followed by QDCOUNT entries in the question section, ANCOUNT RRs in the answer section, NSCOUNT RRs in the authority records section, and ARCOUNT RRs in the additional records section. The question section entries are not resource records, but entries that contain a domain name, a type value, and a class value which together specify which RRs should be fetched from the DNS when making a query.

4.4 Name Servers

Name servers are repositories for the data that makes up a DNS database. The database is divided into zones which are distributed among servers. The information about the zones governed by a name server is usually stored in a set of files accessible to the server. These files, which I call zone files, are typically text files and they are read to memory when a server is launched. Together the servers hold all the RRs associated with the domain names in the database, including those that contain information about the structure of the domain tree.

The main task of a name server is to answer queries regarding the data stored in a database. Both queries and responses to the servers are sent as standard DNS messages. There are differences in the kind of support that name servers offer. Some servers do not offer a recursive service; they will not attempt to acquire data that does not belong to one of their own zones. Some will do so upon request by sending queries to other servers. Not all RR types are supported by all server software. Typically, at least all of the standard, non-obsolete types are supported, unless the standard is very recent. Many servers also support some experimental types, such as `CERT` (see Section 5.1). Currently, the most widely used name server is called BIND, which has varying support for different types depending on the version used.

4.5 Resolvers

A DNS resolver is an interface to the Domain Name System. To be more specific, it is a library of programs or routines that to some extent hides the complexity of constructing, sending, receiving, and interpreting messages required for communicating with name servers.

In order to acquire information from the DNS, a resolver needs to know the IP address of at least one name server and to be able to communicate with it. If a name server offers a recursive service, the resolver should need to do nothing more than to send one query to that server in order to acquire the desired information, or to find out that the information is not in the database. If recursive service is not available, the resolver should still be able to acquire referrals to other name servers until it can either find a name server that has the desired information, or conclude that there is no server that has it.

Resolvers usually have a cache in which RRs may be stored. The time limit for storing an RR is specified by the TTL value of the record. Authoritative data should be preferred over non-authoritative data when deciding what data to keep in the cache. Having information in the cache in many cases drastically speeds up the process of finding the answer to a question.

4.6 DNS Security Extensions

The DNS, in its current form, does not support the checking of data integrity, nor does it do much in terms of authentication. Therefore, hosts using the DNS are vulnerable to certain kinds of attacks. They cannot be sure if the resource records contained in a DNS message originated from a server authoritative for those RRs. Neither can they be convinced that the records were not modified on the way.

The IETF DNS Security Working Group is attempting to create a standard that would, when widely adopted, significantly improve the security of the DNS. This would-be standard is called DNS Protocol Security Extensions [11], henceforth referred to as DNSSEC.

DNSSEC uses digital signatures to achieve data integrity and origin authentication. Three new resource record types have been defined, one for the signatures, one for keys used for checking signatures, and one to convey information about names that do not exist and types of RRs that exist for a given name.

The resource record type for signatures is called `SIG`. Resource records of this type are used to cryptographically bind RRsets to the signer and a validity interval.

A security-aware resolver needs to have access to the public keys of the zones from which it receives signed data, in order to check that the data was properly authorized and is current. Given that a resource record type called `KEY` has been defined for public keys, servers with even basic DNSSEC compliance can be used for public key distribution, even for keys other than those needed by DNS servers and resolvers.

The third new RR type, `NXT`, is needed to securely indicate that certain names do not exist in a zone. The `NXT` RR also contains a bit map which can be used to determine what RR types are present for an existing name.

In addition to adding security to the Domain Name System, DNSSEC can be viewed as a PKI of a kind. Resource records can contain any kind of information. When a `SIG` record is created for an RRset, the information contained in the set is bound to a domain name and the signing key. The RRset and the `SIG` record together effectively create a certificate, a signed statement about the properties of the entity that has the domain name. The validity of a "certificate" is confirmed by checking the signature and its inception and expiration times, as well as the TTL value of the `SIG` RR. It could be argued that certificates do not need their own RR type, as a certificate can be represented using multiple RRs of the already standardized types. However, only those of entities' attributes that have an RR type could be certified through the use of this scheme. To get the expressiveness of such "certificates" on par with SPKI certificates, new RR types would nevertheless be required.

4.7 Updates in the DNS

The DNS standard requires that information about every zone is kept on two or more name servers. This is to ensure availability despite of host or link failures, but it can also be that a query introduces less load to the network if the querying host contacts the server to which is has the best network access. In any case, the duplication of data adds to the effort required to maintain a consistent database.

The DNS was originally designed to be a statically maintained database. The data in it was expected to change, but not very frequently. Thus powerful mechanisms for making updates weren't all that necessary. Editing zone files by hand is still a common way to make updates. To avoid having to make the same changes to several files, the updated files could be propagated using FTP or some other generic method. The DNS standard defines and recommends a particular way to arrange full zone transfers. More recently, other proposals for making updates have been made as well. These proposed standards describe mechanisms for making incremental zone transfers, for notifying of zone changes, and for making dynamic updates in the DNS. All of these are briefly described below.

One of the name servers that are authoritative for a zone is designated as the primary server for the zone, while the others are secondaries. The name of the primary is included in the zone information. The difference between a primary and a secondary is that when a zone needs to be updated, it is the primary server's zone file that should be edited by the zone administrator. After editing, the running name server is signaled to get it to load the updated zone information. The other servers authoritative for the zone periodically check to see if changes to the zone have been made by checking the serial number stored within the zone information. If the number has been incremented, the secondary server sends a message with the opcode `AXFR` to the primary, which indicates that the requester wants to receive a current copy of the entire zone information. The primary should react by sending a sequence of response messages containing the asked information. For more details about the full zone transfer mechanism, refer to [21].

Full zone transfers waste a lot of bandwidth, as the entire zone information needs to be transmitted even if the updates made to a zone have been small. For this reason, a mechanism for only communicating information about the changes has been proposed. Incremental zone transfers are described in [26]. The procedure of making such a transfer is similar to making full zone transfers. A secondary server issues an `IXFR` request to the primary when it wants to receive the changed portions of a zone.

The security of both full and incremental zone transfers can be improved if DNSSEC support is available. This is done by using `NXT` resource records to assure that every authoritative name and type will be present in full zone transfers. For incremental zone transfers, the completeness of the transferred zone information cannot be confirmed, but the integrity of individual RRs can still be checked from the signature records.

A proposed standard, defined in [32], addresses the fact that, as described earlier, all non-primary servers need to frequently check to see if the primary server has updated a zone. The proposal describes a new opcode called `NOTIFY`. Once a primary's zone information has been updated, it can send a notification message to other servers authoritative for the zone so that they know to take action to get their zone information updated. The notification message is a standard DNS message that has `NOTIFY` as the opcode; only a subset of the message fields is used, however.

It would often be convenient if a zone could be updated simply by sending messages to a name server authoritative for it. To eventually allow doing so with a wide selection of server software, there is, at the time of writing, considerable ongoing effort to design and standardize a DNS extension for making updates dynamically. A proposed standard is described in [33]. The proposal defines the `UPDATE` opcode, which, when supported by a name server, makes it possible to add or delete RRs or RRsets from a zone; new zones cannot be added. The presence of the opcode causes a DNS message to be interpreted differently. The answer section, for example, is taken to contain prerequisites which must all be satisfied for the update operation to take place. An `UPDATE` message also contains the name of the zone to be updated and naturally also a list of the updates to be made. Once constructed, the message should be sent to the primary name server of the zone concerned.

The proposal for dynamic updates does not describe a method for checking a message sender's permission to initiate an update. Another proposed standard, the Secure DNS Update [10], describes a mechanism for doing so. Improvements to it are suggested in [9], and a simpler and more flexible alternative is presented in [34]. Authentication is handled using either public or symmetric keys, and the authority given by possession of a key is dictated by the server's policy. It has been suggested that a set of `KEY` RRs could be used to represent the policy, but as the policy is only used by the server to make access control decisions and need not be exposed to others, the zone administrator should be free to select any kind of a presentation. A policy could, for instance, limit the scope of authority of each key to certain domains within a zone.

Chapter 5

Distribution of SPKI Certificates Using the DNS

In order to build a reliable system on top of an infrastructure that makes heavy use of certificates and depends on their availability, one should give careful consideration to certificate management issues. One of the most important of them is certificate storage, which is not necessarily addressed by the certificate infrastructure itself, as is the case for SPKI. The storage solution must scale well if it is to support an Internet-wide system that is open for everyone to participate in. This chapter describes a way to use the Domain Name System as a globally accessible, distributed database from which SPKI certificates can be retrieved on demand.

5.1 `CERT` Resource Record

Any data to be stored in the DNS must first be placed into a resource record. Some of the most common types were already covered in Section 4.2. The RR type most relevant to this thesis is `CERT`, which has the type number 37. It has not yet been standardized, and anything that is said about RRs of type `CERT` in this thesis is subject to change. The specification for it can be found in [12]. The purpose of the type is to facilitate storing certificates in the DNS. `CERT` records have a header of the same format as all the other RRs; the structure of the type specific RDATA field is shown in Figure 5.1.

`CERT` RDATA contains a type value which specifies the type of certificate contained within the field. Among the currently defined certificate types are SPKI and X.509. The type value 2 and the type mnemonic `SPKI` are reserved for SPKI. In addition to the certificate type number, the `CERT` RDATA contains a key tag, an algorithm identifier, and the certificate itself. Alterna-

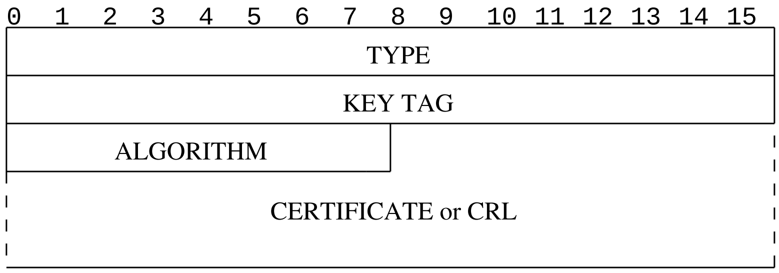

Figure 5.1: `CERT` RDATA structure.

tively, instead of a certificate, the record may contain a certificate revocation list (CRL).

The algorithm value specifies the hash and the public key algorithms used in the creation of the certificate. The values used are the same as for the algorithm fields of the `KEY` and `SIG` RRs (see Section 4.6) except that the value zero indicates that the algorithm has either not been included in the DNSSEC specification or that it is unknown to the entity that created the `CERT` RR.

The current specification states that the key tag is a 16-bit value which should be computed from a key embedded in the certificate. This definition does not seem to account for the fact that not all certificates have exactly one public key embedded in them. Some have none, and some have more than one. Assuming that a suitable key can be chosen, the description of the algorithm to be used for performing the calculation can be found from [11]. Before calculating the value, the key needs to be translated into the the same format as it would have within a `KEY` RR. If the key type is not defined in DNSSEC, the translation naturally cannot be done. In that case the tag should be set to zero. If the tag is not calculated, for whatever reason, the algorithm value must also be set to zero even if the algorithm is known and defined in DNSSEC. This is because it is possible for a tag calculation to yield the value zero, and thus a zero tag by itself cannot be taken to indicate that a calculation has not been performed at all.

The `CERT` RDATA specification does not say anything about the format in which an SPKI certificate should be in the certificate field. However, the canonical S-expression format is an obvious choice in the sense that signatures can then be checked directly from the RR if necessary. For the same reason the S-expression should not be base64 encoded. Using base64 would also increase the size of the certificate. As stated in Section 4.3, space is at premium within DNS messages.

While there is no need for the octets within DNS messages to be 7-bit ASCII characters, zone files are typically text files, as mentioned in Section 4.4. For this reason the CERT specification also defines a text representation for the CERT RDATA. In a zone file a CERT RR could look like:

```
certs.to CERT SPKI 555 222 (
KDg6c2VxdWVuY2UoNDpjZXJ0KDY6aXNzdWVyKDQ6aGFzaDM6bm9oMTA6YTIzNDU
2Nzg5MCkpKDc6c3ViamVjdCgxMDpwdWJsaWMta2V5KDEwOmhlY2RzYS1ub2goNz
pLZXlTcGVjMTY6YjIzNDU2Nzg5MDEyMzQ1NikpKSkoMzp0YWcoMToqKSkpKDk6c
2lnbmF0dXJlKDQ6aGFzaDM6bm9oMTA6YzIzNDU2Nzg5MCkoMTA6cHVibGljLWtl
eSgxMDpoZWNkc2Etbm9oKDc6S2V5U3BlYzE2OmQyMzQ1Njc4OTAxMjM0NTYpKSk
zMjplMjM0NTY3ODkwMTIzNDU2MTIzNDU2Nzg5MDEyMzQ1Nikp )
```

The type field is represented as an unsigned integer or as a mnemonic symbol (e.g. SPKI). The key tag is given as an unsigned integer. The algorithm may be specified either by using an unsigned integer or the corresponding mnemonic, as specified in DNSSEC. The certificate itself is included in a base64-encoded form, and may be divided into white space separated substrings, which are concatenated to obtain the entire certificate. The certificate may span lines if the substrings are appropriately enclosed in a pair of parentheses. In general, multi-line RRs can be indicated by grouping data using parentheses; any line breaks surrounded by a "(" and a ")" are ignored.

5.2 Administration of Certificates

This section discusses the various issues involved in storing SPKI certificates in the DNS. These include the tasks of determining the zones in which to place a certificate, and choosing a name for the certificate. The actual act of storing a certificate in the DNS through a zone update is also addressed.

It is up to an entity that possesses a certificate to decide whether to store it in the DNS or not, and the decision should be based on the content and purpose of the certificate. Naturally, certificates to be considered for storing in a database in which updates are not instantaneous should have a long lifetime, and should not be signed with temporary keys. In some cases it is important for the storage decision to have been made prior to the time of creation of a certificate, as is also explained in this section.

5.2.1 Choosing a Storage Location

An entity wishing to use the DNS for storing its certificates should choose a zone for storing the certificates in and make arrangements for adding RRs to that zone (see Section 5.2.4). Even once that is done, it is not obvious to which party's zone a given certificate should be stored. Each SPKI certificate has an issuer and at least one subject, after all.

Different SPKI certificate types were already presented in Section 3.1. Nevertheless, for the purposes of choosing storage locations it is useful to categorize SPKI certificates differently, as I would like to use the kind of categorization which allows the selection of a storage location to be made based on the category alone. I take the categorization presented in [25] as a basis for mine, which is described below. As the names of the categories are different from the SPKI certificate types described earlier, there should be no confusion.

Trust certificates Certificates belonging to this category may be expressed as $(i, s, true, a_o, v)$, where a_o only contains rights of which none have been acquired through delegation; i.e. a_o only contains rights which i is authorized to grant without possessing any certificates. The name *trust certificate* is appropriate because i trusts someone else to decide about the use of something that is its own. i may want to keep certificates like this to itself, and use them only when verifying a chain. That allows i to easily control access to its own property.

Delegation certificates Certificates which are not trust certificates and can be denoted $(i, s, true, a, v)$ belong to this category. This category should also be used if there is uncertainty about a certificate being a trust certificate.

Permission certificates All certificates belonging to this category can be written as $(i, s, false, a, v)$ using a 5-tuple.

Identity certificates This category contains SPKI name certificates, and generally any certificates expressible as (i, s, n, v).

When considering the categories, a certificate chain beginning from the issuer of the first certificate is of the form $T?D{*}(P|I)?$, where T is a trust certificate, D is a delegation certificate, P is a permission certificate, and I is an identity certificate. Usual regexp conventions are adhered to; i.e. parentheses are used to separate elements, $a|b$ means either element a or element b, $a?$ signifies that there are 0 or 1 a's, and $a*$ signifies that there are 0 or more a's. Chains ending with I cannot be used to prove authorization. I's cannot appear anywhere else, except when used in combination with a certificate of some other category, in which case they are needed to translate a name to information that identifies an entity.

To make it possible to traverse a chain consisting of entities (nodes) and certificates (arcs), each node has to contain information about the arcs leaving from it to all the different possible directions of traversal. Hence, if forward search is to be used it is sufficient to store certificates in the issuers' domains only. The same applies for the combination of backward search and storage at the subjects' domains. However, Aura has compared different algorithms and found that a two-way algorithm has the potential to be faster than forward or backward search algorithms [1]. Thus I propose that certificate storage locations be chosen in a way which allows for the use of two-way search algorithms, instead of forward or backward ones.

To support two-way search all certificates except those which can only be in the beginning or the end of a chain need to be stored in both the issuer's and the subject's domains. From the certificate chain properties described above we can see that delegation certificates (D) and only them can appear in the middle of a chain. Thus they must be stored in two domains. Maintenance of the same records in more than one place is undesirable because of possible consistency problems, and should be avoided when possible. For this reason, certificates belonging to the other categories should only be stored in one domain. The storage locations for each certificate category can be determined from the chain properties, and are shown in Table 5.1.

Certificate	**Name server of**	
category	**issuer**	**subject**
Trust	×	
Delegation	×	×
Permission		×
Identity		×

Table 5.1: Certificate storage locations.

5.2.2 Choosing a Domain Name

Every DNS entry needs to be given a domain name. It is also necessary to know the name once the entry is to be retrieved. Names are, after all, used for indexing in DNS databases, and for any RR to be found the value by which it is indexed must be known. Naturally, the usual DNS naming conventions must be adhered to when selecting a name, regardless of whether the name is for a `CERT` RR or for some other type of RR. These conventions were discussed in Section 4.1.

Let us now consider if there are any additional naming issues that apply to SPKI `CERT` RRs in particular. SPKI certificate retrieval from the DNS can be generalized as an occurrence in which entity k_i (where $i = 1$ or $i = m$) makes queries and acquires a valid certificate chain $C = c_1, c_2, \ldots, c_{m-1} = (k_1, k_2, d_1, a_1, v_1), (k_2, k_3, d_2, a_2, v_2), \ldots, (k_{m-1}, k_m, d_{m-1}, a_{m-1}, v_{m-1})$. This also applies for retrieval of single (RRsets of) certificates, in which case $m = 2$.

Let it be so that delegation certificates concerning k_l (where $l = 1, 2, ..., m$) are stored in domain $n_{l,I}$ if k_l is the issuer, and in domain $n_{l,S}$ if k_l is the subject. Trust certificates issued by k_l are stored in domain $n_{l,I}$. Permission and identity certificates issued to k_l are stored in domain $n_{l,S}$. Other certificates are not stored in the DNS. For convenience, I often write n_l when I mean both $n_{l,I}$ and $n_{l,S}$. The two domains may be the same, but do not need to be.

In a typical case, k_1 wants to acquire and form the chain C between itself and the entity k_m at the other end of the chain in order to verify the existence of a trust relationship between it and k_m by closing the chain into a loop. Alternatively, k_m may need to acquire C in order to give it to k_1 for verification. Let us denote the entity at the end of the chain that is opposite from k_i as k_j (where $j \in \{1, m\}, j \neq i$). It should be safe to assume that k_i knows its own domains n_i prior to acquiring any certificates in C. Let us also assume that k_i knows the domains n_j. This is a reasonable assumption, as k_i wants to form a chain between itself an k_j, and it therefore must already know something about k_j; if it does not know n_j, it may be able to acquire that information by asking k_j. For chains that are not trivially short, knowledge of n_i and n_j is not enough for acquiring the whole chain. Section 5.2.3 proposes a way for making the rest of the required names possible to acquire during certificate retrieval. The proposal does not impose any naming restrictions aside from those already defined in the DNS specification.

From the above analysis of certificate retrieval it is possible to conclude that it matters little how certificates are named. n_i and n_j are assumed to be known, regardless of what they are. $n_2, n_3, \ldots, n_{m-1}$ can be acquired, also regardless of what they are. It is not even essential for the names to be unique. The only requirement really is that the domains need to belong to one of the zones for which the name server chosen for storing a certificate is authoritative for.

Nevertheless, for example in a system in which keys are frequently replaced with new ones, it would be useful to have a method of automatically generating suitable domain names for the RRs concerning the new keys. I will now consider different options for naming certificate data. As I do so, I use the symbols given in Table 5.2.

Symbol	**Meaning**	**Example value**
domain	Domain at which the zone governed by the name server used by *user* is rooted	`tcm.hut.fi`
host	Hostname of the machine used by *user*	`camphor`
user	Username	`alice`
$hash(k)$	Value unique for entity k	`9f7bc4c0563d8042` `1cafebabe5551011`
is	`i` for issuer or `s` for subject	`i`

Table 5.2: Symbols used in naming discussion.

The DNS has traditionally been used to store various attributes of different hosts. Certificates, however, often belong to programs or people rather than hosts, and thus there may be no obvious host whose domain name to use when storing certificates. Even if there were, the use of its domain name could be inefficient. As described in Section 4.3, there are certain limitations to the size of a DNS message. A host could have hundreds of users, each running several programs, and each program using many different certificates. If all the certificates were stored in the host's domain, they would all be returned in a single message when querying for `CERT` RRs. In almost all cases most of the data returned would be irrelevant to whoever made the query. An attempt should be made to choose the domain names in such a manner that as little irrelevant data as possible is returned in response to a query. Therefore domain names of the form *domain*, *host*.*domain*, or even *user*.*host*.*domain* would all be less than ideal in many cases.

Instead, it would be better to find a domain name unique to an entity that a certificate concerns. This can be accomplished by adding a label that, for example, contains a hash of information unique for the entity, a value returned by an incremental counter, or some random data. For reasons of privacy it is best to choose a label which does not directly identify any individual. Any hash algorithm should be chosen in such a way that the length of an encoded hash value is not greater than 63 bytes, which is the limitation for domain labels. By using this scheme, certificates issued to key k_{agent} could automatically be given a name of the form $hash(k_{agent})$.*domain*.

Even using the above naming method, several certificates can still get the same name. One might think that taking a hash of the whole certificate would be the best solution, but because the hash of the certificate that one is looking for is usually not know, such a naming system cannot be used.

A Proposed Naming Scheme

Usually, one is interested in certificates either issued by an entity or issued to an entity, but not both. Therefore it is possible to further reduce the number of `CERT` RRs returned by a query by choosing a different domain name depending on whether an entity is the issuer or the subject of a certificate. I suggest prepending label `i` for issuer, or `s` for subject. Resulting names are of the form *is*.$hash(k)$.*domain*.

The above order of *is* and $hash(k)$ allows one to configure a name server in such a way that any domain name ending in $hash(k)$.*domain* can be used to refer to the same set of records. Suppose one does not was to distinguish between certificates issued by and issued to k_{agent}, and gives all such certificates the name $hash(k_{agent})$`.certs.to`. Through the use of wildcards in the appropriate zone file, one can arrange for a query with a name of the form $is.hash(k_{agent})$`.certs.to` to result in the same RRset as queries named $hash(k_{agent})$`.certs.to`. The result RRs will have the name used in the query, however. The zone file could contain something like:

$hash(k_{agent})$`.certs.to CNAME a.`$hash(k_{agent})$`.certs.to`
`*.`$hash(k_{agent})$`.certs.to CERT SPKI 0 0 KDEyOmNlcnRpZmljYXRlMSk=`
`*.`$hash(k_{agent})$`.certs.to CERT SPKI 0 0 KDEyOkNFUlRJRklDQVRFMik=`

If a permission certificate is issued to k_{agent} by $k_{service}$, it is likely that no other entity wants to receive that certificate if it is interested in certificates issued to it by $k_{service}$. Thus it makes sense to store the particular certificate under `s.`$hash(k_{agent})$`.`$domain$ rather than `i.`$hash(k_{service})$`.`$domain$. Members of any certificate category should be named according to the issuer if stored in the issuer's domain, and according to the subject if stored in a subject's domain. Figure 5.2 illustrates my scheme. I believe that this naming method is, in general, well-suited for selecting a domain name for a certificate. Nevertheless, I do not suggest that it always be used as sometimes some other method may better meet the requirements at hand.

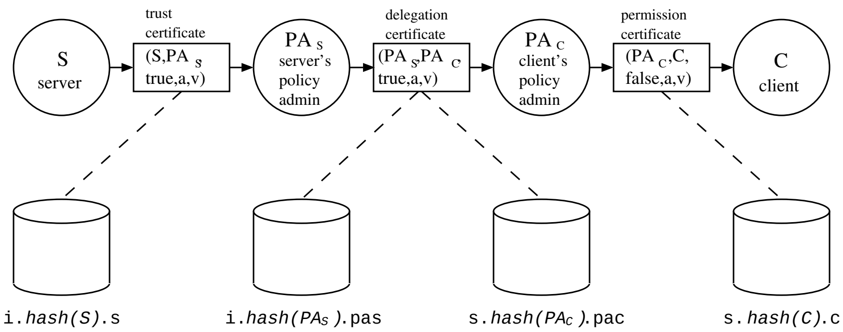

Figure 5.2: Choosing storage locations and domain names according to certificate types and entity identifiers.

Anonymity

While the above naming scheme allows *user* to construct domain names which do not reveal her username or any of her public keys, the *domain* part of a name can still be very revealing. I cannot think of a practical way to avoid this. Hence, if *user* wishes to have completely anonymous certificates, she should anonymously arrange for the use of a zone for storing those certificates, and select a zone which is not known to be used by her for any purpose.

Different Types of Certificates

Applications typically use only a certain kind of certificates, be it SPKI, X.509, or some other certificate type. Suppose that Bob has a set of certificates all concerning entity k_{agent}, and that some of the certificates are SPKI ones and some are not. If Bob places all of them to the DNS with the name `s.`k_{agent}`.bob`, they will all belong to the same RRset because they are all RRs of the same name and type (`CERT`). If Bob has reason to believe that the SPKI certificates will never be needed together with the non-SPKI ones, it is best to give them a different name, for example `s.`k_{agent}`.spki.bob`. Doing so should help reduce the sizes of answers to certificate queries. In general, to reduce the amount of irrelevant data returned as replies to queries, it is best to give a unique name to each set of certificates in which no certificate is ever needed together with any certificate that is stored in the same location and does not belong to the set.

Encoding

As there is often a need to write down domain names, for example in a zone file, labels containing true 8-bit octets should be written as printable decimal or hexadecimal numbers. Considering the size limitations of DNS messages it is better to select the more compact hexadecimal presentation. The commonly used base64 encoding would be even more efficient, but the fact that data encoded using it contains both uppercase and lowercase characters is likely to cause problems. Although the DNS standard states that case differences should be retained when processing domain names, current implementations are likely to use case insensitive string comparisons, as the names and addresses used in the Internet are traditionally case insensitive. Creating a new, efficient, printable and case insensitive encoding just for one purpose would impose a burden upon implementors, and is therefore best avoided unless found to be necessary.

5.2.3 Storage Considerations When Creating a Certificate

Unless someone can create a naming scheme in which keys uniquely define names which both spare the distributed nature of the DNS and allow people to choose which zone administrators to trust, it will be necessary to have some additional information available which allows translation from keys to domain names. Intuitively, it seems impossible to find such a scheme without being selective about which key to accept as one's public key, which is something to be avoided. Hence it seems that location transparency cannot be achieved.

One way to arrange translation of keys to names would be to build another global database that scales well, in which hashes of keys are used to index entries containing a domain name. However, then one of the motivations for attempting to use the DNS for certificate storage would be lost, namely being able to arrange the storage using existing systems.

One very viable option is to include some information about domain names in certificates themselves, and that is what I will mostly consider. Leaving domain names outside the certificates would increase the likelihood of denial of service attacks, as there would then be the possibility of attacks made by stopping someone who has a certificate from acquiring the information necessary for locating the rest of the certificates needed for a complete chain. Blocking queries to some external source of location information does not help an attacker when the information is already in the certificate. Also, it can be quickly noticed if the information in the certificate has been falsified as it is protected by the signature in the certificate.

SPKI authorization and attribute certificates include the fields `issuer-loc` and `subject-loc`, which can be used for adding URIs into certificates, as described in Section 3.2.1. The URI syntax is defined in [3]. A URI is of the form $scheme$`:`$scheme\_specific\_part$, where $scheme$ is a string that specifies the naming scheme used in the scheme specific part of the URI. Registrations of the specifiers are handled by Internet Assigned Numbers Authority (IANA). At the time of writing, no prefix has been assigned for DNS domain names. I define and adapt the use of a new naming scheme specifier, `dns`, but for the time being neglect to register it with IANA. URIs for experimental schemes may be used by mutual agreement between parties [4]. URIs following my scheme are of the form `dns:`dns_domain_name, where dns_domain_name is any valid domain name of any DNS domain name space.

The above `dns` URI syntax definition can be expanded to allow domain names to be somehow abbreviated, which may be desirable as the names can be long. I do not give a proposal for a suitable abbreviation, but may in my own implementations leave out redundant information by writing `dns:@`$domain$ to express is`.`$hash(k)$`.`$domain$, for example. $hash(k)$ should remain obvious from the context, and is can be chosen freely.

The two fields, `issuer-loc` and `subject-loc`, should be used to include the domain names of the issuer and the subject. It is not necessary to include both fields in any certificate, however, since someone who has fetched a certificate from the DNS should know which domain the certificate was acquired from. That information need not be repeated in the certificate itself. Hence certificates stored in the issuer's domain need to include information regarding the subject's domain, and vice versa. Since delegation certificates are stored in both the issuer's and the subject's domain, it may be more convenient to include both fields into them rather than to create two different certificates.

For threshold subjects, there needs to be more than one URI, one for each subject. I recommend that the URIs be listed in the same order as the subjects in the threshold. An effort should also be made to list the domains for all the entities that possess a name given as a subject of an attribute certificate. In that case the order of the URIs is not significant, however. If such a list is unavailable and cannot be created, an URI from which the information can be acquired could be included instead. The information would probably be maintained by the owner of the name space.

Name certificates do not have and do not need to have `issuer-loc` and `subject-loc` fields, because they are not by themselves useful in making authorization decisions. They can only be used to prove authority together with authorization or attribute certificates, and those certificates may contain the location information. [1]

According to the current SPKI certificate specification, the location information can only be placed within the certificate body which will be included into a signature calculation in its entirety. Thus any changes to location information of a certificate cannot be put into effect without issuing a new certificate. The old certificate should then be revoked. Certificate revocation is not a trivial problem, however, and thus it is advisable to try to ensure that the location information used is such that it stays valid for the entire validity period of a certificate. When in doubt, the validity interval should be set to be sufficiently small. Incorrect location information could cause wasted effort to those looking to form a certificate chain, although that should not cause actual denial of service for as long as properly implemented search algorithms are used and the network can handle the additional workload.

Increasing the Lifetimes of Domain Names

The lifetimes of URIs can possibly be increased by using aliases. Suppose there are several certificates stored in the domain `certs.fr`, and that for some reason certificate maintenance can no longer be done for the zone the domain resides in. By adding a `CNAME` RR with the name `certs.fr` and the value `certs.to` one can move and continue maintaining the certificates in the domain `certs.to`.

Suppose that the maintainer of the zone containing `certs.fr` does not agree to add the `CNAME` RR. Then there is not much that can be done except for re-

[1] Acquiring a chain to prove a name could fail, however, because the issuer domain of the last certificate of the chain would possibly not be contained in any of the certificates stored in the domain of the subject of the chain. That would cause a backward search to fail.

placing all certificates affected by the address change. It is, however, possible to take precautions against something like this by using so-called forwarding services that provide a "permanent" address for their clients. Currently, there is at least a service that will forward HTTP requests sent to one domain to whichever domain the service user has configured them to go, as well as a service which provides a user with a DNS domain name and allows the corresponding IP address to be dynamically changed. There could also be a similar service which would assign each client a "permanent" zone and allow the name servers responsible for a zone to be changed upon request. A certificate creator would then have a practically unlimited number of "permanent" domain names to use in certificates and the freedom to choose any available name servers for maintaining the zone information on.

URIs Outside Certificates

One might also consider storing the URIs within `CERT` RRs, but outside the certificates themselves. The same pieces of information would still be contained within `CERT` RRs, but the location information could be changed without replacing certificates. The current `CERT` RR definition does not contain suitable fields for URIs, but that is not a problem as the certificate portion of a `CERT` RR may contain type-specific internal subfields. Integrity of the URIs, when not stored within certificates themselves, could not be guaranteed without using `SIG` RRs, however.

Certificates Outside the DNS

In this section I have assumed that all certificates needed for a chain are stored in the DNS. If some certificates have not been placed there, they need to be available from somewhere else, or it might not be possible to acquire all of the other certificates belonging to a chain. Other locations and protocols to use for fetching certificates can also be specified by including URIs in certificates.

5.2.4 Updating a Zone

A certificate must be packed into a `CERT` RR before it can be added to a zone. The fields to be filled in were covered in Sections 4.2 and 5.1. I propose that if an SPKI certificate is to be stored in its issuer's domain, the key tag be calculated from the subject field, and vice versa. If the field to be used does not contain a single key which can be converted into a standard format for the calculation, then the tag value should be set to zero. If the field contains a hash of a key, we recommend that the key be acquired for the calculation if possible.

All the data in a DNS database is stored in name servers. To add a `CERT` RR to the database one needs to have the authority to make updates, either directly or through some other entity, to the data of the primary server of the zone to which a certificate is to be added. Different methods of making updates were considered in Section 4.7.

One update method is to ask the administrator of the zone requiring an update to edit the zone information and to notify all authoritative servers about the change. Such a solution could consume all of the administrator's available time. The zone maintenance needs introduced by frequent adding of certificate RRs to zones, as well as removal of expired and revoked certificates, are likely to be considerable. Some form of automatization is almost essential.

Ideally, every user that wants to make updates to the database would be assigned a part of the domain name space to administer and given a key with which to sign requests in order to initiate updates. It currently looks like there will a DNS dynamic update standard that offers a practical solution for doing so once widely implemented.

In any case, merely managing the zone maintenance rights given to users is likely to be a large task, and may require further automatization and administrative tools. If maintaining a secure system requires too much work, people will sacrifice the security for convenience. For this reason it is very important to design a convenient-to-use administrative interface to any system that makes extensive use of certificates and involves administrative duties. Designing such an interface could among other things include finding a way to visualize trust relationships and security policies. Usability is an area that warrants a lot of further research, but it is beyond the scope of this thesis.

5.3 Retrieval of Certificates

Communication with name servers is handled solely through sending and receiving DNS messages. To acquire all the certificates stored in the domain `certs.fr`, one would first need to construct a message in which the question section has an entry with the name `certs.fr` and the type `CERT`. The message should be sent to the name server authoritative for the domain. That server is not always known, and needs to be tracked down prior to asking for the certificates. A server offering recursive service can be asked to do so; otherwise the task is left to the client. Once the name server information is known, and possibly cached as an `NS` RR, the certificate request can be sent. The request then either times out, or a reply is received. The reply can indicate an error or that no certificates matching the search criteria exist, or it can contain a set of `CERT` RRs.

Once an RRset has been acquired, the desired certificates must be picked from the set. Throughout this section, I assume that the naming and storage location schemes proposed earlier in this chapter are in use. Suppose we want all the SPKI certificates in domain `i.`$hash(k_1)$`.certs.fr` with the subject k_2. We should first choose those `CERT` RRs which have the type `SPKI`. From them we can quickly rule out those RRs which have a non-zero algorithm identifier and whose key tag value is not the same as that calculated from k_2. It should be noted that use of key tags is not compulsory, but can make the selection process faster. The subject fields of all the remaining certificates must be checked to see if k_2 is included. After the correct RRs have been selected, what remains to be done is to extract the certificates themselves from the RRs.

5.3.1 Caching

Once an answer to a query has been acquired, the results can be cached to improve the performance of further queries. Resolvers usually do caching, as mentioned in Section 4.5. Caching can cause problems, however, unless the zone administrator puts some effort into selecting and updating the time-to-live values associated with RRs. The TTL value can be defined separately for each record in a zone. If a change to some RR is anticipated, its TTL value should be kept low enough so that it will expire in caches before the actual change is done.

Choosing the maximum time for which caching is allowed for `SPKI CERT` RRs is different from doing the same for most other types of RRs, in the sense that an SPKI certificate itself already contains an explicit statement that describes when the certificate is valid. If a zone administrator takes that information into account when maintaining a zone – perhaps using a script or intelligent server software capable of interpreting SPKI certificates – the TTL values could be made to quite accurately correspond to the actual lifetimes of the certificates. Frequent zone updates would be required, however, and possible revocations of certificates are not accounted for by this scheme.

Another possibility would be to have resolvers interpret the contents of `SPKI CERT` RRs before caching them. However, it could be argued that the task of a resolver is not to interpret the contents of RRs which do not contain database indexing information. In any case, adjusting TTL values in the server end will require less network traffic and support the use of thin clients.

The simplest solution is not to consider certificates' validity information in either zone maintenance or resolvers. If the client then notices that she has received expired certificates from a resolver, she should have some way to acquire more recent data. Some resolvers allow the user to specify an option which indicates that returned records should come from an authoritative source; this option can be used to avoid receiving cached data.

5.3.2 Search Algorithm

As proving a permission requires an entire chain of certificates, it is useful to have an implementation of an algorithm that is able to obtain an entire chain of certificates from the DNS available for applications to use. Such an algorithm could possibly be contained within a resolver, or built on top of it into a library of its own. I present a suitable algorithm below. It is based on the two-way search algorithm presented in [25], and attempts to find a suitable path from a directed delegation network formed by a DNS database containing certificates. A path in such a network can be directly translated into a chain of certificates.

It should be noted that there may not be any one certificate chain that alone proves all of the required rights, but more than one chains together might do so. The algorithm presented here does not account for this, and in such a case returns no chains at all.

For simplicity, it is also assumed that there are no threshold certificates in the database. Intuitively, it should be possible to modify the algorithm to obtain proof containing threshold certificates, but the proof would not always be representable as a chain. Aura has presented as similar algorithm that does handle threshold certificates [2].

Symbol	**Meaning**
k	identifier of the current entity
n	base domain name of the current entity
k_v	identifier of the verifier
n_v	base domain name of the verifier
k_s	identifier of the final subject
n_s	base domain name of the final subject
T_{fs}	forward search tree
T_{bs}	backward search tree

Table 5.3: Constant and variable definitions for the algorithm.

Forward search

1. Set $k = k_v$ and $n = n_v$ and T_{fs} to an empty tree.
2. Fetch all certificates using the resolver and the name `i.`n.

3. Filter out all certificates that are irrelevant (delegation rights, authorization, validity).
4. Filter out all certificates whose subject is already present in T_{fs}.
5. For all certificates whose subject is k_s, check the signature. If the check fails, filter out the certificate (and issue a warning). Otherwise add the certificate to T_{fs}, and return it (T_{fs} has no branches, and is therefore still a chain) indicating success.
6. If there are no more certificates left, jump to backward search, but only fetch certificates from the DNS once. If that is not enough, fail.
7. If there is only one certificate left, check its signature, mark it as checked, add it to T_{fs}, set k to its subject and n to its subject's location. Jump to step 2.
8. The search tree branches. Add the remaining certificates in the current fetch set to T_{fs}. Jump to backward search.

Backward search

1. Set $k = k_s$ and $n = n_s$ and T_{bs} to an empty tree.
2. Fetch all certificates using the name `s.`n.
3. Filter out all certificates irrelevant to the current search problem.
4. Filter out all certificates whose issuer is already present in T_{bs}.
5. For all certificates whose issuer is present in T_{fs}, check the signature. If a check succeeds, check the signature of the found target if marked unchecked. If either one of the checks fail, filter out the certificate (and issue a warning). Otherwise construct a chain by traversing T_{fs} backward from the target certificate to k_v, inserting each certificate into the beginning of the chain. Append the current certificate, and then the current T_{bs} search path, also reversed. Return the resulting chain and indicate success.
6. If there are no more certificates left, terminate and indicate failure.
7. Add the remaining certificates to T_{bs}.
8. Select one of the leaves of T_{bs}, setting k to its issuer and n to its issuer's location. Memorize the current T_{bs} search path (for example on stack), then continue from step 2.

Typically existing certificate paths are found quickly, but negative answers can take even millions of steps [1]. This is because the algorithm presented above performs a complete search. In large delegation networks such a search can be too expensive, and some heuristics are required to decide when to terminate the search. Heuristics can also be applied to selecting the most likely successful search path when the backward search tree branches. Both kinds of heuristics that are directly applicable to the algorithm given here are presented in [25].

Chapter 6

Resolver Implementation

I have created a DNS resolver which can be used to retrieve `CERT` RRs, and which can be readily integrated with the TeSSA architecture implementation as a new component. This chapter begins describing the resolver by first covering the goals set for the resulting implementation before the design work was started. It then continues by describing the interface and the high-level structure of the resolver, and explains the major design choices made as well as the reasoning behind them. The chapter ends with a short description of how the resolver was tested.

6.1 Functional Requirements

The requirements for the resolver, based on the needs of the project, were decided to be the following:

- Implements basic DNS resolver functionality, supporting RR types `A`, `CNAME`, `MX`, `NS`, `PTR`, `SOA`, `TXT`, and `CERT`.
- The class `IN` is fully supported. Other classes currently need not be supported, but should not require change of design if later added.
- Is able to pursue answers recursively.
- Performs caching of RRs to avoid long delays fetching frequently requested records. The user interface should, however, offer an option to disable cache lookups by allowing clients to demand authoritative replies. This option is useful at least if the resolver returns expired certificates, as explained in Section 5.3.1.
- Fits well within the existing system of TeSSA, i.e. is built on top of the JaCoB framework (see Section 6.2.2), like the other protocols of the TeSSA architecture implementation.

- Facilitates handling of SPKI certificates by having an interface that returns the certificates contained within `SPKI CERT` RRs as Java objects.
- Is capable of acquiring and returning entire chains of the aforementioned objects to allow convenient acquisition of proof of rights.

6.2 Design Goals

The requirements given in Section 6.1 only concern those features of the resolver that are visible to the user. It is in the best interest of a software implementor to also establish other goals that help to keep his or her work pleasant. When the component to be built is too big to allow the work involved to be easily grasped and estimated, it is best to try to divide the work into smaller pieces. Constructing a small component is rarely a daunting task, and progress is easier to follow as each piece gets completed. Smaller, more generic components are also more likely to get reused.

I thus chose myself the goal of building the resolver by using simple and easy-to-understand components as building blocks. I also wanted to apply design patterns in my work to ensure that I make use of those design choices that have been found to work well in the past. The task of choosing suitable patterns had already been done by the designers of the JaCoB framework (described below). Creating my design the "JaCoB way" forced me to rigorously utilize design patterns. In order to be more certain that the implementation corresponds to the design, I also decided to include some statements in the code with the sole intent of revealing errors in the program logic.

6.2.1 Design Patterns

When examining well-structured architectures one may notice recurring patterns, which are simple and elegant solutions to specific design problems. Typically, they are the kind of solutions that people do not tend to discover until after some redesign cycles. The authors of [16] attempt to capture design experience by presenting several of such solutions in a form they call *design patterns*. The purpose of a design pattern is to name and describe a solution to a general design problem in a particular context, and to do so in a way which allows people to apply the solution to their own designs effectively without having to re-invent it. Utilization of design patterns in software development tends to result in collections of small, highly regular classes and communicating objects.

6.2.2 The JaCoB Framework

The Java Conduits Beans (JaCoB) framework [24] applies design patterns heavily. The framework is intended to provide developers with some of the non-protocol-specific functionality that is commonly found in cryptographic protocols. Our implementation of the framework is written completely in Java, and takes advantage of the language-level security features of Java 2 to add to the security of the JaCoB environment.

Protocol architectures built using the framework consist of software components called *conduits*. There are five kinds of conduits, which are shown in Figure 6.1. By connecting conduits to each other one can construct entire protocol stacks with highly complex functionality. Such a system is made dynamic by the different states that Sessions may have, as well as the information that flows through the system in the form of *messages*.

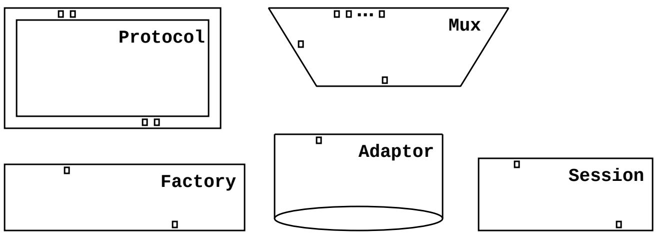

Figure 6.1: The five kinds of conduits provided by the JaCoB framework.

6.3 Functional Description

This section describes how external entities can interact with the resolver, and what they can accomplish by doing so. The preparations necessary before the resolver is ready for use are covered first. After that, the services provided by the resolver are described generally, without going into method-level detail in explaining how they are used. As applications access the resolver through its interfaces, the functionality available through each interface is covered separately.

6.3.1 Instantiation and Initialization

Before the resolver is ready for use, an instance of it must be created and appropriately initialized. To provide the resolver with the information it needs to pursue answers to queries, its cache must be initialized to contain the names and IP addresses of the name servers to use. These are provided in the form of `NS` and `A` RRs. Records of other types can also be cached if desired.

The same instance of the cache is shared by all resolver instances in the same (virtual) machine, and it is thus sufficient to perform the initialization once per execution of the machine.

The resolver depends on an underlying protocol stack as a means to communicate with name servers. As soon as the answer to a query cannot be found from the local cache, the resolver needs to already be attached to a UDP protocol component. To hook the resolver up with UDP it must be passed a reference to an InstallationAdaptor instance that is capable of creating a session with UDP, as well as the UDP port that the resolver should listed to.

Unlike most resolver implementations, this one has no configuration file. As the resolver is rather simple and does not have many options, I felt that it would not be too much trouble for application programmers to have their applications pass the required pieces of information to the resolver through method calls. All of the necessary information can either be passed to the constructor of the resolver during instantiation, or later by calling methods dedicated for supplying the information.

6.3.2 Interfaces

The resolver has multiple interfaces, of which an application developer is free to use any or all. The lowest-level interface is the DnsResolver conduit itself. On top of that, there are components that add to the core functionality and have interfaces of their own. Those components that one does not desire to use need not be instantiated. The higher-level components depend on the lower-level ones, however. The interface layers, ordered from highest-level to lowest-level are shown in Figure 6.2.

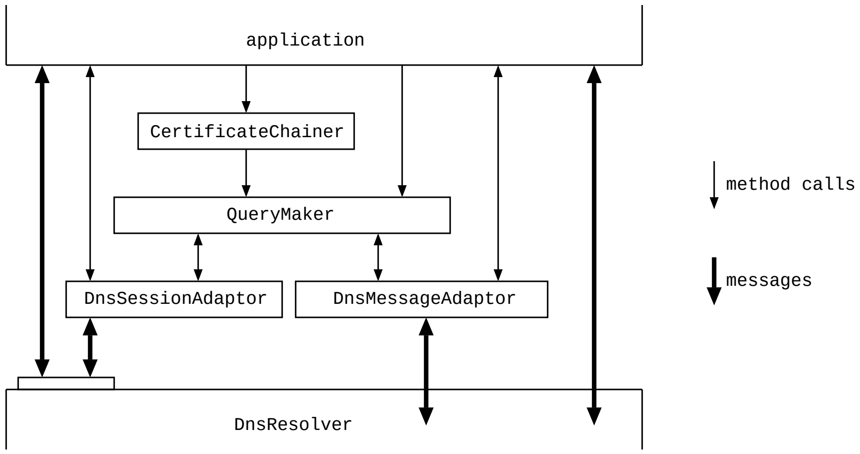

Figure 6.2: The interfaces of the resolver.

DnsResolver

This Protocol conduit contains the logic of constructing and interpreting binary messages understood by name servers, recursively pursuing answers to queries, checking and updating the cache, as well as the creation and termination of sessions upon request. Applications can communicate with the DnsResolver conduit by sending messages containing DnsMessage, InstallMessenger, or UninstallMessenger instances to it. The DnsResolver will send replies as messages that contain either a DnsMessage or a NotifyMessenger, along with other data.

DnsSessionAdaptor

This Adaptor conduit facilitates the creation and termination of sessions with the DnsResolver. With a DnsSessionAdaptor attached to the "top" side of DnsResolver, session requests can be made by calling the methods of the adaptor. Notification of completion or failure of the operation can be requested by supplying a reference to a class instance that implements the callback method to use.

DnsMessageAdaptor

The purpose of this Adaptor conduit is to allow sending and receiving of messages between an application and a DnsResolver simply by passing DnsMessage instances to method calls. Receival of messages is optional, and is accomplished through callback methods. A DnsMessageAdaptor can only be used for sending and receiving messages when a session with it and a DnsResolver exists.

QueryMaker

This component provides a typical resolver/application interface. Such interfaces tend to have at least three functions: domain name to IP address translation, IP address to domain name translation, and a general lookup function for retrieving RRs of any type. QueryMaker has a method for each of the above functions. In addition it also provides a method for retrieving SPKI certificates as Java objects. The method filters out `CERT` RRs of types other than `SPKI`. It accepts an optional key tag value, which, if provided, will be used to determine which SPKI certificates should be included in the result. The use of key tags during retrieval was explained in Section 5.3. Each of the mentioned methods expects domain names to be absolute; no names relative to the local or any other domain should be used. Each method allows the caller to demand an authoritative reply.

QueryMaker is not a conduit, and thus uses Adaptor instances – in this case DnsSessionAdaptor and DnsMessageAdaptor ones – to interface with the conduit system. A reference to a DnsSessionAdaptor must be provided upon instantiation. QueryMaker uses the Adaptors to start a session, send a query, receive a reply, and finally to end the session.

CertificateChainer

This class facilitates retrieval of entire chains of certificates by implementing the algorithm described in Section 5.3.2. Each time the algorithm is executed, it must be provided with the following information: issuer identifier, issuer location, subject identifier, subject location, and the permissions that the chain must prove. Validity will be checked according to the system time at the time of execution. CertificateChainer depends on QueryMaker to acquire the certificates needed for forming the chain.

6.4 Internal Structure and Implementation Details

The reason for using the JaCoB framework to implement the resolver was given in Section 6.1. This section presents the conduit structure of the resolver, which is illustrated by Figure 6.3. Every one of the conduits the resolver consists of are not described in detail, but details are given when they are felt important for understanding how the resolver works.

When a client wants to establish a session with the resolver, it should first create a conduit, e.g. a DnsMessageAdaptor. The client can then request that the created conduit be attached to the resolver by sending a message to side B of the DnsResolver. The request will be received by a ConduitFactory instance, which will create a new DnsSessionProtocol and attach it to the conduit provided by the client. Sessions are terminated in a similar manner. The current implementation allows there to be a maximum of 256 simultaneous sessions. The internal structure of DnsSessionProtocol is shown in Figure 6.4.

A query sent by a client is directed to a DnsSessionProtocol instance, which only has one internal conduit. The conduit is called QuerySession, and it contains the logic required for pursuing an answer to the query. Its states as well as possible state transitions can be seen from Figure 6.4. The picture also shows which states access the cache.

InitialState initializes the session, as well as re-initializes it if a `CNAME` RR is encountered. ChooseServersState selects the name servers to send queries to by choosing the most promising `NS` RRs among those in the cache. Cache-

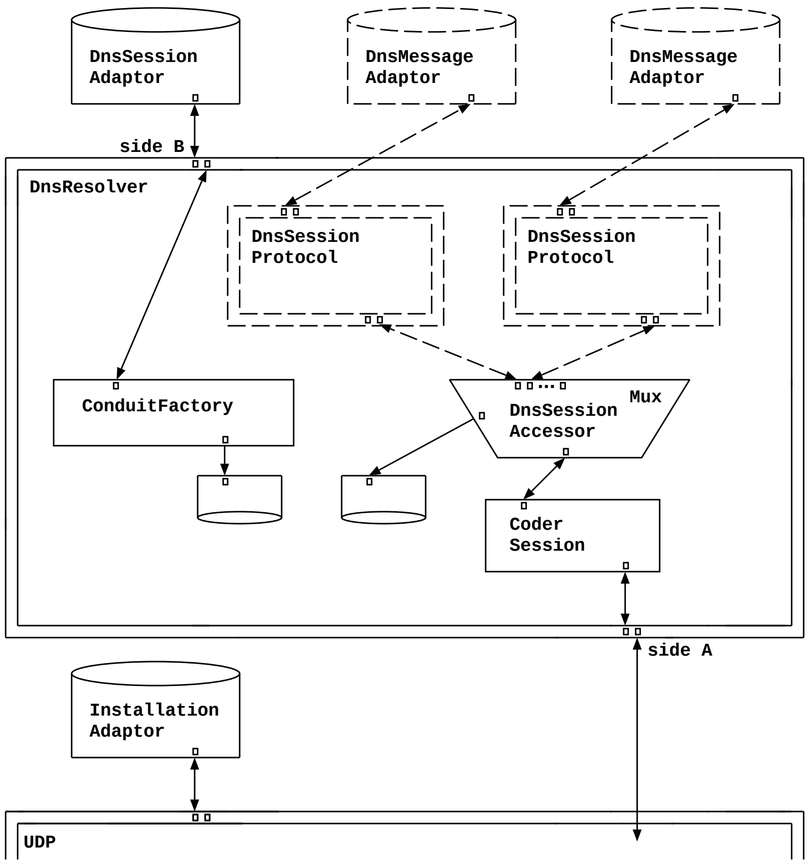

Figure 6.3: The conduit structure of the resolver.

State checks if an answer to the query can be found from the cache, and if so, returns it to the client. QueryState sends and receives messages between name servers, and also handles timeout situations. AnalysisState analyzes messages received from name servers, and decides what to do next.

There can be at most one outstanding query per QuerySession, because it has instance variables that are used to store information regarding the query being processed, if any. A client must either create a new session for each query it wants to make, or it must at least wait for a reply from QuerySession before sending a new query to it. It would have been possible to implement the session in such a manner that there could be multiple outstanding queries, but for simplicity I chose to support only one.

The task of the Mux within DnsResolver is to direct incoming messages to the correct session. To do this, the Mux lets its DnsSessionAccessor instance to check the eight most significant bits of the ID field within the header of the incoming message (the ID field was mentioned in Section 4.3). Each session has its own unique ID, which directly corresponds to the aforementioned bits. Thus the Mux can easily choose the session to which it should forward

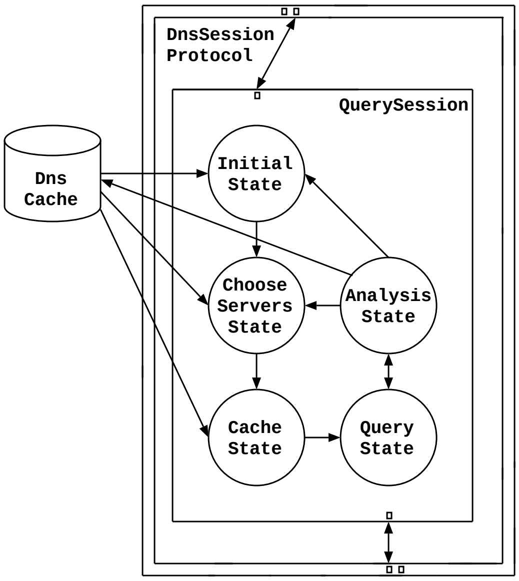

Figure 6.4: The internals of DnsSessionProtocol.

incoming messages. It is possible that there is no matching session; in that case the message ends up in the Adaptor that is connected to the Mux, which simply drops the message. Delays in the network may cause the arrival of an answer to a query to be delayed long enough for the session for which the message was meant for to have already been terminated. The DnsSession-Accessor also has the task of writing the appropriate session ID to all outgoing messages. The least significant eight bits in the message header ID field are used by QuerySession to distinguish between the various messages sent and received between itself and a name server.

The purpose of CoderSession is to translate the message content between DnsMessage object and byte array formats. This allows the conduits above CoderSession to manipulate DNS messages as Java objects. Underlying protocols, on the other hand, have binary encoded data readily available for inclusion in UDP packets, for example. This resolver implementation is meant to be attached directly on top of a UDP protocol conduit.

6.5 Testing

The resolver was initially tested as a stand-alone unit, without the underlying protocol stack. This was accomplished by creating a dedicated Adaptor to simulate the network. It was thought best to check what kind of messages would be going to name servers before actually letting them go there.

The resolver is also faster to launch when other protocol layers need not be instantiated along with it.

Once the output of the resolver looked sensible and the input handling seemed to work, UDP, IP, ARP and Ethernet protocol conduits were instantiated and attached to a DnsResolver. The resulting system was first tested with a simple program to send packets with in order to confirm that packets were getting transferred correctly. After that the resolver was considered ready for testing with name servers.

I was prepared to take an existing name server and to add support for `CERT` RRs to it, but that proved to be unnecessary as version 8.2 of BIND, which does support certificates, became available in time for testing. Minor changes to the code were required, however, as the `CERT` RDATA zone file loading code was recent and still contained errors that needed to be fixed. Slight modifications were also made to accommodate the use of larger UDP packets than the DNS standard allows. This was done because there was no JaCoB implementation of TCP available for attaching to the resolver component, and the DNS UDP packet size limitation would have been too severe to allow proper testing using SPKI certificates.

The modified server software was loaded with a zone file containing SPKI certificates, and run on a machine that was claimed to be authoritative for the zone in fake name server information given to the resolver. Queries were then made for both individual RRsets and chains of certificates. Some of the test results, along with analysis, follow in Section 7.1.

Chapter 7

Evaluation and Consideration

7.1 Evaluation of the Resolver Implementation

The resolver that I implemented is rather simple, and does little more than meets the requirements stated in Section 6.1. It has been successfully used to fetch SPKI certificates from the DNS. The code appears robust when handling typical queries and answers, but error handling has not been completely implemented yet. Thus error messages or erroneous data returned by a name server are likely to cause the resolver to fail.

The amount of time spent locating errors in the code was relatively small compared to the actual writing of the code. I feel that this is largely because of the design goals taken, which were mentioned in Section 6.2. The number of flaws – when compared to the amount of code written – was slightly smaller than I had anticipated, and some of them were immediately revealed as they triggered a sanity check embedded in the program.

The earlier certificate repository implementation of TeSSA was local. Now that the DNS resolver is ready for use, the local repository can be replaced with the DNS to allow for distributed management of information regarding trust relationships. As mentioned earlier, one goal of TeSSA is to use existing standards and solutions whenever there are suitable ones. It is therefore natural to implement the repository within the DNS, as the DNS is a widely used and standardized distributed database. A certificate repository is an integral part of the TeSSA architecture, as portrayed in Figure 7.1.

The trust and policy management infrastructure of the TeSSA architecture has been implemented using SPKI [27]. The current implementation replaces the default class loader of Java 2 with one that uses SPKI certificates to determine the permissions of a Java class as it is loaded. Integrating the resolver with the SPKI component will thus facilitate distributed administration of Java 2 permissions. The integration work remains yet to be done.

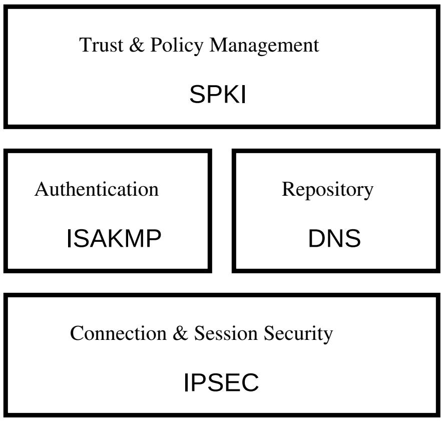

Figure 7.1: Telecommunications Software Security Architecture.

The authentication protocols of the TeSSA architecture are based on Internet Security Association and Key Management Protocol (ISAKMP) [20]. The ISAKMP protocol component is currently being completed [31]; there is no intention for it to access the certificate repository directly. Connection security of TeSSA is to be achieved through the use of the Internet Protocol Security (IPSEC) protocols, but there are no concrete plans of including IPSEC in the current architecture implementation.

Certificates can be distributed between local repositories used by the SPKI component, JAR files containing classes to which the permissions in the certificates pertain, as well as various DNS domains. The administration tasks are then divided between maintainers of systems running a JVM, creators of Java classes, and DNS zone administrators. However, as we have not implemented a name server, and the resolver does not presently support making dynamic updates, the management cannot yet be done through the use of our architecture implementation alone. For now, DNS updates have to be made through other means, several of which were covered in Section 4.7. I believe that support for dynamic updates could be added to the resolver with reasonable effort. Nevertheless, unless there is a pressing need to add such support, it is best to wait until dynamic updates have been standardized and are supported by the latest name server implementations.

Performance

In the design and coding of the resolver, no great effort was put into optimizing it for speed. While its performance is adequate for the tasks it has so far been used, there probably is a lot of room for improvement. The resolver, along with the underlying protocol stack, takes several seconds to launch. However, once the initialization process is complete, the resolver usually returns answers to queries reasonably fast, taking roughly a second per name server it needs to consult. As no name servers need to be consulted when the answer is in the cache, caching yields noticeable time savings. Indeed, caching is important to avoid long delays fetching frequently used certificates. Avoidance of such delays would be practically essential if wide adoption of public key infrastructures resulted in certificate queries being as common as network address queries.

Search Algorithm

The certificate chain search algorithm of the resolver has not yet received extensive testing, but it nevertheless has already been used to acquire and construct chains. It seems to be functioning correctly, but I believe that its performance could be improved considerably. This could be done for example by implementing the heuristics described in [25]. Another way would be to have the algorithm examine several possible branches simultaneously by creating multiple sessions with the resolver. Currently the algorithm spends most of its time waiting for the resolver to return data. The idle time could be reduced by making new queries and processing answers to other queries while waiting for a response to a query.

In order to function correctly, the search algorithm needs to be able to tell which certificates contain permissions greater than or equal to those requested by the user of the algorithm. The comparison routine used for the task is the one implemented for the SPKI component. It is capable of comparing sets of Java permissions. Support for other kinds of permissions will need to be added as required. Online checks possibly specified within SPKI certificates are currently not performed at all. Support for them will not be included in the TeSSA architecture implementation before the ISAKMP component has been completed. This is because ISAKMP will be used for exchanging certificates and their validation messages [31].

The JaCoB Framework and the Protocol Stack

The JaCoB framework implementation is still a work-in-progress, and does not even work completely according to its design [31]. Major changes to it are likely to require changes in the resolver as well.

While the current protocol stack implemented using JaCoB is otherwise well-suited as a foundation for the resolver, it does have one serious shortcoming. TCP is not implemented at all. This was a problem as the DNS standard states that UDP replies of over 512 bytes are truncated, and that the complete reply should then be acquired through the use of TCP. It was therefore essential for all replies to fit into UDP packets, but in most cases even a single `CERT` RR containing an SPKI certificate does not fit into a single 512 byte packet. For this reason I was forced to modify some name servers to have them send replies of any size within UDP packets, without truncating them. Needless to say, implementing TCP would have been preferable, but there were no resources to allocate for the task.

7.2 The DNS as a Certificate Repository

As I worked out the details of using the DNS as an SPKI certificate repository, I noticed no properties of the DNS that would make it unusable for the task of storing SPKI certificates. Nevertheless, there is one significant problem that requires effort to work around. This fundamental problem is that the need for names introduced by the DNS conflicts with the principles of SPKI. One of the ideas behind SPKI is that the use of unique names other than public keys is avoided. Resource records, however, do need to be given a name of a particular form. There often is no natural way to choose such a name for a certificate. If a global database for storing certificates were to be designed from scratch, the indexing method should be chosen differently from that used in the DNS.

The number of certificate entries stored in name servers could be much larger than the number of entries containing information about hosts. This is because certificates typically concern people or programs instead of hosts, as explained in Section 5.2.2. The fact that the DNS was designed primarily for storing information about hosts gives raise to the question if the DNS will scale well enough to also accommodate certificate storage. There is little reason to believe that it would not, as new name servers can be added if the workload becomes too great and cannot be more evenly divided among existing servers.

Some limits are set by what individual servers can handle, but these limits are not very severe. Servers authoritative for the `com` zone have thousands of RRs, and judging from their response times they are clearly capable of handling that amount of data well enough. Advances in database technology, increasing processing power and growing memory sizes should push the limitations even further. Nevertheless, already for reasons of limitations in DNS message sizes it is best to minimize the number of certificates stored in the same domain. Ways to do that were covered in Section 5.2.2. RRs that need not be stored in the same domain can be stored in separate name servers.

While the increase in DNS traffic caused by certificates might not be a problem for dedicated name servers, there still are the clients to consider. Let us suppose that there is a multi-user host that, on the average, launches an application 10000 times a day. Let us further assume that each time an application is launched, one certificate that is not stored together with the application is needed. If no caching is performed, the DNS needs to be accessed each time. To acquire a certificate at least one UDP packet must be sent and at least one must be received, which results in at least 20000 UDP packets of traffic. Suppose that in some cases more than one certificate is needed, or more than one queries need to be made to acquire a certificate, and that the above result needs to be multiplied by 10 to get a good estimate. Still, even 200000 packets worth of extra traffic should be quite a reasonable amount, at least if the DNS accesses are distributed somewhat evenly during the day.

Now suppose that the client does do caching; then it could be that certificates only need to be fetched from the DNS when an application is run for the first time, to see if it has the permission to run on the machine, or when a certificate in the cache has expired. Caching should thus make the above estimates considerably smaller. If a system were to need certificates regularly and for many different purposes, as a part of its normal operation, caching might well be essential to keep the DNS traffic on an acceptable level.

It is also worth considering if the increase to the workload of a host that results from creating, verifying, and managing a large number of certificates is acceptable. That is beyond the scope of this thesis, however.

While the DNS does appear to be suitable for storing SPKI certificates, there are cases in which it does not make sense to store certificates in the DNS, or in any other distributed database for that matter. Privacy and lifetime issues have already been mentioned. In addition to those, it is also worth considering the contexts in which a certificate is needed. For instance, if a certificate is only needed together with a Java class, storing it together with the class (e.g. in a JAR file) is likely to be the most efficient solution. It may even be that only the creator of the class knows what permissions an instance of the class needs to perform its tasks; thus it is natural for the creator to issue the class a certificate that specifies those permissions, and to include it with the class when distributing it. As another example, if a certificate is created by and only needed locally by a verifier, then it often makes sense for the verifier to keep the certificate to itself if the space allows.

The kind of SPKI certificate whose storage in the DNS is the most beneficial is a delegation certificate not issued by the verifier. The verifier may not even know of the existence of such a certificate, and it is thus not natural to arrange for the verifier to store it. Also, such a certificate does not only concern its subject, but indirectly also all those entities to whom the subject has delegated some of the permissions in the certificate. It is thus not natural to store such a certificate with its subject either. From the DNS it can be accessed by all the entities that need it.

7.3 Future Work

The earlier sections already presented evaluation of my resolver implementation and certificate storage scheme, suggesting improvements when appropriate. This section gives suggestions for research and standardization work that could help improve the efficiency or usability of the solutions presented in this thesis.

Both the SPKI and the DNS specifications are still evolving. Modifications or additions to them may necessitate changes in the certificate storage scheme described in this thesis, and naturally also to any existing implementations. When considering certificate retrieval from the DNS, it would be particularly beneficial to have the DNS standard revised to allow the use of larger UDP replies, as explained in Section 4.3.

At present, public keys are typically longer than 1000 bits. Smaller key sizes – and thus also certificate sizes – would be useful. This especially applies when placing certificates in size-restricted environments, such as DNS UDP messages or smart cards. Elliptic curve cryptography looks promising, as it appears to offer drastically shorter key lengths than the public key algorithms currently in common use, while attaining a similar level of security. Elliptic curves have been known for a long time, but applying them to cryptography is something relatively new. More research is required before one can be reasonably confident in the security of elliptic curve cryptosystems. Even less research effort has been put into hyperelliptic curve cryptography, which appears to allow the use of even shorter keys. Both kinds of cryptography are planned to be utilized in the TeSSA implementation. An elliptic curve implementation already exists, but it is yet to be integrated to the rest of the system [17].

Even once implementations of secure distributed computing architectures that offer good support for managing certificates are available, there is still a lot of work to be done in designing and developing applications for them. No matter what the applications are, their implementations should be easy to use. The importance of usability was addressed in Section 5.2.4. Considering certificates in general, there are likely to be uses for them that have not been thought of yet. So far authorization certificates, for instance, have received little attention.

Chapter 8

Conclusions

Societies are increasingly dependent on fragile information infrastructures. Yet, it is hard to develop and maintain information security in distributed settings. In making delegation and access control decisions, it is important to have up-to-date security-related information. Trust, policy, and authorization information can all be represented and managed as certificates, which allows its integrity and authenticity to be verified. As such information tends to be dynamic in nature, it is important for its management to be flexible as well. If not, the result is likely to be either a heavy administrative burden or weak security. To achieve convenient policy management in a distributed setting, it seems natural to use a distributed database, and the DNS is a widely applied distributed database solution.

In this thesis, I have tried to provide a reasonably complete and detailed coverage of how to store and retrieve SPKI certificates using the DNS. I started with information from available sources, and proceeded to fill in the gaps with solutions of my own. The theory got refined and applied to practice as I implemented a resolver with certificate support and experimented with it. In addition to the certificate storage theory, I included a description of the resolver implementation in this thesis, albeit not in a very detailed manner.

The SPKI specification does not define a certificate repository. The current IETF proposal that concerns the storage of certificates in the DNS mostly addresses X.509 and PGP; it does not elaborate upon SPKI. Some other literature on the topic exists, but there still were many previously unaddressed details to work out in determining exactly how to make efficient use of the DNS as a certificate repository. Such details include the naming of certificates in a way that supports efficient retrieval, as well as deciding where to keep the naming information to make it possible to acquire entire chains of certificates without knowing all of the domain names in advance. In this work I also refined an earlier scheme that defines how to choose the DNS node in which an SPKI certificate should be stored.

The experiences gathered from the work done on this thesis support the notion that the DNS can be adopted as an SPKI certificate repository. Together with applications that hide the unnecessary complexity involved in accessing the DNS, it should be possible to make the DNS practical enough to use as a repository even for large amounts of frequently changing certificate data.

Telecommunications Software Security Architecture (TeSSA) is an architecture for secure distributed computing. The resolver created within the course of this work is soon to be integrated with other components, in order to form a TeSSA implementation. Issues of usability will be studied further within the TeSSA project. The usability research will not be limited to the certificate repository, as it is important to make the entire architecture easy to use. This is because the security and reliability of the architecture implementation depend not only on the availability of certificates, but also on what authorizations the certificates grant and to whom. Maintaining a security policy is a challenging task for the administrator. There remains more research to be done in investigating how to help automate the process of keeping the certificate data up-to-date and how to visualize the maintained data.

Bibliography

[1] Tuomas Aura. Comparison of graph-search algorithms for authorization verification in delegation networks. In *Proceedings of 2nd Nordic Workshop on Secure Computer Systems NORDSEC'97*, Espoo, Finland, November 1997.

[2] Tuomas Aura. Fast access control decisions from delegation certificate databases. In *Proceedings of 3rd Australasian Conference on Information Security and Privacy ACISP '98*, volume 1438 of *LNCS*, pages 284–295, Brisbane, Australia, July 1998. Springer Verlag.

[3] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform Resource Identifiers (URI): Generic syntax. *Request for Comments: 2396*, August 1998.

[4] Tim Berners-Lee, Larry Masinter, and Mark McCahill. Uniform Resource Locators (URL). *Request for Comments: 1738*, December 1994.

[5] Marc Branchaud. A survey of public key infrastructures. Master's thesis, McGill University, March 1997.

[6] Matthew Condell, Charles Lynn, and John Zao. Security Policy Specification Language. Internet draft, Network Working Group, October 1998.

[7] David H. Crocker. Standard for the format of ARPA Internet text messages. *Request for Comments: 822*, August 1982.

[8] Donald E. Eastlake. Bigger Domain Name System UDP replies. Internet draft (expired), June 1998.

[9] Donald E. Eastlake. Secure Domain Name System (DNS) dynamic update. Internet draft, August 1998.

[10] Donald E. Eastlake. Secure Domain Name System dynamic update. *Request for Comments: 2137*, April 1997.

[11] Donald E. Eastlake. Domain Name System security extensions. *Request for Comments: 2535*, March 1999.

[12] Donald E. Eastlake and Olafur Gudmundsson. Storing certificates in the Domain Name System (DNS). *Request for Comments: 2538*, March 1999.

[13] Carl M. Ellison, Bill Franz, Butler Lampson, Ronald L. Rivest, Brian M. Thomas, and Tatu Ylönen. SPKI certificate theory. Internet draft, IETF SPKI Working Group, November 1998.

[14] Carl M. Ellison, Bill Franz, Butler Lampson, Ronald L. Rivest, Brian M. Thomas, and Tatu Ylönen. Simple public key certificate. Internet draft (expired), IETF SPKI Working Group, March 1998.

[15] Ned Freed and Nathaniel S. Borenstein. Multipurpose Internet Mail Extensions (MIME) part one: Format of internet message bodies. *Request for Comments: 2045*, November 1996.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, 1995.

[17] Yki Kortesniemi. Implementing elliptic curve cryptosystems in Java 1.2. In *Proceedings of NORDSEC'98 The Third Nordic Workshop on Secure IT Systems*, Trondheim, Norway, November 1998.

[18] Ilari Lehti and Pekka Nikander. Certifying trust. In *Proceedings of the Practice and Theory in Public Key Cryptography (PKC) '98*, Yokohama, Japan, February 1998. Springer-Verlag.

[19] Mark K. Lottor. Domain administrators operations guide. *Request for Comments: 1033*, November 1987.

[20] Douglas Maughan, Mark Schertler, Mark Schneider, and Jeff Turner. Internet Security Association and Key Management Protocol (ISAKMP). *Request for Comments: 2408*, November 1998.

[21] Paul Mockapetris. Domain names - concepts and facilities. *Request for Comments: 1034*, November 1987.

[22] Paul Mockapetris. Domain names - implementation and specification. *Request for Comments: 1035*, November 1987.

[23] Jeffrey Mogul and Steve Deering. Path MTU discovery. *Request for Comments: 1191*, November 1990.

[24] Pekka Nikander and Arto Karila. A Java Beans component architecture for cryptographic protocols. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, January 1998. Usenix Association.

[25] Pekka Nikander and Lea Viljanen. Storing and retrieving Internet certificates. In *Proceedings of NORDSEC'98 The Third Nordic Workshop on Secure IT Systems*, Trondheim, Norway, November 1998.

[26] Masataka Ohta. Incremental zone transfer in DNS. *Request for Comments: 1995*, August 1996.

[27] Jonna Partanen and Pekka Nikander. Adding SPKI certificates to JDK 1.2. In *Proceedings of NORDSEC'98 The Third Nordic Workshop on Secure IT Systems*, Trondheim, Norway, November 1998.

[28] Jon Postel. User Datagram Protocol. *Request for Comments: 768*, August 1980.

[29] Jon Postel (editor). Transmission Control Protocol. *Request for Comments: 793*, September 1981.

[30] Ronald L. Rivest. S-expressions. Internet draft (expired), IETF Network Working Group, May 1997.

[31] Sanna Suoranta. An object-oriented implementation of an authentication protocol. Master's thesis, Helsinki University of Technology, November 1998.

[32] Paul Vixie. A mechanism for prompt notification of zone changes (DNS NOTIFY). *Request for Comments: 1996*, August 1996.

[33] Paul Vixie, Susan Thomson, Yakov Rekhter, and Jim Bound. Dynamic updates in the Domain Name System (DNS UPDATE). *Request for Comments: 2136*, April 1997.

[34] Brian Wellington. Simple secure Domain Name System (DNS) dynamic update. Internet draft, DNSSEC Working Group, February 1999.

Appendix A

Terms and Abbreviations

BIND Berkeley Internet Name Domain.

JAR Java Archive.

JVM Java Virtual Machine.

Maximum Transfer Unit The size of the largest piece of data that can be transferred across a given physical network without fragmentation.

octet Eight bits.

PGP Pretty Good Privacy.

primary name A domain name which is not an alias, i.e. there is no `CNAME` record that has the name.

thin client A simple client program or device which relies on the servers it interacts with to provide most of the functionality required for performing tasks.

X.509 A de facto standard created by ITU (International Telecommunication Union) that defines data formats and procedures related to the distribution of public keys via certificates.