Implementing an SPKI Certificate Repository within the DNS *

Tero Hasu and Yki Kortesniemi

Helsinki University of Technology,
Department of Computer Science,
FIN-02015 HUT, Espoo, Finland
{tero.hasu, yki.kortesniemi}@hut.fi

Abstract. Authorisation certificates can be used to grant access rights from the owner of a resource to other entities and then to further share these rights with others using delegation. However, when access decisions are made, the delegated rights will not be acknowledged unless all the certificates in the delegation chain are available for verification. In this paper we discuss some options for having the necessary certificates available when needed, talk about a proposed solution of storing part of the chain in the DNS, describe our implementation of a DNS based SPKI certificate repository and, finally, elaborate on its use.

1 Introduction

Interactions between entities typically rely on trust, which makes it useful for an entity to be able to prove that it is trusted by another entity. For example, one may be able to access a system if and only if one is able to prove that one is trusted by the owner of the system. To allow the existence of such a trust relationship to be proved, the system's policy administrator can issue a digital certificate stating that a particular entity is authorised for access.

If an access control solution is based on the use of certificates, it is essential for the correct certificates to be available when access rights need to be proved. If only a single entity needs a particular certificate, it may be enough for that entity to store the certificate by itself, provided it has sufficient storage space. However, if numerous entities need the same certificate, storing it in a public, distributed database may be a more appropriate solution. As the signature in a certificate makes it possible to verify both the authenticity and the integrity of the certificate, it is reasonably safe to trust the information contained in a certificate, even if the certificate is transmitted and stored using an untrusted network.

One distributed database solution suitable for certificate storage is the Domain Name System (DNS). It has the benefit of already being widely used – although for other purposes; it could be quickly adapted to storing certificates

* This work was funded by the TeSSA research project at Helsinki University of Technology under a grant from Tekes.

as well. The idea of using the DNS for storing SPKI (Simple Public Key Infrastructure) certificates was already outlined in [22], and more details were provided in [12]. This paper describes how an SPKI certificate repository can be implemented using the DNS.

The TeSSA (Telecommunications Software Security Architecture) project aims at presenting a complete solution for securing communications in a non-trusted internet-like network environment. As a part of realising this goal, we have extended the Java 2 security model to allow distributed management of information regarding code execution rights. This was accomplished by replacing the identity-based security management with a capability-based alternative in which policy decisions are based on authorisation certificates instead of a local configuration file for each JVM. The certificates may be acquired on demand from various sources. Our implementation uses SPKI certificates and three repositories: JARs, local databases, and the DNS. This paper also explains how the repository is being utilised by the current TeSSA architecture implementation.

The rest of this paper is organised as follows. Section 2 presents background information by shortly discussing trust and digital certificates in general. Sections 3 and 4 continue by covering SPKI and the Domain Name System, respectively. Section 5 explains the motivation behind our distributed SPKI repository implementation, and also gives an idea of what it is currently being used for. Section 6 goes on to describe the repository itself, and Section 7 suggests future work related to the repository. Section 8 presents a conclusion of our work.

2 Expressing Trust

As Lehti and Nikander state in [14], *trust* is a belief that an entity behaves in a certain way. Trust is rarely absolute, and it is important for trust to have attributes that make it clear, what kind of trust is in question. For example, one can say: "I trust Alice to complete this task." The source of the trust, the target of the trust, and the kind of trust together form a *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 in the sense that one's trust cannot be betrayed if one does not trust anyone. However, in practice such a model is difficult and often impossible to use. In many cases there is no way to get the task done without trusting at least some external entities. As an example, let us consider mobile code. 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, thus protecting the resources from attacks. The downside is that 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.

In the real world, not only our trust model, but other factors, like laws and company practices, affect our decisions. A *security policy* is the set of rules

and practices that regulate how sensitive information and other resources are managed [20]. It is desirable for the security policy to reflect the trust model, but sometimes this is not possible as there may be conflicts with the other factors. Also, in order to automatically enforce the policy, we must be able to state it explicitly.

A digital *certificate* is a signed statement, in which the signer's belief about the properties of some entity is expressed. The belief may not be justified, but that does not make the certificate invalid. Sometimes the signer may not even believe the statement, but despite that he can express belief in it. It should be noted that certificates are closely related to trust relationships; they can be regarded as representations of trust relationships.

It is assumed that the cryptographic methods used to create the signature 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 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.

3 Simple Public Key Infrastructure

A *public key infrastructure* (PKI) is a system that provides a mechanism for publishing public-key values which are bound to some other piece(s) of information, such as a name or an authorisation. To support the interoperability of applications, PKIs define certificate formats and semantics, as well as the process of verifying that a certificate is valid.

The Internet Engineering Task Force (IETF) is developing a PKI proposal called Simple Public Key Infrastructure (SPKI) [11, 10]. SPKI is more flexible than X.509 [4] and free from the requirement of a global, trusted Certification Authority (CA) hierarchy. Any entity having a private key may issue certificates. SPKI has adopted ideas from the SDSI [26, 25] and PolicyMaker [3] prototype systems. [13]

SPKI was designed to support certificate based authorisation. Just about any a trust relationship can be described using SPKI certificates; thus policy rules can be expressed and permissions can be granted in the form of certificates. Now, authorisation decisions can be supported by or be completely based on a set of SPKI certificates. However, the SPKI specification does not provide a list of authorisations that can be included in a certificate. The decision of which authorisations to support in an application is left to the developer, which adds to the simplicity and generality of SPKI, but necessitates further standardisation if interoperability of applications is desired.

SPKI certificates can be used to certify identity, as well, but unlike X.509 and other name oriented systems, SPKI uses cryptographic keys to represent identities. A public key can act as an identifier for the set of all entities which can prove possession of the corresponding private key. To facilitate certificate

management by humans, SPKI provides for local name spaces that are relative to their creators. These local spaces can then be linked to create bigger, even global, name spaces, if necessary. There are no global names which would refer to the same keys in every name space. SPKI does not attempt to enforce that there be no more than one key per name. Names can therefore be thought to refer to groups of entities.

SPKI certificates can be divided into the three categories given below.

authorisation certificate Binds an authorisation to a key.
name certificate Binds a name to a key.
attribute certificate Binds an authorisation to a name.

An authorisation or attribute certificate can be abstracted into a signed quintuple (I, S, D, A, V) where

I is the Issuer's (signer's) public key, or a secure hash of the public key,
S is the Subject of the certificate, typically a public key, a secure hash of a public key, a name, or a secure hash of some object,
D is a Delegation flag, which, when true, indicates that Subject may further delegate Authorisation or any subset of it,
A is the Authorisation field, describing what rights the Issuer delegates to the Subject,
V is a Validation field, describing the conditions (such as a time range) under which the certificate can be considered valid.

The above abstraction contains all the fields relevant to making access control decisions. The other fields of SPKI authorisation certificates that are of relevance to this paper are `issuer-info` and `subject-info`. These two fields may contain any number of URIs (Universal Resource Indicators) that refer to information regarding the issuer or the subject, respectively.

Often, it is not practical for the administrator of the resource to issue a certificate to every user of the resource. Instead, delegation can be used to form chains of certificates, which grant the rights to new users. An SPKI *certificate chain* 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 $\forall c_k = (i_k, s_k, d_k, a_k, v_k)$, $1 \leq k \leq n-1$, $d_k = true$. C can be used to prove that s_n has been given authorisation $a_1 \cap a_2 \cap \ldots \cap a_n$ by i_1 for the validity period $v_1 \cap v_2 \cap \ldots \cap v_n$.

A name certificate can be abstracted into a signed quadruple (I, S, N, V) where I, S and V are as above, and N is a name for S in I's name space. The name can be any octet string.

Using keys instead of names as identifiers has important privacy consequences. A key without any additional information, like a name certificate, can not be linked to any particular entity and can therefore be regarded as an anonymous identity. The anonymity can be lost, however, if the connection between the entity and the key becomes apparent as a result of careless use of the certificate or bad choice of storage location, as we shall see later.

3.1 Certificate Encoding

SPKI certificates are represented using S-expressions. 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 octets. The lists have zero or more elements, which can be either strings or lists.

While 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.

In order for a secure hash 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 [24].

This S-expression, encoded in a relatively reader-friendly manner,

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

has the following canonical encoding (printed assuming the ISO-Latin1 character set):

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

4 The Domain Name System

The Domain Name System (DNS) [15–17] is the member of the TCP/IP protocol suite that is responsible for translating host names to Internet Protocol (IP) addresses. Other uses are also possible, however, as the DNS is simply 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. Naturally, the design of the DNS makes it more suitable for applications similar to host name lookup and less suitable for data of different characteristics, as we shall see.

All the data stored in a DNS database is composed of so-called *resource records* (RRs). Each record is indexed by its *domain name*. Domain names are not unique identifiers, as more than one record can share the same name. The set of all different values by which data is indexed into a DNS database forms a *domain name space*. A domain name space is always structured like a tree, due to the way the DNS indexing structure has been designed. Domain names must thus be chosen so that they indicate a position in a hierarchical structure. For more discussion about the naming limitations, see Section 4.2.

The RRs stored in a DNS database are divided into *zones*, which in turn are distributed between servers that are responsible for answering queries regarding the data stored in the database. Such a server is called a *name server*, and each zone should have at least two servers that are considered to be authoritative for the zone. The information regarding the records in a zone is typically stored in a file, which is accessible to the servers having the authority over the zone. In this

paper, such a file is referred to as a *zone file*. Zone files are text files, and zone information may thus be updated just by editing all the relevant zone files with a text editor. Other update methods exist, of which most are just proposals [23, 27, 28, 7, 6, 29], and one has been standardised [16].

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. To improve performance, resolvers usually have a cache in which RRs may be stored. The time limit for caching an RR is specified in the header of the RR itself. Data returned by an authoritative server should be preferred over non-authoritative data when deciding, what data to keep in the cache.

The DNS, in its current form, is not particularly secure. 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 (DNSSEC) [8]. DNSSEC adds support for data integrity checking and authentication, among other things. Implementations have already started to appear.

4.1 DNS Message Size

Resource records are carried from one name server to another within DNS messages. These messages can be transmitted either as User Datagram Protocol (UDP) datagrams or in a byte stream formed with Transmission Control Protocol (TCP). UDP packets are preferable, as they offer lower overhead and better performance. However, the use of UDP imposes limitations to the size of messages that can be sent. The DNS specification defines the maximum size of messages transported via UDP 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 must be truncated and clearly marked as having been truncated. The resolver should then resubmit the query using 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 [5, 18], 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 [5].

4.2 Domain Naming Limitations

The index of a DNS database, being structured like a tree, can be thought to consist of nodes and arcs. We use the term *node* to refer to both internal nodes and leaf nodes. Each node has what is referred to as a *label*. Labels are byte strings with the maximum length of 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 is lost 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.

A domain name (which can be regarded as the name of a subtree of the domain name space) 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. The DNS specification defines how domain names should be represented as text: dots are used as separators for the labels in the text representation and 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. The total number of octets that represent a domain name is limited to 255 [16].

4.3 Certificate Support in the DNS

All resource records have a type, and records of different types may have a different format. A proposal for an RR for storing certificates is described in [9]. Each RR type is allocated a number and a mnemonic; for the certificate RR type these are 37 and `CERT`, respectively. The structure of a `CERT` RR is shown in Figure 1. All RRs share the same header format. The fields specific to `CERT` RRs are described below.

certificate type `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.

key tag The current specification states that the key tag is a 16-bit value which should be computed from a public 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 [8]. Before calculating the value, the key needs to be translated into the same format as it would have within a `KEY` RR. This, of course, requires the key type to have been defined in DNSSEC – otherwise the translation cannot be done. In that case the tag should be set to zero. If the tag is not

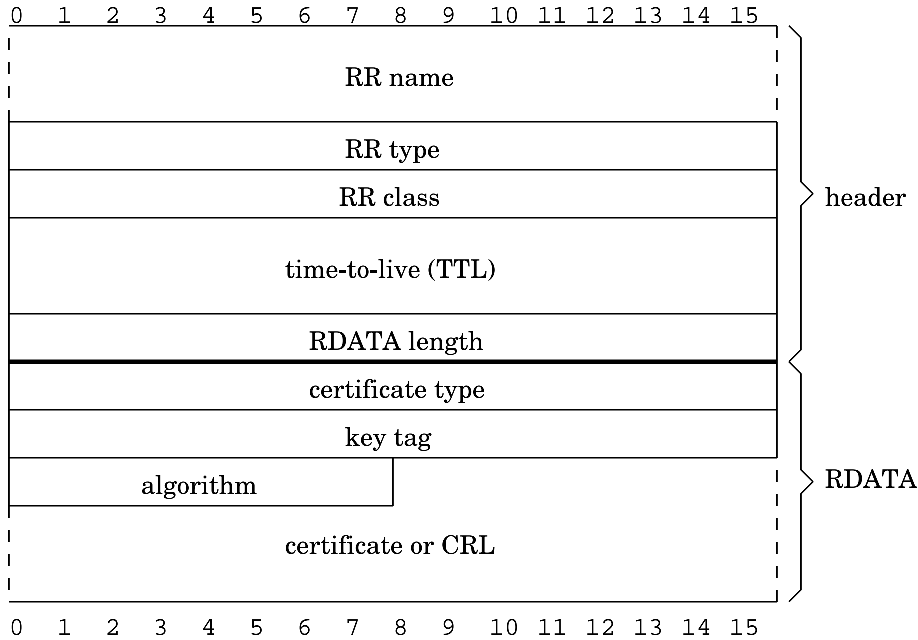

Fig. 1. `CERT` resource record structure.

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.

algorithm 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 (as defined in [8]), 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.

certificate or CRL The certificate itself. Alternatively, instead of a certificate, the record may contain a certificate revocation list (CRL). The `CERT` 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 4.1, space is at premium within DNS messages.

As zone files are typically text files, the `CERT` specification also defines a text representation for the `CERT` RDATA. In a zone file a `CERT` RR could look like:

```
    s.cf9e0250.certs.to. CERT 42 55742 ECC (
) KDQ6Y2VydCg2Omlzc3Vlcjk6c2VydmVyLXBhKSgxMDppc3N1ZXItbG9jMjM6ZG5zOkBjY (
W1waG9yLnRjbS5odXQuZmkpKDc6c3ViamVjdDk6Y2xpZW50LXBhKSgxMTpzdWJqZWN0LWxvYz
IzOmRuczpAY2FtcGhvci50Y2OuaHV0LmZpKSgzOnRhZzE6MSkoODpkbnMtdHlwZTE6MykoNjp
) zaWduZXI5OnNlcnZlci1wYSkp
```

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 Implementing a Repository

To better understand the choices made in the implementation of the repository, let us start by investigating the intended uses for the repository. In particular, the use of certificates for Java 2 policy management is talked about. That discussion is followed by an introduction to the search algorithm used, and an example of the algorithm in action.

5.1 Motivation

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 entity might not need the information between every update.

If, instead, all entities 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".

The certificate storage solution must scale well if it is to support an Internet-wide system that is open for everyone to participate in. Centralised solutions would probably prove inadequate, at least because of congestion caused by all entities in need of a certificate trying to access the same network node. Distributed databases do not have this problem; more servers can be added if the amount of data stored in the database gets too large for the existing servers to handle. Distribution 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.

5.2 Java Policy Management

In Java 2, a security policy defines the rules and the `AccessController` class enforces them. The security policy is implemented as a subclass of the class `Policy`. The default implementation uses a set of configuration files to determine the permissions that should be given to code modules; the permissions may depend on the location the code was loaded from, as well as the keys with which the module has been signed. Using this implementation, the permissions assigned to a class are pseudostatic; they are not amended unless the `Policy.refresh()` method is explicitly called. As described in [21], we have altered the default implementation so that, instead of a configuration file, the permissions are determined dynamically from available SPKI certificates. In addition to a local file, the certificates may also be stored with the code in a JAR file, or in the DNS.

In our implementation, calls to `AccessController.checkPermission(Permission)` will result in a call to our policy manager's `checkPermission(Permission,CodeSource)` method, which, if necessary, will attempt to reduce the set of available certificates to form a valid chain from its own key, called the "Self key", to the hash of the code module. It will only look for chains that prove the `Permission` passed to `checkPermission`.

5.3 The Search Algorithm

This section describes the graph search algorithm that is currently responsible for constructing certificate chains in the TeSSA architecture implementation. Note that fetching certificates from the DNS is only a part of the algorithm, and it is something that is only done as a last resort, after an exhaustive search through the local repositories has been found not to yield the required result. This is because access to local memory or disk is assumed to be much faster than access to the DNS.

Suppose that entity k_v needs to determine if software agent k_s should be allowed to perform operation o_a. k_v has a policy that states that an entity is allowed to perform o_a iff it has authority a. k_v will see if it can form a certificate chain C_a such that C_a belongs to the set of all certificates that prove a and are accessible to k_v. k_s was provided within a JAR (Java Archive) file, which also contains the certificates C_{JAR}. k_v also has access to a local certificate repository R_{local}, as well as to a DNS certificate repository R_{DNS}.

Below is an algorithm that finds C_a if it exists. Note that at any point of the search, no certificate $c_c = (i_c, s_c, d_c, a_c, v_c)$ is to be added to the search tree unless its signature is valid, and $a_c \supseteq a$, and v_c indicates that c_c is valid at the time of the search. Also, non-delegable certificates are only to be considered if their subject is k_s.

JAR search phase

Beginning from k_s, build a backward search tree T_{bs} using depth-first search. That is, first take those certificates in C_{JAR} that have been issued to k_s and attach them as arcs to T_{bs}. Then recursively go through all of the attached arcs

and attach more certificates to T_{bs} if possible. If at any point T_{bs} reaches k_v, terminate the search immediately and return the chain C_a that consists of the arcs leading from k_v to k_s.

Local repository search phase

Take the nodes in T_{bs} as the target set, and build a forward search tree T_{fs} using depth-first search. Let $R_{local}(k)$ be the set of all certificates in the local repository that have been issued by k. First, attach $R_{local}(k_v)$ to T_{fs}, and then recursively go through all of the attached arcs, attaching more arcs to the subject nodes if possible. If at any point T_{fs} reaches T_{bs}, immediately combine the trees and return any path from k_v to k_s in the resulting tree as a certificate chain.

DNS search phase

The DNS search phase has two subphases: a forward search phase and a backward search phase. Almost the same search algorithm is described in [12] with more detail; the only difference is that here we make use of the search trees T_{fs} and T_{bs} that may already contain arcs.

1. *Forward search phase*
   For each entity k in T_{fs} for which DNS location information is known (contained within some of the certificates) do:
   (a) Set the forward search set C_{fs} to $R_{DNS}(k,\text{any})$ (the set of certificates stored within the DNS that have k as the issuer, and any subject).
   (b) If $C_{fs} = \varnothing$, return indicating failure.
   (c) Add C_{fs} to T_{fs}.
   (d) If T_{fs} now reaches T_{bs}, combine the trees, and return any path from k_v to k_s in the resulting tree as a certificate chain, indicating success.
   (e) If C_{fs} only contains one certificate, recursively continue the forward search phase from the subject of that certificate. Return the result.
   (f) Move on to the backward search phase.
2. *Backward search phase*
   For each entity k in T_{bs} for which DNS location information is known do:
   (a) Set backward search set C_{bs} to $R_{DNS}(\text{any}, k)$.
   (b) If $C_{bs} = \varnothing$, return indicating failure.
   (c) Add C_{bs} to T_{bs}.
   (d) If T_{bs} now reaches T_{fs}, combine the trees, and return any path from k_v to k_s in the resulting tree as a certificate chain, indicating success.
   (e) Recursively continue the backward search phase for each certificate c in C_{bs}, choosing the issuer of c as k. If success is returned, stop the search and return the result chain, indicating success.
   (f) Return indicating failure.

5.4 An Example

Figure 2 illustrates an example graph search scenario, in which the search algorithm presented in Section 5.3 is used to attempt to acquire a certificate chain whose elements meet certain criteria C. For example, one criterion could be that

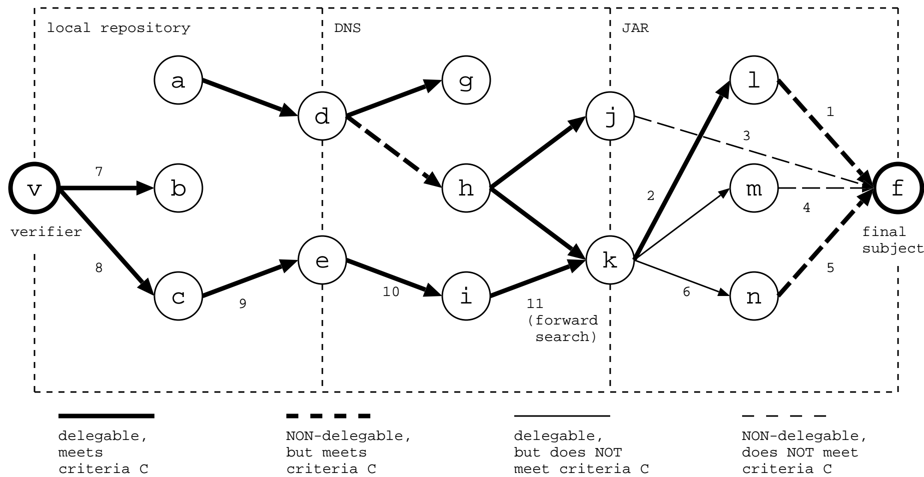

Fig. 2. A graph search scenario.

the certificates in the chain must prove that the subject of the certificate has permission p, and another one could be that the certificates must presently be valid.

The arcs of the graph represent certificates and the nodes represent entities. The verifier's public key is v, and the hash of the Java code whose permissions are to be determined is f.

The numbers show the order in which the certificates are examined by the algorithm. Here we are assuming that the certificates are checked in top to bottom order, but the order is implementation dependent. The certificate chain that is acquired by the algorithm in this case is $v \rightarrow c \rightarrow e \rightarrow i \rightarrow k \rightarrow l \rightarrow f$, which is also the only one in the graph that satisfies the required criteria C.

6 The Certificate Repository Implementation

The repository implementation consists of two parts: the name server and the resolver. In our implementation, the name server is a standard name server, but the resolver was tailor-made for this application. This section describes the implementation in detail and starts by talking about the DNS as a database, moves on to briefly discuss the name server, then details our resolver and finally evaluates the implementation.

6.1 The Database

One of the principles of the TeSSA project is to use existing open solutions whenever a suitable one can be found. The Domain Name System is a standardised distributed database that is already in use throughout the Internet, and it can

be used for storing certificates. Mostly for these reasons, the DNS was chosen as the basis of our distributed certificate repository implementation.

Indexing Storing the certificates in `CERT` RRs was an obvious decision, as the particular RR type has the purpose of allowing certificates to be stored in the DNS. However, choosing an index structure for the certificates to be stored was not so straightforward. The standard DNS indexing practice naturally had to be adhered to, but the two-way graph search algorithm that we wanted to use for certificate chaining also had its own requirements.

Let us assume that we have been given sufficient information, e.g. public keys and domain names, about any two entities k_1 and k_2. Then it must be possible for the algorithm to find all certificate chains between k_1 and k_2, if all the certificates in those chains are stored in the DNS, and the algorithm is not terminated as soon as it finds the first chain (which it might do, for performance reasons).

As the graph search algorithm is a two-way one, it needs to be able to fetch certificates issued by, or issued to an entity, depending on the direction of the search. The sets of certificates returned do not need to be exhaustive, for as long as that will not cause the algorithm to miss any chains. Due to the nature of the algorithm, those certificates that may appear in the middle of any chain must be indexed by both the issuer and the subject. Those that can only appear in the beginning must be indexed by the issuer, and those that can only appear in the end must be indexed by the subject.

As DNS entries are indexed by domain names, it follows from the above reasoning that certificates which may be in the middle of a chain must be stored in two RRs, one of which should be given the issuer's domain name, and the other one that of the subject. It is possible to use RRs of type `CNAME` to avoid placing the same certificate data under two different domain names. A `CNAME` RR specifies an alias; e.g. requests for `CERT` RRs in domain `i.selfadmin.certs.to` could be "redirected" to result in replies consisting of the `CERT` RRs with the name `s.gameprovider.certs.to` by adding a single `CNAME` RR in the `i.selfadmin.certs.to` domain.

To easily distinguish between certificates that require different kind of indexing or handling by the search algorithm, we are using the following terminology, which was originally defined in [22] and revised in [12].

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 authorised 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.

Certificate category	Domain name of issuer	subject
Trust	×	
Delegation	×	×
Permission		×
Identity		×

Table 1. Indexing of `SPKI CERT` RRs.

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).

Note that identity certificates are currently not handled by the algorithm at all. The indexing of certificates stored in the DNS is summarised in Table 1.

Setting `issuer-info` and `subject-info` If we encounter a previously unknown entity while constructing a certificate chain, we do not know which domain name it uses to store its certificates. This information must be available from somewhere. The entity identifiers themselves are contained in the `issuer` and `subject` fields of SPKI authorisation and attribute certificates. As that information is contained in the certificates themselves, it is natural to also store the other necessary information in the same place.

We have used the two fields, `issuer-info` and `subject-info`, 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 rather than to create two different certificates.

As stated in Section 3, the information contained in the `issuer-info` and `subject-info` fields must be in the form of Universal Resource Indicators (URIs) [2]. An URI contains a prefix specifying the naming scheme used, but at the time of writing no prefix has been assigned for DNS domain names. For testing purposes, we have been using an URI of the form `dns:`*dns_domain_name*, where *dns_domain_name* is any valid domain name of any DNS domain name space.

As mentioned earlier, a public key alone, without any other information regarding the key, cannot be linked to any particular entity. As the `issuer-info`

and `subject-info` have the purpose of providing additional information about the public keys (or other entity identifiers) in a certificate, they may potentially compromise anonymity. The domain name used by an entity for certificate storage can be very revealing. Therefore, if an entity 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.

A Naming Convention All certificates that have the same domain name and the same RR type and class (RR class specifies the protocol suite that the information in the RR applies to) will be returned at once by a name server. This may result in large replies unless the domain names are chosen carefully. The search algorithm that we are using only wants to fetch certificates either issued by an entity or issued to an entity, but not both at once. This is something that can be taken advantage of by assigning two domains per entity, one for certificates issued by the entity, and one for certificates issued to the entity.

We have adopted a naming convention in which each entity has its own domain name for storing certificates, and that name is prefixed with the label `i` or `s` when naming certificates issued by or issued to the entity, respectively. While this kind of naming will avoid unnecessary traffic and data processing caused by irrelevant data in replies, it will unfortunately not help avoid the use of TCP replies, unless very short keys are used in the certificates. The size limitations are too severe, as described in Section 4.1.

Setting Key Tags As explained in Section 4.3, `CERT` RRs contain a key tag, but just one. We have used the following rule to decide whether the key tag should be calculated from the issuer field or the subject field of an SPKI certificate:

If an SPKI certificate is to be stored in its issuer's domain, the key tag is 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 is set to zero. If the field contains a hash of a key, the key is acquired for the calculation if possible.

6.2 The Name Server

As our repository implementation is based on the DNS, we naturally needed a name server as one of the components. We have not implemented our own name server, however, nor do we have any intention to do so. There is little need for tight integration between our architecture implementation and a name server, as the resolver alone allows us to request services from any name server that correctly implements the DNS protocol.

Currently our resolver implementation is limited to only making queries, and we have been making zone updates manually at the server end. However, DNS dynamic update standardisation is progressing, and dynamic update support could be added to the resolver implementation if there were sufficient need for

the ability to make flexible zone updates. That kind of functionality could, for example, be needed by a trust and policy management interface which would allow one to interactively update trust information kept in various different repositories.

The `CERT` RR type is a fairly new proposal, and is not yet widely supported by name servers. BIND (Berkeley Internet Name Domain) is currently the most widely used name server, and its source code is freely available. We were prepared to take BIND 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 by the time our resolver implementation was ready 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. Other than that, BIND 8.2 has worked well for our purposes.

6.3 The Resolver

The DNS resolver that we are using was tailor-made to suit the requirements of the current TeSSA architecture implementation. It supports the most common RR types as well as `CERT` RRs, and facilitates convenient handling of SPKI certificates by providing a class interface with explicit SPKI support. `CERT` RRs of types other than `SPKI` may be filtered out from replies. A key tag may also be specified; when this is done, the answer returned by the resolver will be limited to `CERT` RRs that either have the provided key tag value, or for which the key tag has not been set at all. The filtering process is fast, because the certificates themselves need not be examined at all.

The resolver provides recursive service, i.e. is capable of following hints returned by name servers until it finds the name server that is authoritative for a particular domain. RRs received from name servers are automatically cached, and those from authoritative sources are preferred over non-authoritative ones. Sometimes caching may result in a client receiving expired certificates, when in fact there would be fresh, valid certificates in the authoritative name servers. For this reason, the resolver interface allows one to optionally demand the resolver to only return data that came directly from an authoritative source. If expired certificates are received, the query should be remade using this option.

For uniformity with the existing system of TeSSA, the resolver was written in Java. The JaCoB framework [19] was utilised in implementing the protocol, as has been done with other protocol components used in TeSSA. The resolver needs to be hooked into a UDP protocol component before it can communicate with name servers. We have two interchangeable components suitable for the task, one of which has been built using the JaCoB framework; the other one is based on the functionality provided by the `java.net` package. TCP support has been added to the resolver recently, and it was also implemented using `java.net`.

6.4 Evaluation of the Repository

Based on our experiences with the certificate repository described in Section 6, we believe the DNS to be suitable for storing SPKI certificates. Use of the DNS should enable one to relatively quickly put together a distributed database from which SPKI certificates can be retrieved on demand, especially once `CERT` RRs are fully supported by common DNS software. Choosing domain names and the method of making updates to the database seem to be the biggest questions that need to be considered when creating a certificate database within the DNS.

The indexing scheme described in Section 6.1 works well enough to satisfy the needs of our search algorithm, but we would have preferred to index certificate data by the canonical S-expression form of both the issuer and subject of each certificate. Because of the insufficiently flexible hierarchical indexing used in the DNS, we could not think of a way to do so. Strict rules have to be adhered to when naming domains (see Section 4.2), perhaps the most severe of which is that the hierarchical structure has to be preserved. Not just anyone can put data in any domain, and thus it appears impossible to somehow derive an appropriate domain name from a public key, for example, if one has no knowledge of to whom the key belongs.

Better support for making updates is a lesser problem, as it is easier to add to the DNS without breaking backward compatibility. As explained in Section 4, there already are several new proposals for making DNS database updates. It is to be expected that implementations will offer dynamic update support in the near future, if they do not do so already.

Our resolver implementation could also use update support, but other than that it provides all the functionality that we have had use for so far. We have experienced no performance problems when accessing the DNS with the resolver; it takes roughly a second to consult one name server. In some cases the required information can be found from the cache of the resolver, in which case an answer is returned almost instantaneously. Considering that speed optimisation has not had the main emphasis in the current architecture implementation, DNS access is not a bottleneck in its performance.

7 Future Work

In addition to creating new applications utilising the certificate repository described in this paper, further work can be put into improving our implementation and researching alternative repository solutions. Adding dynamic update support to the resolver and creating an administrative interface to facilitate easier management of the certificates in a repository have already been mentioned. A related question is whether SPKI certificates could be used to control zone updates, and whether the ability to do so would offer some clear benefits over those authentication and authorisation mechanisms suggested in the current dynamic update specifications, e.g. the use of `KEY` and `SIG` RRs.

Both the performance and the proof-finding capabilities of the graph search algorithm described in this paper could be improved. In the DNS search phase,

the current implementation 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 [12]. Exhaustive searches can be very time-consuming, and [22] suggests heuristics to help avoid them without significantly reducing the algorithm's success rate. The algorithm is not capable of findind proof for rights that require more than one certificate chain to prove, nor can it correctly handle certificates with threshold subjects (which specify multiple subjects of which a specified number must co-operate to exercise a right). Aura has presented a graph search algorithm that does support threshold certificates [1].

As mentioned in Section 5.2, our Java policy manager implementation determines permissions in a fully dynamic manner. While this allows for flexible policy management, performance would probably benefit from a less dynamic solution. Because chains may contain certificates which must be validated before each use, and new relevant certificates may appear in repositories at any time, a fully static solution is inadequate. However, at least `Permissions` granted by chains consisting only of certificates without validity limitations could be cached for future use; future queries regarding these permissions could be answered without referring to any certificate repository or constructing any chains. We intend to research these and other possibilities for optimisation in the future.

8 Conclusion

In this paper we have discussed, how many decisions are based on trust and how this trust can be expressed using authorisation certificates, like the SPKI certificates. Often, the rights contained in these certificates are further delegated thus forming chains. To facilitate the management of the certificate chains, it is a good idea to have a globally accessible storage for certificates belonging to more than one chain.

We have explored the benefits and drawbacks of using the DNS as a certificate repository and described a naming scheme and an algorithm for finding the certificates. We have also introduced and evaluated our implementation of a resolver and, finally, suggested ideas for future work.

References

1. 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.
2. Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform Resource Identifiers (URI): Generic syntax. *Request for Comments: 2396*, August 1998.
3. Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Computer Society Symposium on Research in Security and Privacy*, Oakland, California, May 1996. IEEECSP.

4. CCITT (Consultative Committee on International Telegraphy and Telephony). Recommendation X.509, the directory – authentication framework. *CCITT Blue Book*, 8:48–81, 1988.
5. Donald E. Eastlake. Bigger Domain Name System UDP replies. Internet draft (expired), June 1998.
6. Donald E. Eastlake. Secure Domain Name System (DNS) dynamic update. Internet draft (expired), August 1998.
7. Donald E. Eastlake. Secure Domain Name System dynamic update. *Request for Comments: 2137*, April 1997.
8. Donald E. Eastlake. Domain Name System security extensions. *Request for Comments: 2535*, March 1999.
9. Donald E. Eastlake and Olafur Gudmundsson. Storing certificates in the Domain Name System (DNS). *Request for Comments: 2538*, March 1999.
10. 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.
11. 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, May 1999.
12. Tero Hasu. Storage and retrieval of SPKI certificates using the DNS. Master's thesis, Helsinki University of Technology, April 1999.
13. Yki Kortesniemi, Tero Hasu, and Jonna Partanen. A revocation, validation and authentication protocol for SPKI based delegation systems. To appear in *Network and Distributed System Security Symposium*, San Diego, California, February 2000.
14. 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.
15. Mark K. Lottor. Domain administrators operations guide. *Request for Comments: 1033*, November 1987.
16. Paul Mockapetris. Domain names – concepts and facilities. *Request for Comments: 1034*, November 1987.
17. Paul Mockapetris. Domain names – implementation and specification. *Request for Comments: 1035*, November 1987.
18. Jeffrey Mogul and Steve Deering. Path MTU discovery. *Request for Comments: 1191*, November 1990.
19. 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.
20. Pekka Nikander, Yki Kortesniemi, and Jonna Partanen. Preserving privacy in distributed delegation with fast certificates. In *Proceedings of the Practice and Theory in Public Key Cryptography (PKC) '99*, Kamakura, Japan, March 1999.
21. Pekka Nikander and Jonna Partanen. Distributed policy management for Java 1.2. In *Proceedings of Network and Distributed System Security Symposium*, San Diego, California, February 1999.
22. 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.
23. Masataka Ohta. Incremental zone transfer in DNS. *Request for Comments: 1995*, August 1996.
24. Ronald L. Rivest. S-expressions. Internet draft (expired), IETF Network Working Group, May 1997.

25. Ronald L. Rivest and Butler Lampson. SDSI – A simple distributed security infrastructure. (See SDSI web page at `http://theory.lcs.mit.edu/~cis/sdsi.html`).
26. Ronald L. Rivest and Butler Lampson. SDSI – A simple distributed security infrastructure. In *Proceedings of the 1996 Usenix Security Symposium*, 1996.
27. Paul Vixie. A mechanism for prompt notification of zone changes (DNS NOTIFY). *Request for Comments: 1996*, August 1996.
28. Paul Vixie, Susan Thomson, Yakov Rekhter, and Jim Bound. Dynamic updates in the Domain Name System (DNS UPDATE). *Request for Comments: 2136*, April 1997.
29. Brian Wellington. Simple secure Domain Name System (DNS) dynamic update. Internet draft, DNSSEC Working Group, February 1999.