Certificate Distribution with Local Autonomy

Pankaj Kakkar, Michael McDougall, Carl A. Gunter
University of Pennsylvania*

Trevor Jim
AT&T Laboratories†

May 5, 2000

Abstract

Any security architecture for a wide area network system spanning multiple administrative domains will require support for policy delegation and certificate distribution across the network. Practical solutions will support local autonomy requirements of participating domains by allowing local policies to vary but imposing restrictions to ensure overall coherence of the system. This paper describes the design of a such a system to control access to experiments on the ABone active network testbed. This is done through a special-purpose language extending the Query Certificate Manager (QCM) system to include protocols for secure mirroring. Our approach allows significant local autonomy while ensuring global security of the system by integrating verification with retrieval. This enables transparent support for a variety of certificate distribution protocols. We analyze requirements of the ABONE application, describe the design of a security infrastructure for it, and discuss steps toward implementation, testing and deployment of the system.

Keywords: Security policy, certificate distribution, local autonomy, access control, ABone, active networks, QCM, Query Certificate Manager.

1 Introduction

Active network systems will require practical approaches for managing access control information securely and conveniently on a wide area network. The

*Point of Contact: Michael McDougall, mmcdouga@saul.cis.upenn.edu, PhD Student, Dept. of CIS, University of Pennsylvania, 200 South 33rd Street, Philadelphia PA, 19103. Phone: 215-898-0677 (day), 215-925-5621 (night), 215-898-0587 (FAX).

†This work was carried out while Trevor Jim was at the University of Pennsylvania.

technology for doing this will need to reach beyond the current state of the art for policy description techniques and certificate distribution. A good test case for understanding requirements and possible solutions for the problem is the management of the access control mechanism for experiments on the ABone [2], an emerging testbed for research in active networks. Requirements and solutions for the ABone also have relevance for many other wide area network systems beyond active networking.

The aim of this paper is to discuss the requirements and design of an access control infrastructure suited to wide area systems like the ABone. The principal focus is on the concept of *local autonomy,* wherein nodes within multiple administrative domains are allowed to define their own policies. This capability has two primary policy aspects: authorization policy and certificate distribution policy. There have been a number of recent proposals about how to express authorization policies in large-scale distributed systems [4, 16, 5, 3]. Work on certificate distribution has focused on the design of directory systems such as DNS/DNSSEC [19, 17, 1], the ISO Directory [13], and the Lightweight Directory Access Protocol (LDAP) [22, 12]. Such directories are used to hold certificates (digitally signed documents) providing information on which authorization policies are based. Other relevant work [14, 11] focuses on the formats of certificates and protocols like chaining of certificate authorities and revoking certificates. A key challenge for the use of such systems by wide area network applications is reconciling the demands of local autonomy with the functionality of the distributed system as a whole. Local autonomy is needed in the ABone and many other wide area distributed systems because different domains have needs and goals that may be in conflict. For scalability, the policy of one domain may have to rely on the policy of another domain. But a domain should be free to define its policy by taking what it wants from another domain's policy and discarding what is not appropriate. However, it is possible that such variations lead to a situation in which no domain is able to maintain the policy it requires, given its reliance on other domains with different policies.

Our own work on security infrastructure has focused on ideas for integrating verification and certificate retrieval using a technique called *policy-directed certificate retrieval* [7]. The basic idea is that the verifier is in the best position to determine what certificates are required, so it can be used effectively for the retrieval of certificates. Our implementation of this idea is a system called *Query Certificate Manager (QCM),* which enables a verifier to express policies for distributing certificates and retrieving them automatically as part of verification. The aim of QCM is to accomodate significant flexibility for both access and retrieval policies while ensuring consistent global security and tractable distributed computation. We therefore explore the idea of using policy-directed certificate retrieval as realized by QCM in maintaining the ABone access control infrastructure.

The ABone is under design currently (see [2] for a description of objectives and approach), so it requires both short and long term solutions. A short-term solution must support a modest number of participating nodes with an approach that can be implemented almost immediately with little impact on existing

```
alice.com        r8K+gZ4ZRo5usA675...
bob.com          udoN0w7B0K65hhwpw...
careless.org     umVy3uv1LpaSx7W83...
```

Figure 1: An ABone access file, with keys abbreviated.

software for the ABone, namely the various *Execution Environments (EE's)* for evaluating active packets and the *Active NETwork (ANET)* [20, 21] system for installing EE's. A long-term solution will need to provide support for an expanding collection of nodes and more complex access policies and distribution strategies.

We begin our analysis in the second section of this paper with a discussion of the simple authorization architecture initially used by ANET. We explain why this approach can be improved and present an architecture and language for our approach. In the third section we describe secure mirroring protocols, which can be used to provide a simple ACL managment system with modest local autonomy. In the fourth section we describe more advanced features based on policy-directed certificate retrieval as applied to the ABone and show how these features provide significant local autonomy. A final section summarizes conclusions.

2 ABONE Requirements and Proposed Infrastructure

The ABone is a collection of computers being used to run active networking experiments over the Internet. An active network system consists of one or more *Active Applications (AA's)* running on top of an EE that defines the semantics of code contained in active packets. ANET is the system for installing EE's on ABone nodes, which are mainly Unix hosts currently, but could be special-purpose active routers. ANET provides support for a server process called *ANETD*, the ANET Daemon, that responds to requests to run ABone experiments on the node where the daemon is running; ANET also provides a client that allows users to request and configure experiments on the ABone testbed by contacting ANETD servers. In the current version of ANETD, permission to carry out such an experiment is determined by the nodes on which the experiment will be run by consulting an Access Control List (ACL) consisting of public keys of principals permitted to perform experiments. In a typical scenario, a *claimant* client approaches a *verifier* server running ANETD with a certificate requesting access to the server for the purpose of conducting an experiment. Each user of the ABone generates their own 512-bit RSA key pair and registers the public key with a master server, currently operated at SRI. The master server at SRI maintains a master ACL listing the public key and host of every ABone user. An example is given in Figure 1. When a claimant

makes a request, it is signed with his private key, and the verifier checks this against public keys of permitted users. In the early versions of ANET, the first time ANETD ran on an ABone node, it queried the master server for the current ACL of permitted users. Once acquired, maintenance of this list was left to the administrator of the ABone node. Each local administrator was free to modify their copy of the list by adding or deleting users. This provided every ABone node autonomy over its own access policy.

This approach is sufficient for a few nodes if administrators are diligent about maintaining their ACLs, and the number of nodes is not changing much. However, it is hoped that the ABone will grow to more than a thousand nodes within a few years, and, for several reasons, this strategy will not scale. First, when a new user joins the ABone, their key is posted to the ACL of the master server, but there is no mechanism for propagating the new key to machines already running ANETD. If the user needs to run an experiment on these machines, their administrators will probably need to be contacted individually. Second, each site administrator maintains their copy of the list by hand. So, even if the hosts and keys of new users were distributed from the master server automatically, the administrators would have to process them by hand. Also, local policies are not written down anywhere; they just make their effects known in the local copy of the list. For example, if an administrator believes that `careless.org` has been infiltrated, he can delete every key associated with `careless.org` from his copy of the list, but there is no record to tell him not to put new keys from `careless.org` onto the list.

There are at least four basic strategies for dealing with these problems using certificates (signed documents) based on the following fundamental tradeoffs:

1. Whether the claimant or the verifier is responsible for obtaining certificates, and

2. Whether the certificates are long-term or short-term.

For instance, one idea is to allow the master server to issue certificates to permitted users asserting their right to do experiments. The opposing idea is to provide a means for ANETD servers to consult the master server about requests to set up experiments before or as the requests arrive. Both cases break down into significantly different solutions depending on whether certificates are long-term or short-term. Consider first the case in which the claimant proves permission by providing a certificate. In the short-term case, the master server could issue a certificate to a principal for a period just long enough to set up an experiment. This has the disadvantage of requiring the server to be consulted many times by the same principal if many experiments must be conducted. If a long-term certificate is issued instead, the need for repeated requests for new certificates will be reduced. However, this opens the possibility that if the principal loses privileges or suffers a compromise of his private key, then some system may be needed to *revoke* the certificate. Now consider the case in which the verifier proves permission based on a signed request from a claimant. The tradeoff between long-term and short-term certificates remain the same, but in

this case the verifier can check a local ACL for information about the claimant and act accordingly, without expecting the claimant to supply any additional certificates. An advantage of this approach is that it need not place any new responsibilities on either the ANET client or server. A claimant does not need to obtain or maintain the freshness of any certificates to make requests, and the verifier need not know how its ACL is being kept up-to-date.

Given these considerations, the use of verifier-gathered certificates provides a simpler and more modular approach to improving ABone ACL maintenance. Given a design goal for a short-term solution that entails no changes in the behavior of ANETD, the best solution is to *mirror* the ACL of the master server at each server location. We consider two protocols for secure mirroring based on an online signature from the master server and the ability of ANETD servers to establish local policies about freshness. Note that ANETD servers are clients of the master ANET server, so they are the *clients* in the following protocols:

Client Pull The client periodically requests a fresh copy of the ACL by sending a hash of its current copy. The server checks to see if the master ACL has this hash. It sends a fresh signed copy if it does not, otherwise it sends a notification that the client ACL is still up-to-date.

Server Push The server accepts requests from clients to register for updates to the master ACL and supplies the master ACL upon registration. Whenever the master ACL changes, the new ACL is signed and distributed to registered clients. Since clients may become unreachable, the server times out entries in its register so clients must periodically re-register themselves.

Details and comparative discussion of these protocols will be provided in the next section. The mirroring protocols provide a very modest degree of local autonomy to ABone nodes. A node may choose how frequently it wishes to update its ACL, but will not be able to customize the contents of that ACL. Moreover, claimants are unable to provide credentials certifying their rights, so they will need to rely on the freshness of ACLs at verifiers. If a node uses a client pull with a low frequency of update then a claimant may be unable to obtain access for a substantial period.

Although it is a substantial improvement over manual maintenance of ACLs, local autonomy over mirroring the local ACL is a somewhat weak degree of authority. To go beyond this, it is desirable to think in terms of a different architecture where ANETD and QCMD communicate in a more sophisticated manner. A possible architecture is illustrated in Figure 2. In this version of ABone security, a QCM daemon, QCMD, runs on each ABone node along with ANETD. QCMD is responsible for maintaining the policies of the node and for certificate distribution. ANETD addresses all policy questions directly to QCMD, instead of looking at a local copy of the ABone user list. When QCMD needs to perform certificate distribution, it exchanges messages with QCM daemons on other nodes.

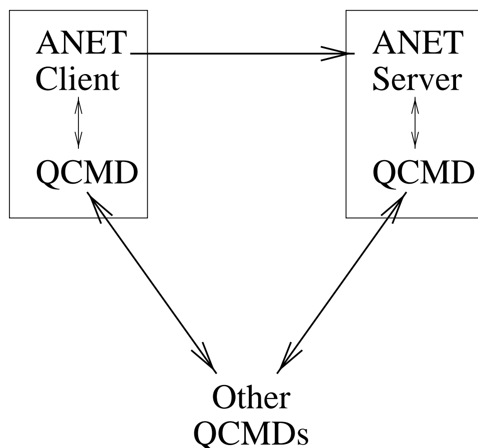

Figure 2: QCMD Communication

With this architecture, the ABone is able to take advantage of the mirroring protocols as well as the following QCM protocols:

Certificate Push Certificates are accepted from a claimant and used to avoid reference to remote policies.

Online Query/Response A QCM policy may refer to a policy at a remote principal. If a request is made for a certificate from the remote principal this is retrieved with appropriate signatures. The certificate is created dynamically using an online key.

Verify Only Verification is based only on local policies and certificates pushed by the claimant.

Offline Query/Response A collection of certificates may be created in advance with offline signing. A request for a certificate is answered with all relevant certificates thus created and the recipient constructs the necessary response from these.

The main point here is that all six of these retrieval and verification mechanisms can be made to work coherently together with each claimant and verifier choosing its own strategy. These protocols are further illustrated in Section 4.

Our overall proposal for the ABone security infrastructure is to provide a special-purpose langauge supporting six protocols to enable local autonomy with policy-directed certificate retrieval. A grammar for the language is provided in Table 1. This is simplified from what would be required for the actual system. For instance there is a need for wrappers to expose underlying data sources.

Table 1: QCM With Mirroring

$d ::= x = e$	definitions
$\mid\ x = \text{import}(data)$	import
$\mid\ x_1 = \text{pullclient}(c\$x_2, t_1, t_2)$	pull client
$\mid\ \text{pullserver}(x)$	pull server
$\mid\ x_1 = \text{pushclient}(c\$x_2, t_1, t_2)$	push client
$\mid\ \text{pushserver}(x, t)$	push server
$e ::= c$	constants
$\mid\ t$	time periods
$\mid\ x$	local names
$\mid\ (e\$x)$	global names
$\mid\ (e_1, \ldots, e_n)$	products
$\mid\ \{e_1, \ldots, e_n\}$	sets
$\mid\ \bigcup e$	set union
$\mid\ \{e \mid g_1, \ldots, g_n\}$	comprehensions
$g ::= (e_1 = e_2)$	
$\mid\ (e_1 \neq e_2)$	guards
$\mid\ (p \in e)$	generators
$p ::= x \mid c \mid (p_1, \ldots, p_n)$	patterns

3 Mirroring Protocols

Mirroring is a common strategy for certificate distribution. Under mirroring, the master ACL is kept at the master server, but all the other ABone nodes have a copy, and changes to the master ACL are propagated to the copies. Both push and pull protocols ensure a weak consistency between the master policy and the mirrors: the mirrors may be out of date with respect to the master, but changes are guaranteed to propagate within a specified time window based on the reliability of the connection between the master server and its mirrors.

3.1 Characteristics of the Protocols

In the ABone implementation, the access control lists are treated as public information, so we are not concerned about maintaining the confidentiality of the lists. However we *are* concerned about integrity; we want the copy of the access control list to be an accurate copy. By 'accurate' we mean the copy of the ACL is the same as a recent version of the central access control list. There are two ways integrity can be violated:

1. The mirrored list is corrupted—for example, it contains entries that were never in the master ACL.
2. The mirrored list is out of date—for example, the ACL contains an entry that was valid at one point but is invalid now.

The mirroring protocols use digital signatures to ensure integrity. The messages are not encrypted, since the data is not considered confidential. We are less concerned about availability since we expect the ACL to change comparatively slowly. Also, the characteristics of the server and network mean that some kinds of denial of service attacks are very difficult to stop. On the other hand, we do not want the protocol to make it easier for an adversary to mount a denial of service attack. To aid this objective, both the mirror protocols use timestamps to frustrate certain kinds of attacks. Since clocks will not be perfectly synchronized, we use a freshness threshold time period f. Timestamps are considered fresh if are within plus or minus f time units of local time.

3.2 Pull mirror protocol

The pull protocol puts the onus on the client to make sure it has an up-to-date copy of the mirrored data. The protocol is parameterized by two time descriptors: the period for sending update requests to the server, `CRequestP`, and the amount of time that the client waits for a valid server response before it resets its ACL, `CResetTO`. The parameters are passed in the declaration on the client $\mathcal{C}$:

$$x = \text{pullclient}(K_{\mathcal{S}}\$y, \texttt{CRequestP}, \texttt{CResetTO}).$$

The server $\mathcal{S}$ must declare its willingness to engage in the protocol for this data:

pullserver(y).

We use the notation $S_K(M)$ for the triple M, K, σ, where M is a message, K is a public key, and σ is the hash of M signed by the private key corresponding to K. The protocol works as follows:

1. The client periodically (with a period of `CRequestP`) checks with the server to be sure that its copy of the data is up-to-date. It does so by sending a `Changed?` message to the server. The `Changed?` message contains the name y of the policy being mirrored (e.g., `ACL`), the hash h of the client's version of the data, a timestamp $t_{\mathcal{C}}$ and the server's public key $K_{\mathcal{S}}$.

   $$\mathcal{C} \rightarrow \mathcal{S} : S_{\mathcal{C}}(\texttt{Changed?}(y, h, t_{\mathcal{C}}, K_{\mathcal{S}}))$$

   The client keeps a record of the timestamp $t_{\mathcal{C}}$, which is used to synchronize with the server's responses.

2. On getting such a request from a client, the server first checks the signature on the message, checks the freshness of the request using the timestamp in the message, and verifies that the message was meant for it (using $K_{\mathcal{S}}$). If the hash of the data at the server is different from the hash h sent by the client, then the client is out-of-date, so the server sends the new data v to the client, including the timestamp sent by the client, a new timestamp based on its current time, and the client's public key:

   $$\mathcal{S} \rightarrow \mathcal{C} : S_{\mathcal{S}}(\texttt{NewVersion}(y, v, t_{\mathcal{C}}, t_{\mathcal{S}}, K_{\mathcal{C}}))$$

   If the hashes are the same, then the client must have an up-to-date version of the data, so the server responds by sending a `NoChange` message:

   $$\mathcal{S} \rightarrow \mathcal{C} : S_{\mathcal{S}}(\texttt{NoChange}(y, t_{\mathcal{C}}, t_{\mathcal{S}}, K_{\mathcal{C}}))$$

3. When the server's response is received by the client, it checks the signature and freshness as before, then verifies that the message was meant for itself, and then confirms that the $t_{\mathcal{C}}$ in the server's response matches the value it remembered.

   If the above checks succeed, and the message received by the client contains a new version of the policy being mirrored, then the client updates its policy. Otherwise the policy is left unchanged.

4. If `CResetTO` time passes since the last response was received from the server (responses could be lost due to network congestion or a malicious agent in the network) the client resets the local copy of the policy to the null set.

The protocol preserves data integrity: because the reply from the server is signed, the client can be sure that the data was not tampered with, and, because it contains the timestamp, the client knows it is fresh. Moreover, we can guarantee that anybody who is in the local copy of the policy was in the server's master copy at some point in time. The worst damage an attacker could cause is to deny everyone access to a client by blocking traffic between the client and server for a sufficiently long time, thus causing the client's copy to reset. We believe this is better than allowing access to someone who is no longer in the server's policy, which could happen with a stale local copy.

The timestamps are essential to the security of the protocol and are intended to reflect recommendations in the ISO/IEC standard for entity authentication using digital signatures [15]. Consider an alternate version of this protocol in which we eliminate the timestamps. Of course, this would allow a replay attack; an adversary could save old messages and then send them later to confuse clients. This weaker protocol may also enable a kind of denial of service attack we call *traffic amplification*. Adversaries could exploit the protocol to effectively amplify the amount of junk traffic that they can generate.

If an adversary $\mathcal{E}$ wanted to overload the network connection of $\mathcal{C}$, then $\mathcal{E}$ could simply send junk packets to $\mathcal{C}$. But $\mathcal{E}$'s ability to overwhelm $\mathcal{C}$ is limited by the bandwidth of $\mathcal{E}$'s network connection. The traffic amplification problem could occur if the adversary $\mathcal{E}$ saved a `Changed?` message from $\mathcal{C}$. $\mathcal{E}$ could then send the message over and over again to the server $\mathcal{S}$. If $\mathcal{S}$ responded to each message by sending a large file to $\mathcal{C}$ then the resulting traffic may clog $\mathcal{C}$'s network connection. Thus $\mathcal{S}$ amplified $\mathcal{E}$'s ability to clog $\mathcal{C}$'s bandwidth.

The timestamps allow the server and clients to ignore messages which are not fresh. An adversary can only clog $\mathcal{C}$'s connection while the `Changed?` message is fresh. If the server keeps track of which timestamps it has seen then the attack can be prevented entirely; the server can discard messages that contain timestamps that the server has already seen. This does not put much burden on the server because timestamps only have to be saved until they become stale.

3.3 Push mirror protocol

The push approach puts the onus on the server to make sure that changes to the data are propagated to the clients. The protocol is parameterized by three time descriptors: a client side re-registration period `CRegisterP`, a server side registry flush period `SRegisterTO` and a client side policy reset timeout `CResetTO`. Two parameters are passed in the declaration on the client:

$$x = \text{pushclient}(K_{\mathcal{S}}\$y, \texttt{CRegisterP}, \texttt{CResetTO}).$$

The remaining parameter is supplied by the server declaration:

$$\text{pushserver}(y, \texttt{SRegisterTO}).$$

Here is the protocol:

1. The client tells the server that it wants to receive updates for the policy y. It sends to the server the name of the policy, a hash of the current local copy of the policy, a timestamp, and the server's public key.

   $$\mathcal{C} \rightarrow \mathcal{S} : S_{\mathcal{C}}(\texttt{RegisterMe}(y, h, t_{\mathcal{C}}, K_{\mathcal{S}}))$$

2. The server performs checks for the signature and freshness and whether y is available as a push server, then adds the client to its table of registered clients if these checks succeed. It sends an immediate `NoChange` or `NewVersion` message as in the previous protocol. After that, whenever the server's policy y changes, the server sends updates through `Update` messages to registered clients:

   $$\mathcal{S} \rightarrow \mathcal{C} : S_{\mathcal{S}}(\texttt{Update}(y, v, t_{\mathcal{S}}, K_{\mathcal{S}}))$$

3. Whenever the client receives an `Update` message, it checks the signature, freshness, and origin. If these succeed, it updates the local copy of y to v.

4. After `SRegisterTO` time passes the server removes the client from the table of registered clients. The server will not send any more updates to the client until it receives a new `RegisterMe` message. The client will re-register by sending a new `RegisterMe` every `CRegisterP` time.

5. As in the pull protocol, if an update from the server is not received for `CResetTO` time, the client resets its local copy of the policy to the null set.

3.4 Which Strategy: Push or Pull?

The protocols have different advantages that depend on the kind of data that is mirrored and the network capabilities of the client and server. If the data changes infrequently then the push protocol may be more appropriate since messages will only be sent when the data actually changes. The push protocol also allows faster propagation of changes since the change can be passed on to the client immediately. The pull protocol allows the client to control its interaction with the server. If a client does not want frequent updates or can only connect with the server at certain times (at midnight, for example) then the pull protocol would be more appropriate.

We have left open the question of whether the protocols are implemented reliably or unreliably and, in the case of the push protocol, whether updates from changes to the data are sent to registered clients by unicast or multicast. The experiments we describe next used unreliable unicast (UDP), and we have implemented a reliable unicast (TCP) version of the push protocol.

3.5 Experiments

Since mirroring is intended to provide short-term support for ABONE security, we wanted to know about how many nodes could be supported by these protocols. Test data is unavailable currently. Various attempts have been made to

simulate certificate retrieval, such as using DNS resolvers as a source of traffic information [18], but we are not convinced that this is worth the trouble for us, given the likely differences between DNS, which has a retrieval mechanism based on referrals and on-demand caching, and the system we propose, which uses mirroring. Moreover, it is somewhat questionable whether access control information for the ABone has a traffic profile at all similar to resolution of domain name bindings. Hence we have used a straight-forward stress model that assumes extremely frequent registrations of ABone users. We measure the failure rate of the protocols under this stress. Our experiments were conducted on a cluster of 5 dual Pentium-II machines running Linux and an UltraSparc machine running SunOS. We ran an ANETD/QCMD server on the UltraSparc, and 500 client ANETD/QCMD's on the cluster, 100 per machine. We conducted the following experiments:

1. In the first experiment, we started all clients on the cluster almost simultaneously. All 500 clients executed the pull protocol, with a 60 second gap between successive requests for the ACL. As a result, the server had to deal with very intense but short bursts of requests. We found that on the average, about 70% of the requests sent by the clients were dropped by the server.

2. In a second experiment, we staggered the startup of the clients so that the demand on the server was more constant. Again, all clients executed the pull protocol. We found in this case that the server could handle all 500 clients well, and that no requests were being dropped.

3. Finally, we tried a mixture of push and pull clients. 250 of the clients used the push protocol, while the rest used the pull protocol as before. Again, the server could handle all requests sent to it, and could get updates out to the push clients on time.

The experiments suggest that mirroring will scale well beyond 500 nodes. In practice, we expect the master access list to change very slowly, so a 60 second delay in updates is overkill. Mirroring will be able to handle the projected growth of the ABone for the immediate future. However, since the system is currently being deployed, we will have the ability to conduct more direct tests of this claim in the future.

4 Distribution Beyond Mirroring

Let us now discuss other protocols required to provide more substantive support for local autonomy of ABone nodes. We describe how policy-based certificate retrieval can be achieved for the ABone by describing the QCM system protocols with sample ABone policies as examples. The basic functionality provided by QCM is to securely evaluate a policy to a table. If the policy is defined in terms of remote policies, then QCMD is responsible for securely obtaining those policies. Once QCMD has produced the table, ANETD can use it to decide

```
{ ("alice.com",    Principal(RSA-MD5("r8K+gZ4ZRo5usA675..."))),
  ("bob.com",      Principal(RSA-MD5("udoN0w7B0K65hhwpw..."))),
  ("careless.org", Principal(RSA-MD5("umVy3uv1LpaSx7W83..."))) }
```

Figure 3: QCM syntax for the table of Figure 1.

whether to authorize requests. The grammar for the language is provided in Table 1. To keep things simple we have omitted the lifetimes for certificates; our implementation provides and checks expirations to prevent replay attacks and accidental use of old data. More details on the QCM implementation can be found in [7], and a formal semantics is provided in [6].

4.1 Strategies for Supporting Local Autonomy

Security policies are often given in the form of a table. For example, access matrices, public key directories, and access control lists can all be thought of as tables, as can the ABone policy given in Figure 1. QCM was designed to support table-based security: in QCM every policy defines a table. The syntax for table definition is illustrated in Figure 3, which gives the QCM equivalent of the ABone policy of Figure 1. For backwards compatibility, we can also provide notation to import ABone-style lists from files into QCM's internal format: `import("hosts.allow")` is the QCM table (policy) obtained by reading in the ABone file `hosts.allow`.

In QCM policies are defined and controlled by principals who give them names. For example, a principal K_1 could give an access control list the name `ACL`, or a public key directory the name `PKD`. Nothing prevents another principal K_2 from assigning different definitions to the names `ACL` and `PKD`. To distinguish between the policies of different principals, we use global, or fully-qualified, names: K_1`$ACL` (pronounced "K_1's `ACL`"), or K_2`$ACL`.

When a principal needs to distribute its policies out into the network, it typically does so using a signed document. Such a document cannot be used without verifying the principal's signature, and this requires the principal's public key. To ensure that a principal's key is available when needed, we identify principals with their keys; that is, in QCM, principals *are* keys. Names and the strategy of using keys as principals are used in other policy languages, and we borrowed our notation for principals from one of them, SDSI [16]. Some examples appear in the table of Figure 3. Since principals are long, we usually abbreviate them with K, K', etc.

We can now give the QCM policy of the master ABone server, $\mathcal{S}$: the server has a public key $K_{\mathcal{S}}$ that is widely known, its policy is a definition

```
ACL = import("hosts.allow"),
```

and other ABone nodes can refer to the server's policy as $K_{\mathcal{S}}$`$ACL`.

We still need a way for an ABone node to define a policy that incorporates the server's policy, but overrides it where desired. QCM supports this through

composite policies that refine, augment, and combine the policies of multiple principals. `ACL1`, below, is a composite policy.

```
ACL1 = { (h,k) | (h,k) <- K_S$ACL, h != "careless.org" }
```

Here, '`|`' stands for 'where,' and '`<-`' stands for 'is an entry of' and `!=` stands for 'not equal to' $\neq$. Thus, in words, the policy says

> *ACL1 is a table with entries (h,k), where (h,k) is an entry of $K_{\mathcal{S}}$$ACL, and h is not "careless.org".*

So, `ACL1` discards some unwanted entries from the server's table. Entries can also be added, using union:

```
ACL2 = union(ACL1,
             { ("claire.com", K_claire) })
```

where `union(e1, e2)` is a syntactic sugar for $\bigcup$`e1, e2` ($\bigcup$ is an operator that takes a set of sets as an argument and returns the union of those sets a result).

Finally, a composite policy can be built from the policies of multiple principals:

```
ACL3 = { (h,k) | (h,k) <- K_S$ACL, (h',k') <- K_bob$ACL,
                 h=h', k=k' }
```

This is the intersection of the `ACL`'s of $K_{\mathcal{S}}$ and K_{bob}: `(h,k)` is only an entry of `ACL3` if it appears in the policies of *both* $\mathcal{S}$ and Bob. This is how QCM can define policies that depend on the joint authority of multiple principals.

For example, suppose a client ANETD uses the policy `ACL1` from above to decide whether to grant requests to run active network experiments:

```
ACL1 = { (h,k) | (h,k) <- K_S$ACL, h != "careless.org" }
```

If ANETD receives a request originating at `alice.com` and signed by K_{alice}, it needs to find out whether `(alice.com,`K_{alice}`)` appears in the policy `ACL1`. It uses QCMD to find out, by asking it to evaluate the following policy to a table:

```
{ "yes" | ("alice.com",K_alice) <- ACL1 }
```

If `("alice.com",`K_{alice}`)` is an entry of `ACL1`, the policy will evaluate to the table `{"yes"}`. Otherwise, the policy will evaluate to the empty table, `{ }`. So, if QCMD's answer is `{"yes"}`, ANETD grants the request, otherwise, the request is denied.

Policy evaluation is easy if all of the policies can be gathered in one place, as we see here:

```
K_S$ACL = { ("alice.com",    K_alice),
            ("bob.com",      K_bob),
            ("careless.org", K_careless) }

ACL1 = { (h,k) | (h,k) <- K_S$ACL, h != "careless.org" }

{ "yes" | ("alice.com",K_alice) <- ACL1 }
```

For example, to evaluate `{ "yes" | ("alice.com",`K_{alice}`) <- ACL1 }`, first evaluate `ACL1` to a table, then iterate over every entry. If `("alice.com",`K_{alice}`)` appears, then add the entry `"yes"` to the result. `ACL1` can be evaluated similarly. The final answer is `{"yes"}`.

Usually, however, the necessary policies will not be available at every node. In this case QCMD must use some kind of retrieval protocol to obtain the remote policies. We support a variety of protocols, each with advantages and disadvantages for the parties involved. The rest of this section discusses the protocols and their tradeoffs.

4.2 Certificate Push

Most systems for verifying access control policies will not retrieve missing certificates. Instead, they require certificates to be presented to the local policy engine by the party who wants to be authorized. We call this a 'push' protocol because certificates are not requested by the policy engine, they are supplied as inputs by some out-of-band means. QCM supports a push protocol using certificates of the following form.

```
<Document = Member("ACL",("alice.com",K_alice)),
 Signature = "mQinGCBzKGtza4X6...",
 Signer = K_S>
```

The certificate says that `("alice.com",`K_{alice}`)` is an entry of the table $K_{\mathcal{S}}$`$ACL`. For the certificate to be valid, its signature must have been produced by the signing key of $K_{\mathcal{S}}$; this can be verified with $K_{\mathcal{S}}$. (We also support certificates with expiration dates.) If Alice presents this certificate to ANETD along with her request, ANETD simply passes it on to QCMD. QCMD verifies the signature on the certificate, uses it to construct an approximation to the table $K_{\mathcal{S}}$`$ACL`, and adds the approximation to the local collection of policies:

```
K_S$ACL = { ("alice.com",K_alice) }

ACL1 = { (h,k) | (h,k) <- K_S$ACL, h != "careless.org" }

{ "yes" | ("alice.com",K_alice) <- ACL1 }
```

At this point, QCMD proceeds with local policy evaluation as usual, giving the answer `{"yes"}`. Of course, the approximation is not the actual value of the table $K_{\mathcal{S}}$`$ACL`. QCM has an important *monotonicity* property that justifies use of the approximation. The table that QCM computes for the policy is an approximation of the real value of the policy. Monotonicity means that if QCM starts off with a better approximation for remote policies, then it computes a better approximation for the result. This means that if QCM says that `"yes"` is an entry of the table, then no additional information about $K_{\mathcal{S}}$`$ACL` could imply that `"yes"` is not in fact an entry.

Monotonicity guarantees that no request is granted when the remote policy says it should be denied. But there is no guarantee that all requests that should be granted, will be granted. If Alice presents an invalid or expired certificate

with her request, then under this protocol her request will be denied even though she appears on the master access list at $\mathcal{S}$. This places a significant burden on Alice: she has to manage her certificates, and periodically request new ones as they expire. Virtually all policy languages use the 'push' approach, but we believe it is unrealistic to expect unsophisticated users to manage their own certificates. At the very least, users will require some automated support to help them manage certificates. And it may be necessary for the administrators of the system (e.g., the ABone nodes) to take on more responsibility for certificate distribution. Our other protocols show how this can be handled transparently in QCM, without any policy changes.

4.3 Online Query/Response

When QCM needs the value of a policy that is not available locally, its default retrieval policy is to obtain it using a secure query/response protocol. The protocol involves two QCM daemons, a client that requests the policy, and a server that supplies it.

In our example, the client C runs at an ABone node, and the server is running at the master ABone node at $\mathcal{S}$. To obtain $K_{\mathcal{S}}$`$ACL`, C can invoke the following protocol.

1. The client sends a query asking for the value of the policy $K_{\mathcal{S}}$`$ACL` to the server. (Recall that the principal $K_{\mathcal{S}}$ can be tagged with the location of its server.)

   $C \rightarrow \mathcal{S}$: `Query(`$K_{\mathcal{S}}$`$ACL)`

2. The server has the complete definition of the policy $K_{\mathcal{S}}$`$ACL`, so it can evaluate the query by simply looking up the table. It sends the table, T, back to the client:

   $\mathcal{S} \rightarrow C : S_{\mathcal{S}}$(`Response(`$h$`, `$T$`)`)

   The response contains a one-way hash h of the query, which can be used to ensure that this is the answer to the question asked rather than a replay of an answer to a different question.

3. The interpreter verifies the signature on the response using the key $K_{\mathcal{S}}$, and uses the hash to match up the response with the query.

The policy asked in a `Query` message does not have to be a policy name like $K_{\mathcal{S}}$`$ACL`; it could be any policy at all. In our example, a better query would be:

$C \rightarrow \mathcal{S}$: `Query({"yes" | ("alice.com",`K_{alice}`) <- `$K_{\mathcal{S}}$`$ACL})`

Instead of signing and returning the entire table $K_{\mathcal{S}}$`$ACL`, the server just checks whether `("alice.com",`K_{alice}`)` is an entry, and returns `{ "yes" }` or the empty table accordingly. QCMD uses query optimization to automatically choose queries that result in smaller responses.

The query/response protocol is invoked automatically by QCMD—ANETD does not have to do anything different. Just as with the push protocol, ANETD formulates the query it wants answered, then submits the query and any certificates it might have received from Alice to QCMD. QCMD uses whatever certificates it can, and sends queries to obtain remote policies when it needs to.

The server could of course refuse to answer the client's query, or, the server might be down. If so, the client QCMD returns an error. Another idea is to allow QCMD to assume its response is the empty table and continue evaluation. Since the empty table is a valid approximation for any other table, monotonicity guarantees that this cannot cause the request to be granted when it should be denied. Other available certificates might be sufficient to grant the request, so the client will continue with evaluation.

4.4 Verify Only

The normal QCMD evaluation strategy is to take whatever certificates are pushed at it, use them to the extent possible, and use query/response to obtain any other needed certificates. Sometimes the client may want to rule out even this limited query/response. Therefore we provide a mode, called verify-only mode, in which QCMD never invokes query/response. In this mode, a request will only be granted if all necessary certificates are presented to the client up front.

4.5 Offline Query/Response

The query/response protocol we described above required the server to sign responses online. The server at $\mathcal{S}$ might prefer not to have the signing key online. In that case, pre-signed responses can be prepared on a machine not attached to the network, and stored on the server. Then the protocol works as follows.

1. The client formulates its query, Q, as usual and sends it to the server.

   $\mathcal{C} \rightarrow \mathcal{S} : \texttt{Query(}Q\texttt{)}$

2. On startup, the server is given the certificates that were signed offline. The certificates describe the content of tables defined by $K_{\mathcal{S}}$; the server can reconstruct the tables from the certificates. Once the server has the tables it can evaluate Q; as it does so, it remembers what entries in the tables are accessed, and what certificates contributed to the entries. The server does not return the answer that it computes, since it cannot sign it; instead it returns the required certificates. Each individual certificate is signed, so the complete message does not have to be signed.

   $\mathcal{S} \rightarrow \mathcal{C} : \texttt{Certificates(}c_1\texttt{,...,}c_n\texttt{)}$

3. The client receives a `Certificates` message instead of the `Response` message that it would have received if the server was doing online signing. The client checks each of the signatures on the supplied certificates, and evaluates the original query in verify-only mode, using the certificate push protocol to take the supplied certificates into account.

5 Conclusion

We have described a system and policy description language to maintain access control infrastructure for the ABone experimental testbed. Our approach is based on an extension of the QCM system and provides a significant degree of local autonomy as well as support for policy-directed certificate retrieval. We have described two secure mirroring protocols and how they can be integrated with a collection of other protocols. The protocols have been implemented, and this paper describes some simple tests to measure the scalability of the mirroring protocols for the ABone. We end with a brief discussion of some additional issues: revocation, integration with other policy and certificate systems, and control of loops.

If long-term certificates are used, then there is usually demand for a revocation mechanism, that is, an ability to announce that a valid certificate which is properly formed and not expired should no longer be respected by verifiers. This introduces many complexities. We have developed a way to do policy-directed certificate retrieval with revocation: [6] describes a language and analyzes its security model rigorously. We have also developed an implementation for an internal language supporting this model of revocation (an external language would be used by policy writers and then compiled into the internal language). These extensions could be added to the language in Table 1 at the cost of more complexity than we could discuss in this paper. We refer the reader to [6] for details. In its initial deployment, QCMD will avoid both long-term certificates and revocation.

QCM uses its own certificate formats, but there does not appear to be any impediment to using X.509v3 formats if this would aid interoperability with other systems. It is unlikely that any single policy description system will satisfy all needs, so our expectation is that some ‘glue’ between such systems will be necessary. QCM seems best suited for coarse-grained access control such as the ABONE rather than fine-grained access control like the policy of a reference monitor in an operating system. An interface for using QCM for access control in the PLAN EE [10] was developed by Hicks [8] and used to develop an active firewall application [9]. In this case, QCM was used to determine policies about which network services various agents were allowed to use. Efficiency was enhanced by caching information about QCM verification decisions.

Another interesting problem with QCM is the threat of circular dependencies such as a situation in which principal A delegates to prinicipal B and principal B delegates to principal C and principal C delegates to principal A. This problem can be addressed by assuming that such circles of common interest work out

a reasonable delegation structure among themselves ‘out-of-band’. Research is underway on an automated solution; for instance QCM queries could carry more information about their origins in order to detect the loop dynamically.

We have developed implementations for all of the protocols described in this paper, but not yet in a unified system. The mirroring portion of the interface has been implemented for deployment on the ABone, and this deployment is currently underway. Once deployed we hope to gain additional insights about load and interface requirements, as well as experience with what kinds of local autonomy its users demand.

Acknowledgements.

We appreciated help and encouragement from the ABone developers, especially Steve Berson, Bob Braden, and Livio Ricciulli. The work was supported by DARPA Contract N66001-96-C-852, ONR Contract N00014-95-1-0245, and NSF Contract CCR94-15443.

References

[1] D. Eastlake 3rd and C. Kaufman. Domain name system security extensions. IETF Proposed Standard RFC 2065 (Updates RFC 1034 and RFC 1035), January 1997.

[2] Steve Berson, Bob Braden, and Livio Ricciulli. Introduction to the ABONE. `http://www.isi.edu/abone/DOCMUMENTS/ABoneIntro.ps`, March 2000.

[3] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming: Issues in Distributed and Mobile Object Systems*, 1999.

[4] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the 17th Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, 1996.

[5] Carl M. Ellison, Bill Frantz, Ron Rivest, and Brian M. Thomas. SPKI certificate documentation. `http://www.clark.net/pub/cme/html/spki.html`.

[6] Carl A. Gunter and Trevor Jim. Generalized certificate revocation. In Thomas Reps, editor, *Conference Record of POPL ’00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–329, Boston, MA, January 2000. ACM.

[7] Carl A. Gunter and Trevor Jim. Policy directed certificate retrieval, June 2000. To appear in Software Practice and Experience.

[8] Michael Hicks. PLAN system security. Technical Report MS-CIS-98-25, Department of Computer and Information Science, University of Pennsylvania, April 1998.

[9] Michael Hicks and Angelos D. Keromytis. A secure PLAN. In Stefan Covaci, editor, *Proceedings of the First International Workshop on Active Networks*, volume 1653 of *Lecture Notes in Computer Science*, pages 307–314. Springer-Verlag, June 1999. Extended version at `http://www.cis.upenn.edu/~switchware/papers/secureplan.ps`.

[10] Mike Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93, Baltimore, Maryland, September 1998. ACM Press.

[11] R. Housley, W. Ford, W. Polk, and D. Solo. *Internet X.509 Public Key Infrastructure: Certificate and CRL Profile*. IETF RFC 2459, January 1999.

[12] Timothy A. Howes, Mark C. Smith, and Gordon S. Good. *Understanding and Deplying LDAP Directory Services*. Network Architecture and Development Series. Macmillan, 1999.

[13] ISO/IEC 9594–1. *Information technology—Open Systems Interconnection—The Directory: Overview of concepts, models and services*, 1997. Equivalent to ITU-T Rec. X.500, 1997.

[14] ISO/IEC 9794–8. *Information technology—Open Systems Interconnection—The Directory: Authentication framework*, 1997. Equivalent to ITU-T Rec. X.509, 1997.

[15] ISO/IEC 9798-3. *Information technology–Security techniques–Entity authentication–Part 3: Mechanisms using digital signature techniques*, October 1998.

[16] Butler Lampson and Ron Rivest. SDSI—a simple distributed security infrastructure. `http://theory.lcs.mit.edu/~cis/sdsi.html`.

[17] C. Liu and P. Albitz. *DNS and BIND*. O'Reilly & Associates, 1992.

[18] Patrick McDaniel and Sigih Jamin. Windowed revocation. In Raphael Rom and Henning Shulzrinne, editors, *Proceedings of the Nineteenth IEEE Computer and Communication Society Infocom Conference*, Tel Aviv, Isreal, March 2000.

[19] P. Mockapetris and K. Dunlap. Development of the domain name. *ACM Computing Reviews*, 18(4):123–133, 1988. Also in Proceedings ACM SIGCOMM '88 Symposium, August 1988.

[20] Livio Ricciulli. Service configuration and management in adaptable networks. In *Tenth Annual IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, 1999.

[21] Livio Ricciulli and Phillip A. Porras. An Adaptable Network COntrol and Reporting System (ANCORS). In *Integrated Network Management*, Boston, 1999.

[22] W. Yeong, T. Howes, and S. Kille. *Lightweight Directory Access Protocol*. IETF RFC 1777, 1995.