Policy-Directed Certificate Retrieval

Technical Report MS–CIS–99–07

Carl A. Gunter and Trevor Jim
University of Pennsylvania

September 1998

Abstract

Any large scale security architecture that uses certificates to provide security in a distributed system will need some automated support for moving certificates around in the network. We believe that for efficiency, this automated support should be tied closely to the consumer of the certificates: the policy verifier. As a proof of concept, we have built QCM, a prototype policy language and verifier that can direct a retrieval mechanism to obtain certificates from the network. Like previous verifiers, QCM takes a policy and certificates supplied by a requester and determines whether the policy is satisfied. Unlike previous verifiers, QCM can take further action if the policy is not satisfied: QCM can examine the policy to decide what certificates might help satisfy it and obtain them from remote servers on behalf of the requester. This takes place automatically, without intervention by the requester; there is no additional burden placed on the requester or the policy writer for the retrieval service we provide. We present examples that show how our technique greatly simplifies certificate-based secure applications ranging from key distribution to ratings systems, and that QCM policies are simple to write. We describe our implementation, and illustrate the operation of the prototype.

1 Introduction

Current research on languages for expressing security policies provides algorithms for deciding whether to grant requests based on certificates signed by trusted parties. These systems assume that the relevant certificates are present, leaving the collection of the certificates to some separate, unspecified mechanism. We have implemented a prototype system, QCM (Query Certificate Manager), that verifies policies based on certificates submitted by the requester, and in addition can automatically obtain missing certificates on behalf of the requester. We argue that a policy verifier capable of retrieving certificates on its own is more efficient and convenient than current verifiers, and does not compromise security.

Figure 1 displays a common architecture for verification systems. Arrows

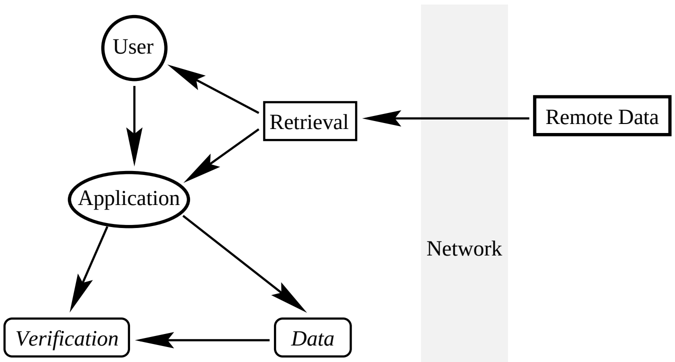


Figure 1: A Security Architecture

indicate a flow of certificates. For example, an application will supply certificates to the verifier to see whether a policy is satisfied, and the verifier may draw more certificates from a local database. These local certificates typically have been collected by the user in advance from remote sources using some retrieval mechanism (possibly the application itself).

An example of this is PGP [18], a popular system supporting secure electronic mail. A mail application can use PGP to verify signed e-mail messages: the PGP verifier performs this function by examining public key certificates held in a PGP key ring (the database). The key ring itself is built up over time by the user, who can retrieve certificates for inclusion in the key ring by browsing web sites (e.g., public key servers), receiving them by e-mail, or similar means. Systems such as PolicyMaker [2] and SPKI/SDSI [5] follow this same architecture, but seek to make the verifier more general so that it can work with many different kinds of applications.

There are some obvious inefficiencies in this architecture when the right certificates are not available. First, it brings the user into the process. For example, when the user wants to send secure mail to someone, but does not have the right public key certificate, he must obtain it (by using a web browser or exchanging e-mail), place it on the key ring, and invoke PGP again. As certificate-based systems become widespread, the user will spend more and more time obtaining certificates; clearly, this won't scale.

It is possible to remove the human element by making applications smart enough to retrieve missing certificates without user intervention, though few such systems exist today. The problem with this idea is that it introduces *duplication* of two kinds: between the verifier and the application, and between different applications.

To see how there can be duplication between the verifier and the application, suppose the e-mail application needs to send an encrypted message to Bob, and the PGP policy is, "rely on either Alice or Trent for key bindings." The application invokes the verifier, which examines the policy and then looks for

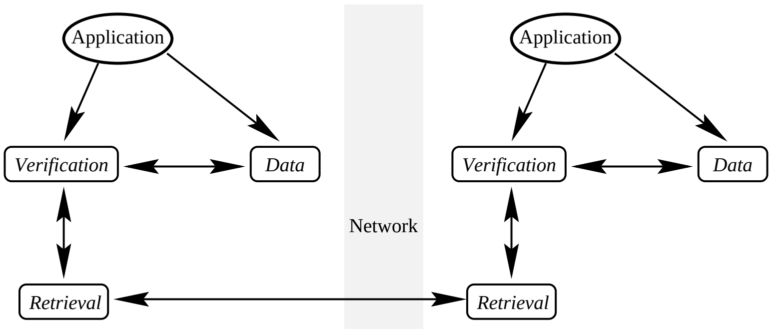


Figure 2: QCM Architecture

a certificate for Bob signed by Alice or Trent in the local database. If no such certificate is found, it reports failure to the application. The smart application would then study the policy to determine a query to send to the key server: "give me any key certificates for Bob signed by Alice or Trent." So, the verifier and the application both perform what we call *policy-directed certificate retrieval*: they examine the policy to determine what certificates to retrieve. The only difference is that in the case of the verifier, the certificates are retrieved from the local database, while in the case of the application, they are retrieved over the network from the key server. This means that the logic for understanding policies is duplicated in the verifier and the application, and will be executed not once, or even twice, but three times: once for the failed verification, a second time by the application to formulate the query to the key server, and a final time by the verifier when the application submits the retrieved certificates for approval.

Another sort of duplication exists between different applications. An application that wants to have automated certificate retrieval may not be able to re-use the retrieval mechanism of the e-mail application, for example. The code might be proprietary, or e-mail specific, or the writer of the application might not trust the writers of the e-mail system. This is not a source of inefficiency in itself, but it is a decided inconvenience. The policy languages and verifiers of systems like PolicyMaker and SPKI/SDSI were made as general as possible to eliminate just this sort of duplication. We believe that this approach needs to be taken not only for verification, but policy-directed retrieval as well.

To address these issues we propose a new architecture in which the verifier itself can direct both a retrieval mechanism and a local database. This is pictured in Figure 2. As in the previous architecture, users and applications can still obtain and supply certificates to the verifier and the local database. However, the verifier can invoke a retrieval mechanism directly to obtain certificates if necessary, and can store them in the local database (e.g., for caching). The retrieval system can also act as a server, accepting requests from the network, submitting them to the verifier, and returning a signed response. This is very convenient for users and applications, who do not have to be concerned with most of the details of certificate acquisition. And it should be more efficient because we do not suffer the penalty imposed by a system in which policy-directed retrieval is duplicated between the application and the verifier.

The new architecture does not sacrifice security for efficiency and convenience. Of course, retrieval is time-consuming, and automated retrieval could be the target of a denial-of-service attack that tries to provoke the server into making many queries. But this is not particular to our verifier architecture: an application providing automated retrieval, like the smart e-mail application, would be as vulnerable. It is easy to turn off retrieval for cases where it is inappropriate, so we provide a *verify-only* mode for our system, in addition to its default *verify-retrieve* mode. In verify-only mode, the system will act just like SPKI or PolicyMaker, and will verify policies on the basis of locally available certificates only; it will never try to retrieve missing certificates.

It could be argued that a verifier with automated retrieval will be larger, more complicated, and hence less secure than a verifier based on the model of Figure 1. We have found that this is not so. Our system, QCM, which combines a verifier, retrieval mechanism, and local data storage mechanism, is about the same size as SDSI 2.0, which does not provide retrieval. This is because the tasks of verification, retrieval, and local data management have much in common, so we can share code between them. This re-use will become apparent when we describe our implementation. QCM is simple enough to be given a formal semantics, which enhances the correctness, and, therefore, the security of the system.

In designing QCM we have taken a conservative approach and based it on a well-established language, the language of sets that forms the core of most database languages (SQL, the relational calculus, and so on). This language serves as the policy language and the query language of QCM. Verification in QCM consists of a database evaluation, while certificate retrieval corresponds to distributed database evaluation. Database evaluation has been well-studied over the past 30 years, so QCM can take advantage of extensive research into query optimization and distributed database implementation: when QCM needs to query for missing certificates, we use well-known techniques to choose queries that minimize message traffic.

A principal advantage of QCM's policy language is that the policy writer does not have to write code that explicitly makes remote queries. By examining the policies, QCM automatically detects when queries need to be made, formulates the queries, sends them out, and collates the replies. This makes policies written in QCM much simpler to write and understand. Exactly how this works will be hinted first by example, then explained in more detail when we describe the QCM implementation.

Overall, we have three goals for QCM: first, to show that policy-directed certificate retrieval greatly simplifies the task of building certificate-based secure applications; second, to show that it does not require writing complicated policies; and third, to explain and illustrate our methods in enough detail that they can be applied to other verification systems. For example, it should be possible to create a policy-directed retrieval system for SPKI. This paper focuses on our implementation and examples of its use; we also aim to explain some of the design space for QCM and the reasons for the choices we made within that space.

Outline. In Section 2, we will introduce QCM and policy-directed certificate retrieval by example. Section 3 discusses the design of a system to carry out automatic retrieval and describes how these issues are resolved in our system. Section 4 describes our implementation. A detailed example is presented graphically in Section 5, and we conclude with related work and future directions in Section 6.

2 Policy-Directed Certificate Retrieval

In this section we give an informal introduction to QCM and policy-directed certificate retrieval. We will show how policies are written and evaluated, beginning with a simple key distribution example that we explain in full. We then present some more interesting examples of policies at a high level. For reference, we give a high-level syntax of QCM in Table 1. QCM has the usual assortment

Table 1: The Syntax of QCM

$e ::= c$	constants
$\mid\ x$	local names
$\mid\ (e\$x)$	global names
$\mid\ (e_1, \ldots, e_n)$	products
$\mid\ \{e_1, \ldots, e_n\}$	sets
$\mid\ \mathbf{union}(e_1, \ldots, e_n)$	set union
$\mid\ \{e \mid g_1, \ldots, g_n\}$	comprehensions
$g ::= (e_1 = e_2)$	guards
$\mid\ (p \in e)$	generators
$p ::= x \mid c \mid (p_1, \ldots, p_n)$	patterns

of constants (3, “Alice”, **true**, …). In particular, *principals* are constants in QCM. As in SDSI, SPKI, and PolicyMaker, principals are public keys. For brevity we will use K to range over principals here, but in our prototype we use a SPKI-like syntax for principals. In QCM a principal can come attached with a URI that identifies a server for the principal’s certificates; again for brevity, this does not appear explicitly in the notation we are using here. Section 5 gives an example of our full syntax of principals.

We use SDSI’s concept of linked, local namespaces: $K\$x$ is the *global name* of the *local name* x in K’s namespace, and is pronounced, “K’s x.” In QCM, a global name always refers to a set. A typical example is

$$K\$PKD = \{\, (\text{“Alice”}, K_{\text{alice}}), (\text{“Bob”}, K_{\text{bob}}) \,\}$$

This defines $K\$PKD$ to be a global name refering to a set of (user,key) pairs. This set can be thought of as a public key directory: it says that Alice's key is K_{alice} and Bob's key is K_{bob}. It can also be thought of as K's *policy* about Alice and Bob's keys.

The directory $K\$PKD$ is under K's control; it may be kept in secure storage on K's machine, for example. Other principals can only determine the contents of $K\$PKD$ through certificates signed by K or by querying K's server (if K has a server). However, QCM does not contain any features for explicitly using certificates or making queries. Policies just refer to global names like $K\$PKD$, and the QCM evaluator will automatically use certificates or make remote queries when appropriate.

2.1 A basic example

To illustrate how policies can drive certificate retrieval in a simple case, consider a policy that says to rely on $K\$PKD$ to determine Alice's key(s):

$$\mathit{AliceKeys} \quad = \quad \{\, k \mid (\text{“Alice”}, k) \in K\$PKD \,\}.$$

The expression on the right is a *set comprehension* denoting the set of keys paired with "Alice" in $K\$PKD$: it contains every key k such that there is a pair $(\text{“Alice”}, k)$ in the set $K\$PKD$. *AliceKeys* might be evaluated in the context of a certificate signed by K:

$$K \textbf{ says } (\text{“Alice”}, K_{\text{alice}}) \in PKD.$$

This is our notation for a document containing a signature (not shown), a signer (K, or more accurately, the private key corresponding to K), and an assertion about a global name (that $(\text{“Alice”}, K_{\text{alice}})$ is a member of the set $K\$PKD$). When QCM is given this certificate, it can evaluate *AliceKeys* to the result

$$\{\, K_{\text{alice}} \,\}.$$

QCM can actually evaluate *AliceKeys* given a collection of certificates, for example,

$$\begin{array}{l} K \textbf{ says } (\text{“Bob”}, K_{\text{bob}}) \in PKD, \\ K' \textbf{ says } (\text{“Alice”}, K') \in PKD. \end{array}$$

Here QCM would detect that neither certificate is relevant: the first because it gives Bob's key according to K, and not Alice's key; the second because it gives Alice's key according to K', and not K. The result of evaluation would be the empty set,

$$\{\,\}.$$

If QCM is asked to evaluate *AliceKeys* without any certificates, and K has a server, it will send a query to the server to obtain appropriate certificates.

(Recall that QCM can look at K since it appears in *AliceKeys*, and K may be tagged with a server location.) A query is a QCM expression, just like a policy. In this case two appropriate queries are

$$\begin{array}{rl} & K\$PKD \\ \text{and} & \{\, k \mid (\text{“Alice”}, k) \in K\$PKD \,\}. \end{array}$$

The first query asks for the entire set $K\$PKD$, while the second query asks for just Alice's key(s). An answer to either query would allow QCM to calculate the final result, but the second query is likely to produce a smaller reply from K's server. QCM uses standard query optimizations to choose the second query—so in this case, the query is in fact the expression defining *AliceKeys*. Call this expression P.

When K's server receives this expression, it will submit it to its own QCM evaluator, which has access to the definition of $K\$PKD$. The result will be $\{K_{\text{alice}}\}$, which can be returned in a certificate

$$K \textbf{ says } P \supseteq \{K_{\text{alice}}\}.$$

After checking the signature on the certificate, the first QCM evaluator can calculate the final result,

$$\{\, K_{\text{alice}} \,\}.$$

Notice that the reply from the server states that $P \supseteq \{K_{\text{alice}}\}$, not $P = \{K_{\text{alice}}\}$. This is because the server is free to base its evaluation on any membership certificates it might have. A membership certificate does not give the exact value of a set; it only gives an approximation of a set. Therefore, QCM is a 'best effort' system that only calculates approximate answers to queries.

Also, notice that since the signed reply from K's server includes the query, and the server does not know in advance what queries it will receive, it could not have prepared the certificate in advance. This means that the private key has to be online and available to the server, and the server will have to sign each response dynamically. This may sometimes be appropriate, but it places a burden on the server (signing is expensive) and makes the private key more vulnerable to compromise.

Therefore QCM also has an offline signing mode, where the server works with pre-signed certificates, and no private key needs to be online. For example, the server could be provided with the following certificates.

$$\begin{array}{l} K \textbf{ says } (\text{“Alice”}, K_{\text{alice}}) \in PKD \\ K \textbf{ says } (\text{“Bob”}, K_{\text{bob}}) \in PKD \end{array}$$

From these certificates, the server's QCM evaluator can recover the full definition of $K\$PKD$, and evaluate a query, like P, that refers to $K\$PKD$. As it evaluates the query, it keeps track of what certificates were useful in producing the answer, and these certificates make up the reply. So the response to P would be the certificate

$$K \textbf{ says } (\text{“Alice”}, K_{\text{alice}}) \in PKD.$$

Now, the original QCM evaluator has to do some more work, because the reply does not answer its query directly—its query asked for a key, and the certificate contains a pair of a string and a key. What we do is evaluate the query, P, on the original server in verify-only mode, with the certificates from the reply as input. This gives the final answer,

$$\{\, K_{\text{alice}} \,\}.$$

In our prototype we have adopted the convention that it is the server's decision whether to use online or offline signing, and anyone making a query should be prepared for either kind of reply.

So far we have shown how QCM can obtain Alice's key by examining policies and certificates, and possibly exchanging messages. Sometimes we do not want to obtain Alice's key, but, rather, check whether a particular key is Alice's key (for example, to verify the signature on an e-mail message). In QCM this amounts to asking a membership query:

$$\mathbf{member}(K_{\text{alice}}, \mathit{AliceKeys})\mathbf{?}$$

Intuitively, a membership query should evaluate to "yes" or "no," or more accurately, "yes" or "not sure" (if we are working with membership certificates that only give us partial information about the set in question, we will not be able to give a definite "no"). Up until now, all of our queries have evaluated to sets, so it might seem that we need to define a new kind of evaluation. But the two kinds of evaluation have much in common, so we have instead used a trick that lets us use set evaluation to perform membership evaluation. QCM considers the membership query above as an abbreviation for a set comprehension that evaluates to either the empty set, or to a singleton set:

$$\{\, \text{“yes”} \mid K_{\text{alice}} \in \mathit{AliceKeys} \,\}.$$

The expression will evaluate to the set $\{\text{“yes”}\}$ if QCM can determine that K_{alice} is in $\mathit{AliceKeys}$, and otherwise will evaluate to the empty set $\{\,\}$. In the first case, the answer to the membership query is "yes," and in the second case, the answer is "not sure." For example, if the query was submitted to QCM along with the certificate

$$K \ \mathbf{says}\ (\text{“Alice”}, K_{\text{alice}}) \in \mathit{PKD},$$

then QCM would answer "yes," without sending any queries. If no certificates were supplied along with the query, QCM would contact K about K_{alice}, and in the end, it returns the same result, "yes." If no certificates were supplied and QCM were put into verify-only mode, it would not contact K, and could only answer "not sure."

That summarizes the basics of policy-directed certificate retrieval. The remainder of this section gives more examples of policies that can be written in QCM, without discussing how QCM evaluates the policies. We hope that these examples will show that interesting policies are easy to write in QCM, and that our policy-directed certificate retrieval service can simplify the task of writing secure applications. The specifics of policy-directed certificate retrieval are given in subsequent sections, followed by an extended example which we use to illustrate the prototype in more detail.

2.2 A web of trust

The last example showed how a public key directory is expressed in QCM. We now show how multiple public key directories can be combined in a way similar to PGP's 'web of trust' [17].

In the web of trust, users specify *introducers*, principals that are to be relied on for key bindings. In QCM this can be expressed as follows.

$$\begin{array}{l} \mathit{introducers} = \{K_1, K_2, K_3\} \\ \mathit{local} = \{\ (u,k) \mid x \in \mathit{introducers}, (u,k) \in x\$\mathit{PKD}\ \} \end{array}$$

This defines *local* to be a public key directory that is the union of the introducers' directories: $K_1\$\mathit{PKD}$, $K_2\$\mathit{PKD}$, and $K_3\$\mathit{PKD}$. This constitutes a 'chain of trust' of length 1: Alice trusts principals she knows personally to provide key bindings. Like PGP, QCM supports trust chains of arbitrary length. For example, we could have defined Alice's local directory using a chain of length 2 as follows.

$$\begin{array}{l} \mathit{introducers2} = \mathbf{union}(\mathit{introducers}, \\ \qquad\qquad\qquad\qquad \{\ k \mid x \in \mathit{introducers},\ (v,k) \in x\$\mathit{PKD}\ \}) \\ \mathit{local2} = \{\ (u,k) \mid x \in \mathit{introducers2},\ (u,k) \in x\$\mathit{PKD}\ \} \end{array}$$

Here *introducers2* consists of *introducers* as well as any key k appearing in the *PKD* of an introducer. Similarly, *local2* is the union of the *PKD*'s of principals that Alice knows personally, or who are known by a principal known by Alice.

For example, given the certificates

$$\begin{array}{l} K_2\ \mathbf{says}\ (\text{"Bob"}, K_{\mathrm{bob}}) \in \mathit{PKD}, \\ K_{\mathrm{bob}}\ \mathbf{says}\ (\text{"Alice"}, K_{\mathrm{alice}}) \in \mathit{PKD}, \\ K_{\mathrm{alice}}\ \mathbf{says}\ (\text{"Carol"}, K_{\mathrm{carol}}) \in \mathit{PKD}, \end{array}$$

QCM can evaluate the query

$$\{\ k \mid (\text{"Carol"}, k) \in \mathit{local2}\ \}$$

to $\{K_{\mathrm{carol}}\}$. And in verify-retrieve mode, if QCM were only given the first two of these certificates, it would query K_{alice} for Carol's key.

In this way, we can program any finite length chain of trust in QCM (though in practice, trust chains will be short, since longer chains are considered less secure).

2.3 A DNS-like secure database

We now show how to use QCM to define a secure hierarchical database similar to DNSSEC [1]. We have chosen to structure our database in a slightly different way than DNS, to make it easier to perform reverse lookups.

Our database has a root server K_{root} that understands the top-level structure of domain names. It maintains a table,

$$K_{\mathrm{root}}\$\mathit{DNSTable} = \{\ (\text{“com”}, 128, K_{\mathrm{com}}), (\text{“edu”}, 182, K_{\mathrm{edu}})\ \}.$$

The table is a set containing triples that explain the mapping of the top-level DNS domains. For example, the triple $(\text{“com”}, 128, K_{\mathrm{com}})$ means that com maps to 128 and the principal in charge of the com domain is K_{com}. K_{com} maintains a similar table,

$$K_{\mathrm{com}}\$\mathit{DNSTable} = \{\ (\text{“ibm”}, 66, K_{\mathrm{ibm}}), (\text{“dec”}, 99, K_{\mathrm{dec}})\ \}.$$

K_{com}'s table says that K_{ibm} is in charge of IBM's domain, and K_{dec} is in charge of DEC's domain. It is crucial that responsibility for different parts of the database be split up in this way, since IBM and DEC are competitors.

Each table defines a part of the DNS mapping. The full mapping is defined by a *resolver* that queries the various servers for table entries. For example, the following QCM definition resolves the full mapping up to depth two:

$$\begin{aligned} \mathit{DNS} = \{\ ((d_2, d_1), (a_1, a_2)) \mid\ & (d_1, a_1, k_1) \in K_{\mathrm{root}}\$\mathit{DNSTable}, \\ & (d_2, a_2, k_2) \in k_1\$\mathit{DNSTable}\ \}. \end{aligned}$$

It specifies how a table entry of the root, K_{root}, can be combined with a table entry of a principal k_1, provided that k_1 is a principal in charge of a subdomain in the root's table. For example, a typical element of *DNS* would be $((\text{“ibm”}, \text{“com”}), (128, 66))$.

Once we have defined *DNS*, we can use QCM to evaluate some interesting queries:

- *DNS*: return the entire mapping.
- $\{\ a \mid ((\text{“ibm”}, \text{“com”}), a) \in \mathit{DNS}\ \}$: return the address of ibm.com.
- $\{\ d \mid (d, (128, 66)) \in \mathit{DNS}\ \}$: return the domain with address 128.66.
- $\{\ d \mid ((d, \text{“com”}), a) \in \mathit{DNS}\ \}$: return the subdomains of com.

QCM can evaluate these queries in a manner similar to DNS: it starts by querying the root, learns who is in charge of subdomains, and sends those principals queries as well.

As usual, QCM can also incorporate certificates into its computation. For example, it might be presented with a certificate giving an entry in the root table:

$$K_{\mathrm{root}}\ \textbf{says}\ (\text{“com”}, 128, K_{\mathrm{com}}) \in \mathit{DNSTable}.$$

QCM can use the certificate to short-circuit queries to the root: for example, to find the address of ibm.com it can contact K_{com} directly, without contacting K_{root}. The certificate is in effect a cache-line of K_{root}'s table.

2.4 A SPKI-like authorization system

In SPKI [5], authorization certificates and access control lists (ACL's) are the means by which principals make statements about what authorizations they grant to other principals. 5-tuple reduction is the way that SPKI evaluates such statements to see whether they imply that a given principal has a particular authority.

In this section we describe how SPKI's authorization mechanism can be simulated in QCM. SPKI ACL's will be written as QCM programs, SPKI authorization certificates will correspond to QCM certificates, and QCM program evaluation will take the place of 5-tuple reduction.

A SPKI ACL can be thought of as a table associating principals with authorizations called *tags*. Such a table can be defined directly in QCM:

$$\begin{aligned} K_0\$ACL = \{\ &(K_1, \text{“read /etc/passwd”}), \\ &(K_2, \text{“write /etc/motd”})\ \}. \end{aligned}$$

This defines the ACL of K_0, which states that K_1 should be allowed to read the file /etc/passwd, and so on. K_0 might sign a QCM authorization certificate:

$$K_0 \textbf{ says } (K_1, \text{“read /etc/passwd”}) \in ACL.$$

This certificate contains three of the five parts of a SPKI 5-tuple: the issuer (K_0), the subject (K_1), and the authorization (“read /etc/passwd”). A fourth part, indicating a time period over which the certificate is valid, is omitted from our presentation, but is implemented in our prototype.

In SPKI, K_0 can use a fifth part, the delegation field, to say that K_1 is permitted to delegate the authority granted to it by K_0 to other principals. In QCM K_0 can delegate to K_1 by including all or part of K_1's ACL in its own. For example, K_0 can assert that K_1 is allowed to assign authority on its behalf using the following definition.

$$K_0\$ACL = \textbf{union}(K_1\$ACL, \ldots)$$

The corresponding QCM delegation certificate looks like

$$K_0 \textbf{ says } ACL \supseteq K_1\$ACL.$$

K_0 can limit the degree to which it delegates to K_1. For example, K_0 can specify that K_1 is only permitted to delegate the authorizations in an arbitrary set A by including the following set in its ACL:

$$\{\ (k, a) \mid (k, a) \in K_1\$ACL, a' \in A, a = a'\ \}. \tag{1}$$

Then K_0's ACL will include any entry (k, a) in K_1's ACL, as long as a is in the allowed set A. In other words, the authorizations that K_0 grants to k are the intersection of A and the authorizations that K_1 grants. This is exactly the behavior of SPKI's tag intersection. Abbreviating the expression (1) with $\mathbf{delegate}(K_1, A)$, we can express limited delegation in a QCM certificate

$$K_0 \textbf{ says } ACL \supseteq \mathbf{delegate}(K_1, A).$$

SPKI 5-tuple reduction can then be simulated by taking these kinds of QCM certificates and evaluating QCM queries in verify-only mode. For example, to find what authorizations K_0 grants to K_1, we evaluate

$$\{\, a \mid (K_1, a) \in K_0\$ACL \,\}.$$

The current implementation of QCM uses strings as tags, while SPKI specifies a more elaborate algebra of tags, including encodings of infinite sets ("any string with prefix http://www.ietf.org/"). We may add such tags to QCM in future versions, as we learn more about the needs of end applications.

3 Design Issues and Decisions

In designing a policy-directed certificate retrieval system we encountered many issues and tradeoffs that had to be resolved before we were able to implement a usable prototype. The primary issues were: what security guarantees we should try to provide; how to handle privacy of communications, policies, and data; how to manage both certificates that are supplied ('pushed') and those that are retrieved ('pulled'); whether to support online or offline signing; what to do about failure and partial information in a distributed context; and how to control resource utilization. We discuss each of these issues in turn here. In another paper we provide a formal semantics for QCM and express its correctness properties mathematically. Key issues there include the formal security model, concurrency, failure, and what the query optimizer must satisfy.

Security assumptions and guarantees. The first priority of QCM is to preserve the integrity of policies even though communication takes place over the untrusted network. This means that a distributed QCM computation taking place over the network should be consistent with a non-distributed QCM computation in which all policies are gathered together on a single, secure machine. In other words, if a distributed QCM computation says "Alice's key is K_{alice}," then the non-distributed computation should say the same.

We ensure policy integrity by a variety of mechanisms. QCM data is always accessed by public key (the principal, e.g., the K of $K\$x$), and the response of a QCM server is always signed by that key. QCM also time stamps certificates, and discards certificates that have timed out. Of course, if trust is delegated to a principal K, and K's private key is compromised, integrity is lost. We do guarantee that if private keys are never contained in QCM definitions, then QCM does not reveal the keys.

QCM aims to relieve applications from dealing directly with certificates. In particular, communication between an application and QCM does not need to be signed. However, this means that the communication link between QCM and the application must be secure, generally because this communication will occur on a secure machine. Security of communication between application processes is not directly maintained by QCM, although QCM can be of help. For example, QCM can be used by the applications to obtain each other's public keys, but it is up to the applications to use those keys to secure their communication. QCM does ensure that data transferred between QCM processes cannot be tampered with.

Certificates are not the only mechanism that QCM can use to preserver policy integrity. For example, if a QCM process expects that it will have to make many queries to a particular server, it could establish a long-term connection to the server protected by a symmetric session key. Replies from the server would not be certificates, but rather their *contents*; for example, instead of a certificate K **says** $K' \in S$, K would send $K' \in S$ over the link, suitably encrypted. This would be useful because symmetric cryptography is much more efficient than public key encryption or signing.

Privacy of communications, policies, and data. Although QCM could form part of a system to support privacy, QCM itself does not attempt to provide privacy of communications, and provides only limited privacy for policies and data. In this our philosophy is similar to that of DNSSEC [1], where certificates are provided to anyone who wants them and no effort is made to encrypt the information maintained by the DNS servers. QCM is not incompatible with privacy, since queries and responses could easily be encrypted in an extension of the system, or could be kept private by use of a virtual private network. But ignoring the issue of access control for the policies themselves let us greatly simplify the system.

Anyone can query a QCM server to obtain information on its policies (the sets $K\$x$ that it maintains), but QCM answers queries extensionally rather than intensionally, so complete information about policies is not revealed. For instance, if a principal K defines a set $S = K'\$R$, then a query to K about S will only yield elements of $K'\$R$, and not the fact that K' was consulted to obtain them. Thus policies and data can be known in only a limited way. In future work we may extend QCM to return intensional results. This might help clients frame better queries, or indicate where clients could obtain useful certificates on their own.

Push and pull of certificates. A key question is how certificates 'pushed' into QCM by an application or another QCM node should interact with queries that the QCM node generates to 'pull' information from other QCM nodes. Consider the applications and QCM servers pictured in Figure 3. Let A, B, C, D be principals with associated applications and QCM servers. Suppose that A's application makes a request of B's application, and B's application is programmed to approve the request if A appears on an ACL, maintained in its local QCM server. B's QCM server defines this list by delegation to C, and C in

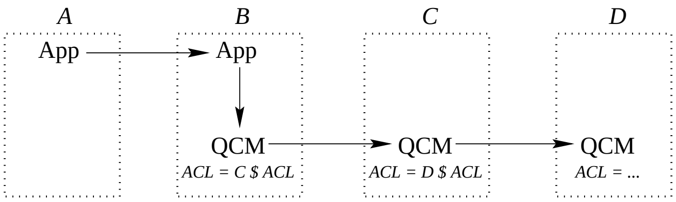


Figure 3: Pushing and Pulling Certificates

turn delegates the definition of its ACL to D. Now, when A submits its request to B, then, assuming that verify-retrieve mode is used by B and C, the QCM server of D is eventually asked whether A is in $D\$ACL$. A's application can try to speed this up by supplying certificates to B's application, which B can then supply to QCM. For instance, if A supplies a certificate C **says** $A \in C\$ACL$, then B's QCM server will use this information instead of sending a query to C.

The interesting issue arises when A supplies a certificate D **says** $A \in D\$ACL$ to B. By itself, this is not enough for B's QCM server to determine that $A \in B\$ACL$. In fact, as far as B knows, the certificate is not even relevant to the query. In QCM at present, then, B's server ignores the certificate and queries C directly, which queries D in turn.

An alternative would be for B's server to 'push' the certificate along to C when it makes the query. This would let C's server avoid a query to D. We decided not to do this because it seems just as likely that the certificate would not be of use to C, and pushing it along would therefore waste bandwidth. Strategies for pushing certificates intelligently through a distributed QCM computation would require some means of knowing who might make use of them, and we have not yet experimented with this enough to know how it could be automated.

Offline versus online signing. In online signing, the server signs responses to queries as they go out. In offline signing, the server does not have a key to sign responses; it only has a set of certificates that were pre-signed by some offline principal. Responses in the offline case consist of a set of these certificates.

Neither signing method is clearly superior to the other. Offline signing offers more protection to the private key of the signer, and the server does not have to continuously sign responses, which could be expensive. However, there is an added burden of coordination between the signer and the server to maintain the certificates, which typically time out periodically. It is also more expensive for the client, who might have to verify signatures on a set of certificates instead of just one, and do some extra work to extract the answer to the query from the returned certificates.

One of the primary design objectives of QCM was to achieve greater automation even in the presence of diverse kinds of servers. We have therefore sought to provide support for networks of QCM nodes in which some nodes provide online signing while others provide offline signing and where some nodes operate in verify-only mode while others operate in verify-retrieve mode. The main question about QCM is whether a sufficient level of automation can be achieved to adequately support typical applications without excessive intervention of users.

Failure and partial information. An essential aspect of QCM is that it is a distributed system, and must therefore cope with failure. If a server is not responding or if there is any other kind of error it is essential to carry out the computation in a safe way. A simple approach would be for QCM to halt and report back to the user or application when an error or unavailable server is detected.

We have tried a more sophisticated approach in which computation can continue even if some servers are down. This is possible because we have designed QCM around sets: every global name in QCM refers to a set, certificates communicate information about sets, and QCM policies and queries are expressions that evaluate to sets. Thus when failure occurs, we have a default result to fall back on: the empty set, which approximates all other sets. For example, if QCM is given a query $\mathbf{union}(K_1\$ACL, K_2\$ACL, K_3\$ACL)$, and the server of K_2 is unavailable, it will work on $\mathbf{union}(K_1\$ACL, \{\,\}, K_3\$ACL)$ instead. Quite often, we can return useful results even if failure occurs.

In theoretical terms, QCM provides a monotonic data approximation. That is, the less failure is encountered, the larger (and more accurate) the response provided. This allows us to assert exactly what can be known for sure about a QCM response regardless of the failures that may have been encountered.

Control of resource utilization. Currently, we provide no access control for QCM data, and it is possible for a single user to make many requests for data, or a single request for an extremely large amount of data. Web servers and DNS name servers are similarly vulnerable to this kind of abuse. It would be useful for QCM to have a way to deal with requests that are taking more time than expected or allowed, but we have not implemented one yet. Another danger is the creation of request cycles in distributed QCM programs, which could lead to infinite loops. Cycles come about because of mutually recursive QCM definitions, and the only solution in QCM right now is not to write such programs. We leave a better solution to these problems to future research.

4 Implementation

Before describing the details of the implementation, it is helpful to recall the *expressions* of Table 1, page 5, and also introduce some new syntax.

$d ::= (K\$x = e)$	definitions
$P ::= (d_1; \ldots; d_n; e)$	programs
$v ::= c \mid (v_1, \ldots, v_n) \mid \{v_1, \ldots, v_n\}$	values
$I ::= (e \supseteq e')$	inclusions

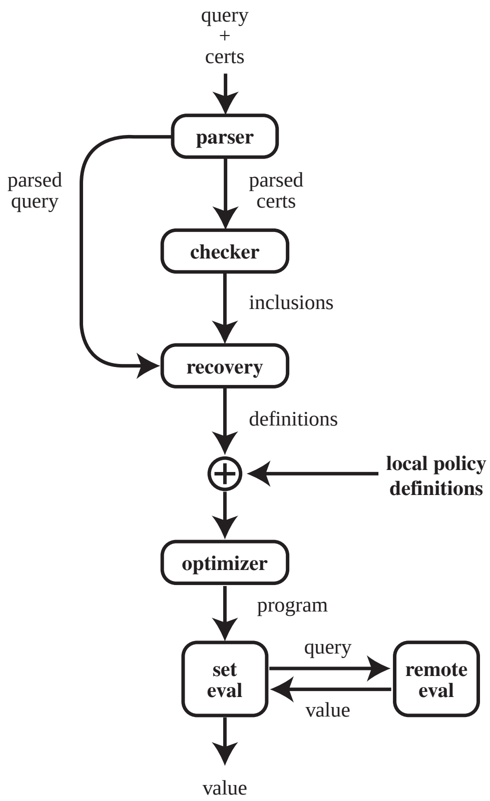


Figure 4: QCM evaluation

Evaluation of a QCM expression proceeds in the presence of a set of *definitions* for global names. A QCM expression together with such a set of definitions is called a *program*. The *values* are the results of QCM program evaluation; notice that they are a subset of the expressions (they are the fully evaluated expressions). The *inclusions* are the assertions of certificates, and they state that one set expression includes another.

4.1 The local evaluator

An application invokes QCM by giving it a query (i.e., an expression) and a set of certificates. This input passes through a number of phases, structured as in Figure 4. We describe each phase in turn.

Parsing The first phase parses the query and certificates into QCM's internal abstract syntax. This is a security-critical step: QCM assumes that the application treats the certificates as black boxes (strings of bytes) which are obtained from a possibly hostile source, and which could therefore be syntactically ill-formed. Furthermore, although the application has formed the query, its actions could have been influenced by an adversary, so we need to check the form of the query as well.

If the parser detects any errors in the input, it reports them to the application, and evaluation halts.

Checking After the certificates have been parsed, their signatures are verified and we check their *provenance*. Informally, this means that a principal should only sign a certificate that makes a statement about its own names. For example, K **says** $K'\$x \supseteq \{4,5\}$ is rejected if $K \neq K'$. A certificate such as K **says** $x \supseteq \{4,5\}$ is accepted, because unqualified names in a certificate are assumed to be implicitly qualified by the signer of the certificate.

When checking provenance, we examine only the left-hand sides of the inclusions in the certificates. For example, a certificate of the form K' **says** $K'\$x \supseteq K\$x$ is allowed, even though it is signed by K' and refers to $K\$x$. This is how K' can delegate power over $K'\$x$ to K; in other words, the signer is permitted to give up control. Conversely, a certificate K **says** $\{\} \supseteq K'\$x$ is not allowed, since otherwise K could force $K'\$x$ to be empty. We can formally prove that our restrictions give each K control over its own names.

The output of the checking phase is the set of inclusions asserted by the acceptable certificates.

Recovery The recovery phase takes in a set of inclusions and constructs a set of definitions for the names appearing on the left of the inclusions (we say the definitions are *recovered* from the inclusions). These definitions will be used in evaluating the query at the local node, and *they may not be the same as the actual definitions.* Instead, they are the "best approximation" to the actual definitions that can be constructed from the available inclusions. For example, given inclusions $K\$x \supseteq \{3,4\}$ and $K\$x \supseteq K'\$y$, the recovery phase would construct the definition $K\$x = \mathbf{union}(K'\$y, \{3,4\})$. We have a formal definition of "best approximation," and a proof that our recovery phase produces one.

The recovery phase checks the resulting definitions to make sure that they are not circular. We do not allow mutually recursive definitions, because they are harder to evaluate (though it should be possible to extend QCM to handle recursive definitions by using Datalog evaluation techniques).

Optimization The recovered definitions are combined with local policy definitions (the local database) and the parsed query to form a program, which is passed through an optimization phase. Our optimizer uses standard techniques to transform the program into a more efficient program. This includes deciding the form of any remote queries that will be sent during the next phase, set evaluation.

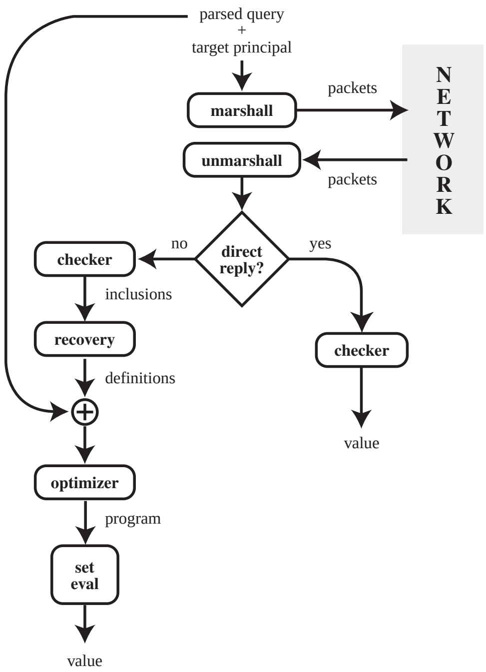


Figure 5: QCM remote evaluation

Set evaluation Finally, the program is run through the set evaluator, producing a value that is returned to the calling application. The value is unsigned, because applications should not have to perform cryptography to use QCM. This does not compromise security because in all local computations we are already forced to rely on the security of the operating system of the local machine.

If the program refers to global names that do not appear in its definitions, the set evaluator may need to make remote queries to obtain information about the names. The optimization phase has previously determined what queries to make, and the actual queries are carried out by a remote evaluation phase which we describe next. The set evaluator can also be configured to make no remote queries (verify-only mode). In this case, whenever a remote query is indicated, the evaluator simply assumes that the query would result in the empty set, and continues without sending any messages.

4.2 The remote evaluator

The phases of the remote evaluator are given in Figure 5. The remote evaluator takes as input a parsed query and a principal to whom the query should

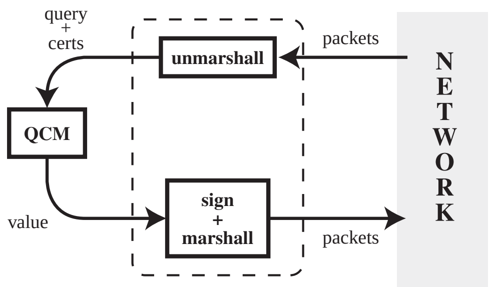


Figure 6: A QCM server with online signing

be directed. It first marshalls the query into a format suitable for network transmission, possibly adding some information that will help coordinate the response with the query. (The evaluator is multi-threaded and there can be multiple outstanding queries.) The marshalled query is then shipped to the principal over the network.

When the reply arrives, it is unmarshalled and parsed. Several kinds of replies are possible for any given query. A *direct reply* is the easiest to deal with. If the query is e and the target principal is K, then a direct reply is a certificate of the form K **says** $e \supseteq v$. In this case, the remote evaluator just checks the signature on the certificate and returns the value v as the answer to the query.

In general, a remote principal can only make a direct reply if it keeps its private key online, because the query must be signed and included in the reply. The remote principal might want to keep its private key offline to make it less vulnerable. If this is the case, the remote evaluator will not get a direct reply to its query, but rather will receive a set of certificates, signed offline by the principal, that can be used to answer the query.

An offline reply can easily be handled by phases that we have already described. Essentially, we use the local evaluator to process the reply. The remote evaluator passes its original query and the certificates that it got in response from the target principal through the checking, recovery, optimization, and set evaluation phases of the local evaluator. The set evaluator is put in verify-only mode, so that it does not invoke the remote evaluator recursively. The resulting value is returned to the original invocation of the local evaluator (the one that invoked the remote evaluator in the first place).

4.3 A QCM server with online signing

Now that we have seen how applications on a single machine can use the QCM local evaluator to query local policies, it is easy to create an application that acts as an online QCM server, processing queries on the local policies for other machines on the network. Figure 6 shows how this works. The server waits for queries to come in from the network. When a query arrives, it is unmarshalled and sent to the local evaluator, resulting in an unsigned value. The value is then signed (the server has the private key), marshalled, and shipped back across the network. The marshalling and unmarshalling phases are the complements of the unmarshalling and marshalling phases of the remote evaluator.

4.4 Query evaluation with offline signing

We have already described the client side of QCM's offline signing mechanism: when the remote evaluator makes a query and receives a collection of offline certificates in response, it runs the query and certificates through its local evaluator in verify-only mode to get the answer to the query. We now explain the server side of the offline signing mechanism: how a server with a collection of offline certificates chooses what certificates to return in response to a query.

Offline signing for the server requires very little new functionality, because we are able to apply existing phases to the problem. In particular, we re-use set evaluation. The idea is that given a query e, we can construct a query e' that evaluates to the same result as e, along with the set of certificates it used along the way.

This is best illustrated by example. Suppose a server is given the following certificates, which were signed offline.

$$\begin{array}{l} K \ \textbf{says} \ (\text{“Alice”}, K_{\text{alice}}) \in \mathit{PKD} \\ K \ \textbf{says} \ (\text{“Bob”}, K_{\text{bob}}) \in \mathit{PKD} \\ K \ \textbf{says} \ K_{\text{alice}} \in \mathit{Superusers} \end{array}$$

Call these certificates C_1, C_2, and C_3. From these certificates, the server can construct approximations of $K\$\mathit{PKD}$ and $K\$\mathit{Superusers}$ called PKD' and $\mathit{Superusers}'$, instrumented to keep track of certificates. Every element of PKD' will be an element of $K\$\mathit{PKD}$, paired with the set of certificates that prove that the element is a member of $K\$\mathit{PKD}$; and similarly for $\mathit{Superusers}'$.

$$\begin{array}{rcl} \mathit{PKD}' & = & \{\, (\{C_1\}, (\text{“Alice”}, K_{\text{alice}})),\ (\{C_2\}, (\text{“Bob”}, K_{\text{bob}}))\, \} \\ \mathit{Superusers}' & = & \{\, (\{C_3\}, K_{\text{alice}})\, \} \end{array}$$

Now suppose that the server receives the query

$$\{\, x \mid (x, k) \in K\$\mathit{PKD},\ k' \in K\$\mathit{Superusers}, k = k' \,\}.$$

This finds the names of all superusers. The server will transform the query to use PKD' in place of $K\$\mathit{PKD}$ and $\mathit{Superusers}'$ in place of $K\$\mathit{Superusers}$, and also keep track of certificates:

$$\begin{array}{l} \{\, (\textbf{union}(m, n), x) \mid (m, (x, k)) \in \mathit{PKD}', \\ \qquad\qquad\qquad\qquad\qquad (n, k') \in \mathit{Superusers}', k = k' \,\}. \end{array}$$

The idea is that whenever $(x, k) \in K\$\mathit{PKD}$, we know this because of the certificates in the set m, and similarly, if $k' \in K\$\mathit{Superusers}$, we know this because

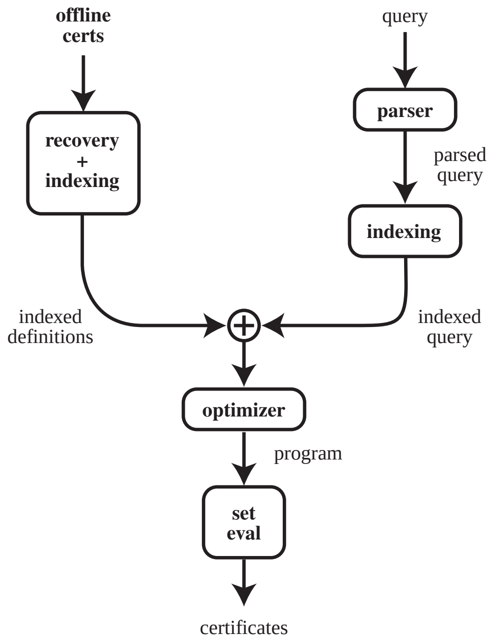


Figure 7: Offline QCM evaluation

of the certificates in n. So if x ends up in the result, it is due to the certificates in both m and n.

Call this instrumented query Q. It evaluates to

$$\{ (\{C_1, C_3\}, \text{“Alice”}) \}.$$

This is the set of elements of that would be calculated by the original query, each paired with a set of certificates. The full set of certificates used to calculate the result is given by

$$\{ x \mid (y, z) \in Q, x \in y \}.$$

The server can simply run this through the evaluator and return the resulting set of certificates.

Figure 7 summarizes the process. The offline certificates are first passed through a recovery and indexing phase, to produce definitions that take certificates into account (like PKD' and $Superusers'$ above). We call these *indexed definitions*.

To process a query, we first parse it, then pass it through an indexing phase so that it refers to the indexed definitions, calculates the value of the original query and the relevant certificates, and discards the value (like the last query above). The indexed query and definitions are then passed through the optimizer and set evaluator, in verify-only mode. The resulting set of certificates is used as the final answer.

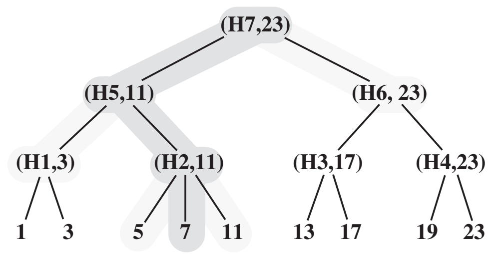


Figure 8: A 2-3 hashtree and the membership path for 7

4.5 Offline certificate update and refresh

In offline signing, a principal creates certificates with its private key at a location isolated from the network, and supplies them by some means to one or more servers that will answer queries from the network on the principal's behalf. Notice that the server is maintaining both sets (e.g., the value of some $K\$x$) and *proofs* of set membership (e.g., a certificate K **says** $K\$x \supseteq \{7\}$).

These proofs of set membership are not trivial to maintain over an extended period. Policies (sets) are likely to change over time, so the offline principal must communicate new proofs (certificates) to the server periodically. Moreover, certificates are typically marked with a time interval over which they are valid. Even if the offline principal's policies never change, it will have to "refresh" the certificates on the server as they expire. If we use the certificates that we have described up to now, this would mean that at the end of every expiration period, the offline principal would have to sign and transmit n certificates of the form K **says** $K\$x \supseteq \{v\}$ for each set $K\$x$ of size n.

To support more efficient offline certificate update and refresh, we have implemented a second kind of membership certificate, based on a scheme of Naor and Nissim [13]. The same scheme supports *non*-membership certificates, which will be needed for revocation.

The main idea is that both the offline signer and the online server will maintain the set in a balanced binary hash tree data structure. We use 2-3 hash trees, whose interior nodes have either 2 or 3 children. Figure 8 contains an example. This is a 2-3 hash tree for the set $\{1, 3, 5, 7, 11, 13, 17, 19, 23\}$, whose elements appear as the leaves of the tree, in increasing order. Each interior node of the tree contains a hash and the maximal leaf reachable from the node. The hash value of a node is the hash of the values at the children of the node. For example, H2 = hash(5,7,11) and H5 = hash(H1,3,H2,11). We use a one-way hash function (SHA), so that signing the root hash value is as good as signing the whole tree—it is computationally infeasible to find a tree with the same root hash.

Both the offline signer and its online server maintain a copy of the tree. When the offline signer needs to bring the server's tree up to date, it sends a change list to the server, along with a signature for the new root hash. The server applies the change list (insertions and deletions) to its tree, thereby obtaining an exact copy of the offline signer's new tree. Insertion and deletion take $O(\log n)$, so an update on a set of size n takes time $O(m \log n)$, where m is the size of the change list. The important thing about these bounds is that they show that we can efficiently maintain not only the set, but also the *certificates of membership and nonmembership* for the set.

A membership certificate for an element of the set consists of the signed root hash, and just enough of the tree to prove that it has the element as a leaf. "Just enough" turns out to be the path from the root to the leaf, plus the children of all nodes in the path. For example, to be convinced that 7 is an element of the set of Figure 8, we need the path from the root to 7 (in the dark outline), plus any children of the path (in the light outline). We just have to check the hashes (H7 = hash(H5,11,H6,23), H5 = hash(H1,3,H2,11), H2 = hash(5,7,11)) to verify that 7 is a leaf of a tree with root hash H7.

Naor and Nissim showed that the same idea can be used to show that some x is *not* a member of the set. We implemented a simplification of their method based on the following observation.

Note that the algorithm for testing whether x is a member of the set is deterministic: start at the root, move to the leftmost child whose "max leaf" value is greater than or equal to x, and repeat until a leaf is reached. If the leaf is x, then x is a member of the set, otherwise, x is not a member of the set. The parts of the tree examined in a failed search for x are exactly the same as those needed for a membership certificate: a path from the root to a leaf, and all children of the path. Verifying that such a structure proves non-membership is just the same as proving membership, except that we must also check that we have a *correct search path* for x that does not end in x.

For example, the outlined structure of Figure 8 is enough to convince us that 8 is not an element of the set. We can easily verify that the hashes are correct and that the path does not end in 8. Furthermore by examining the "max leaf" values of the path and its children, we can see that this is the correct search path for 8. Thus the structure shows that 8 is not a leaf of the tree. The structure also shows that 9 and 10 are not leaves; but it does not show that 12 is not a leaf. The search path for 12 would start with the right child of the root, since the max leaf of the left child is 11.

These new "tree" certificates are transparently handled by QCM: we just extended the checking and recovery phases to handle both our original certificates and tree certificates.

4.6 Revocation

We are currently extending QCM to support certificate revocation. In this section we discuss how we plan to do this, and some of the subtleties that come up.

One way to add revocation to QCM is to add the operation of set difference.

For example, suppose K issues a certificate for Alice's key:

$$K \textbf{ says } (\text{“Alice”}, K_{\text{alice}}) \in PKD.$$

If Alice's key becomes compromised, K might want to revoke the certificate. This could be done by maintaining a set containing the compromised bindings:

$$K\$Revoked = \{\ (\text{“Alice”}, K_{\text{alice}}), \ldots\ \}.$$

Any principal that relies on $K\$PKD$ but wants to avoid revoked bindings can simply use $(K\$PKD - K\$Revoked)$ instead of $K\$PKD$. Or, K could force others to consult the revocation list by issuing a certificate such as

$$K \textbf{ says } PKD \supseteq \{(\text{“Alice”}, K_{\text{alice}})\} - Revoked.$$

To show that a binding is in $(K\$PKD - K\$Revoked)$, you have to show that it is *not* in $K\$Revoked$. So we will need a new kind of certificate, indicating non-membership:

$$K \textbf{ says } (\text{“Bob”}, K_{\text{bob}}) \notin Revoked.$$

The Naor-Nissim scheme already described is one way of representing these certificates efficiently.

Unrestricted use of membership and non-membership certificates can lead to security loopholes. For example, the following scenario was possible in SDSI 1.1 [11]:

$$\begin{aligned}
&students = K_1\$students,\\
&school = \textbf{union}(teachers, admin, students),\\
&employees = school - students.
\end{aligned}$$

This defines two groups, *school* and *employees*, in terms of some other groups such as $K_1\$students$. In order to decide whether or not a principal K_2 is a member of *school* or *employees*, we require a document, signed by K_1, stating whether or not K_2 is a member of *students*: that is, a certificate K_1 **says** $K_2 \in students$, or K_1 **says** $K_2 \notin students$. Suppose an ‘error’ occurs and we are presented with *both* certificates. This might seem impossible, but the set of students is bound to change (students graduate) so an adversary could collect contradictory certificates over time and submit them together, perhaps attacking a gap in the timestamping procedures of the database. Since $K_2 \in students$, by the first definition we have $K_2 \in school$. And then since $K_2 \notin students$, we have $K_2 \in employees$—even though K_2 was never a teacher or administrator!

The problem here seems to be that we have conflicting positive and negative information about *students*. We will prohibit this in QCM by creating two syntactically distinct classes of names: positive names that can be used in membership certificates, and negative names that can be used in non-membership certificates. By enforcing such restrictions we can guarantee that inconsistencies like the example above can never occur.

4.7 Details of the implementation

QCM is implemented in Caml [4], a dialect of the language ML. We chose Caml based on our own background as ML programmers and because of very positive experiences with Caml as a language for writing distributed and network programs. In particular, our own active network implementation [14] and projects elsewhere such as Ensemble [7] and MMM [12] have shown that Caml can be used to build efficient distributed applications quickly. This success is supported by language features such as *strong typing* and *automatic memory management*. Another good candidate would have been Java, which also offers these features, but the Caml compiler and target code are more efficient than current Java implementations, and Caml has excellent support for language development.

We have implemented several variants of QCM, all of which run on platforms that support Caml; these include Windows 95 and NT, and a number of Unix variants. The primary implementation uses TCP sockets to send messages between QCM servers. Another variant runs in a single-machine mode, simulating distributed computing using threads. This variant is useful for prototyping, debugging, and simulating QCM computations conveniently. A third variant of QCM was built to run on top of the PLANet active network [10]. In this implementation QCM carries out network communication using PLAN [9] active packets. Mike Hicks has used this system to implement a security infrastructure for access control of PLANet switchlets. The examples in this paper run under each of our implementations, and we have developed a number of other small QCM programs to help us understand how well our query optimization works and to test how expressive the policy language can be. We have also implemented a graphical user interface for instigating and observing QCM computations, that runs with all variants of QCM. It produced all of the illustrations given in the next section.

Our implementation is about the same size as other verifiers. The IP, PLANet, and simulated variants of QCM combined take up about 9,000 lines of Caml; this includes approximately 2,000 lines for basic cryptographic algorithms (SHA, DSA, and key generation). The GUI adds another 2,500 lines of code. In comparison, the SDSI 2.0 distribution is about 13,000 lines of C and Perl, not including basic cryptographic algorithms or GUI support.

5 Observing QCM Computation

In this section we give a complete example of an actual QCM computation as viewed with our graphical user interface. The example is based on PICS [16, 15], the Platform for Internet Content Selection. PICS is a system for assigning ratings to web pages. These ratings come in the form of *labels* that identify the page, rating, and issuer of the label, and that can optionally be signed. To use PICS to filter objectionable web pages, a browser must first decide what issuers to rely on and what ratings are acceptable; this amounts to defining a policy [3, 6]. Second, the browser has to obtain labels. Labels can be embedded in pages, in which case the browser would simply get the page in question, and submit the labels to a verifier. It is also possible that the page might not have labels issued by someone the browser wants to rely on—after all, it is too much trouble for the page provider to coordinate with all possible issuers. QCM's policy-directed retrieval handles both cases nicely.

5.1 A PICS-like rating system in QCM

A QCM program expressing the policy of a browser is given in Table 2. The program is large because it contains keys. For readability, we will give an abbreviated version later, but it is instructive to consider the difficulties involved in programming with keys, including their effect on program size.

The first line of the program is a comment, and it says that the program is intended to run at saul:3335. QCM computation is carried out between a family of QCM processes running on various computers and acting as servers on ports, where they receive queries and send responses over TCP sockets. The QCM process of the browser will run on a computer named Saul, and will accept queries on port 3335. Strictly speaking, there is no need for the browser's QCM process to accept queries from the network, but this is convenient for our example.

The second line says that this is an online program: answers to queries will be signed online. The signing principal is given next, in a notation similar to that of SDSI; the principal contains both the public key, and the private key needed to produce signatures. The keys are DSA keys given in base 64 notation.

The main part of the program is given next, in a sequence of definitions that should look familiar: we use exactly the same syntax as in the rest of the paper, except that the principals are written out in full, and '$\in$' is not an ASCII symbol, so we use '`<-`' instead. The first definition relies on a principal with two parts: a server, saul:3336, and a public key. Let's abbreviate this principal as `K6`, so the definition is easier to parse:

```
Ratings = { x | ("alice",k) <- K6$PKD, x <- k$Ratings };
```

The policy says to rely on the PKD of `K6` to say what key `k` is associated with Alice, and then rely on `k` for ratings. There is a second definition, `OK`, that says that G-rated pages are OK to view.

Our example uses a handful of other QCM processes, which for simplicity were all run on a single computer (Saul) at different ports. However, all communication was carried out through the socket interface just as it would have been if the processes were on different computers. The programs of the processes are displayed in Table 3, where they are listed one after the other beginning with a comment line indicating which port they are listening on and a brief explanation of their function. For brevity, principals have been abbreviated in the table listing: `K3` is the principal of saul:3333, `K4` is the principal of saul:3334, and so on. In the actual QCM programs the principals are written out in full. They always include a public key, and also private keys and server addresses when appropriate.

Table 2: The policy of a browser written in QCM

```
# saul:3335 -- Policy of a browser

online

Principal(
 <PrivateKey="AxuvCtdEoj0VtGnBD2bMAIhX18Y=",
  PublicKey="P: 6e+1xVU3D5dHFrZo+Cd2YsR/CxvFVcDyAImn5nESz17GjBurur
3xrEVj7XV+wpS7N2XtZ0QPeNoUId1RRXAY2qDK8Cc7WyofsomqCzW5Sdz4d4sa45J2
ur/+pdiZtQEnfGoYd15mUPhShE5BtfaY0VrBzrfX7pqIPLhq476zobs=, Q: 2dfTn
Tvli0m4HzCh9rlIaKT2alk=, ALPHA: 5amSL0VSdMxCRphMFIqiFq+ZFlo5Q9gPVs
w2Fxm5HDCxCxXr3sjixg0pQ6ZcZe/ea0KGyjvE8xQuwfJZnn1GvSDAv0iyyDyoorHm
qHkzjM9+m7zetZvZ13F3zDnVKQR3MfgQpix6qeri9xsivuU4fE2ius0ZsrTwB1F49z
N2OMA=, Y: 1jO40fQJ5KhSk+wRwvfRypl4Y2Yk7XHNvYYwRcQyTD+e5dMz6eryYjD
qXNrVZUnV4cuDX65tRSBcTR3TI8l5Ort3NmnnGoflo6E/G6LDFvsAqv5JbmsNvehL7
nmbkHok3W/37XHxrAKcA5DgLgNQDzV7PejxKh0afjjABq9/xQM=">)

{

  Ratings =
    { x | ("alice",k) <- Principal(
                             <Server=[saul:3336],
                              PublicKey="P: 6e+1xVU3D5dHFrZo+Cd2YsR/
CxvFVcDyAImn5nESz17GjBurur3xrEVj7XV+wpS7N2XtZ0QPeNoUId1RRXAY2qDK8C
c7WyofsomqCzW5Sdz4d4sa45J2ur/+pdiZtQEnfGoYd15mUPhShE5BtfaY0VrBzrfX
7pqIPLhq476zobs=, Q: 2dfTnTvli0m4HzCh9rlIaKT2alk=, ALPHA: 0XmMrRFw
HdRY2i/ya7d1murBUkhJwc+H4lJTpZ7LJNKcOs3+HolntEwtWiO5B40wwSO2Pl5B7a
zZxm23zFXUJVXQATGEH13XsX+BEaxrIMj6Vh+dOw5B3286wjhAm/lgyglrSsl20BNq
NeW42zTMCoYQeFrs+vI4Z10Zx8yCMmY=, Y: D+4Mymn0id6KMo0gZjappc1Azd/qM
3LI2A8aaTERymYQe0bqULkLxX/SiYkGaLZNJ9LJ2Iy9EQGiG2NPEaC+nENHnYYDfSK
0XjaaeCOaulREkyOQVK5qvvXql3lYAbinnBMRrJx5nZoL2Te1VAW/KY5JNM9riUSp6
KdKKnT+dE4=">)$PKD,
          x <- k$Ratings };

  OK = { p | (p,"G") <- Ratings };

}
```

Table 3: QCM Programs

```
# saul:3333 -- A ratings database
offline
{ <Document = <Name = "Ratings",
               Includes = {("www.microsoft.com","R")}>,
   Signature = ..., Signer = K3>,
  <Document = <Name = "Ratings",
               Includes = {("www.yahoo.com","G")}>,
   Signature = ..., Signer = K3>,
  <Document = <Name = "Ratings",
               Includes = {("www.ietf.org","G")}>,
   Signature = ..., Signer = K3>
}

# saul:3334 -- A ratings database
online K4 {
  Ratings = {
    ("www.netscape.com","R"),
    ("www.nsa.gov","R")
  };
}

# saul:3335 -- Policy of a browser
online K5 {
  Ratings = { x | ("alice",k) <- K6$PKD,
                  x <- k$Ratings };
  OK = { p | (p,"G") <- Ratings };
}

# saul:3336 -- A public key directory
online K6 {
  local = { ("cindy", K7),
            ("doug", K6) };
  PKD = union(local, K7$PKD);
}

# saul:3337 -- A public key directory
online K7 {
  PKD = { ("alice", K3),
          ("bob", K5),
          ("alice", K4) };
}
```

Notice that we have included one offline program, for saul:3333. This simply consists of a collection of certificates, all signed by `K3`. Each certificate gives a single member of the set `K3$Ratings`. In previous sections of the paper we've used a more readable notation, for example, we would have written the last certificate as

$$K_3 \textbf{ says } \mathit{Ratings} \supseteq \{(\text{“www.ietf.org”}, \text{“G”})\}$$

or even

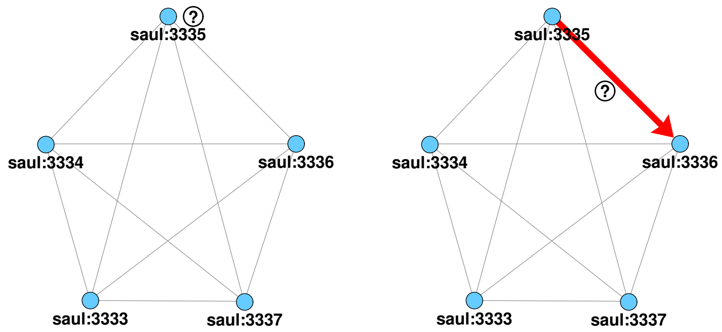
$$K_3 \textbf{ says } (\text{“www.ietf.org”}, \text{“G”}) \in \mathit{Ratings}.$$

The table gives the actual syntax, except that the signatures have been truncated for brevity.

5.2 Tracing QCM execution.

We have built an application, the QCM GUI, whose primary purpose is to instigate QCM computations and observe their actions. The GUI can submit queries to QCM just like any other application, and in addition, can cause any QCM node to send it activity reports. Over time the GUI gathers a trace of the computation of a distributed collection of nodes, which can be displayed, replayed, and even exported as a movie. The pictures shown in this section were generated by the GUI from a trace of an actual QCM computation carried out by our IP implementation of QCM.

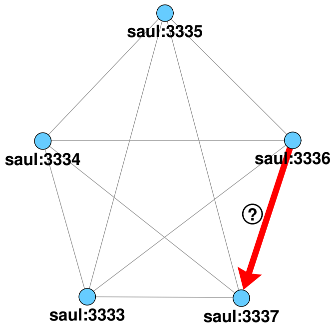
No computation takes place until a query is submitted, so we use the GUI to submit the query `OK` to saul:3335. According to Table 3, this should evaluate to the set of web pages that `K5` (the principal of the browser) considers to be ‘ok’. This was defined to be the set of web pages that a key of `"alice"` rates as having `"G"` content. We used the GUI to watch how QCM determines the keys of `"alice"` and the ratings associated with these keys. According to the program of saul:3335, keys for `"alice"` are drawn from `K6`, so we expect this principal to be queried first, and this is confirmed by the first two steps shown by the GUI:



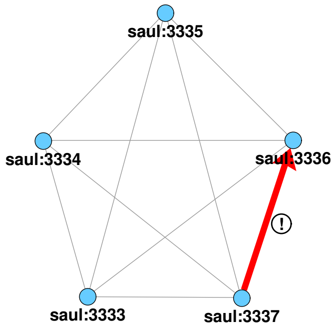
`OK;` `{k | ("alice",k) <- K6$PKD};`

Each picture shows the activity of the five QCM processes at a step of the computation. In the first picture, the question mark next to node saul:3335 indicates that the node has received a query, `OK`, printed under the picture. In the second picture, the arrow indicates a message sent from saul:3335 to saul:3336; again, the question mark indicates a query, printed below the picture. The query asks for the keys of `"alice"`, according to the PKD of `K6` (the principal at saul:3336).

Since `K6$PKD` is defined in terms of `K7$PKD`, we expect saul:3336 to exchange messages with saul:3337 (the server of `K7`), and this is exactly what happens:



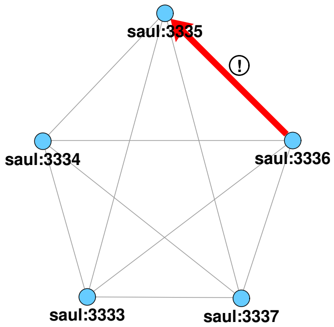

```
{k | ("alice",k) <- K7$PKD};
```

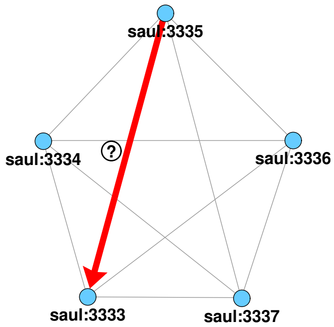



```
Signed(<Document =
         <QueryHash = "X5gs2eAn...",
          Response = {K3,K4}>,
        Signature = "R: TycLDa...">)
```

The response (indicated with an exclamation point) is a signed message. It has three parts: the response to the query itself (the keys `K3` and `K4`); a hash of the original query; and a signature. (We've truncated the hash and signature to save space.) Note that there is no `Signer` field, because we assume that saul:3336 expects `K7` to be the signer. The hash prevents a 'man in the middle' from using a previous response signed by `K7` as the response to this new query. The hash is calculated by converting the query into a canonical string form, which is then run through SHA; the signature is produced by running DSA on the canonical string of the `Document`. Both the hash and signature are checked by QCM at saul:3336.

Saul:3336 can now tell saul:3335 the keys of `"alice"`, and saul:3335 can begin collecting ratings:



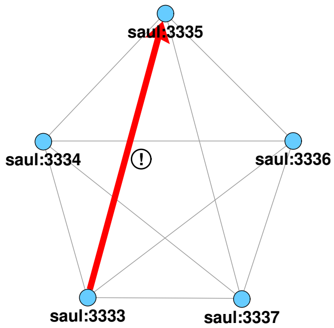



```
Signed(<Document =
          <QueryHash = "ZGW...",
           Response = {K3,K4}>,
         Signature = "R: K...">)
```

```
{p | (p,"G") <- K3$Ratings};
```

On the left is the response to saul:3335. Note that although saul:3336 has no keys to add to those it obtained from `K7`, the certificate signed by `K7` is not simply forwarded. This is because saul:3335 asked about `K6$PKD`, not `K7$PKD`, and expects a response signed by `K6`. Consequently, saul:3335 will have no way of knowing that the response was derived from `K7`.

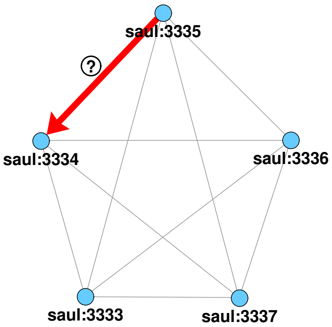
Above right, saul:3335 queries the first `"alice"` for the `"G"` rated pages, provoking the following response:

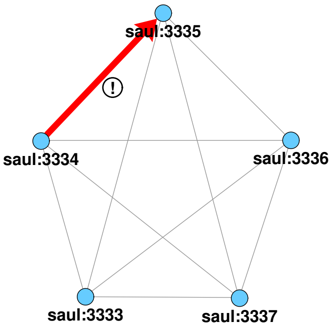



```
Certificates(<Document = <Name = "Ratings",
                          Includes = {("www.yahoo.com","G")}>,
              Signature = "R: dejgOAkQJfSUsoYQ...", Signer = K3>,
             <Document = <Name = "Ratings",
                          Includes = {("www.ietf.org","G")}>,
              Signature = "R: SkhPiA74jrkrnC1B...", Signer = K3>)
```

This is an offline response, i.e., a set of certificates that were signed offline by the principal `K3`. Notice that saul:3333 is smart enough to return only the subset of its certificates that are relevant to the query being asked (in Table 3, we can see that there is also a certificate for an `"R"` rated page). These certificates are different from the signed responses we have seen so far. First, they do not directly answer the query: the query asked for a set of pages, and the certificates speak of both pages and their ratings. This forces saul:3335 into some extra work, extracting the answer to its query from the certificates, as described in Section 4. Second, they are self-contained in that they have a `Signer` field, and an explicit `Name` instead of a `QueryHash` field. Self-contained certificates are also used for 'pushing' certificates at QCM.

Querying the second `"alice"` does not yield any new pages (none there are rated `"G"`):

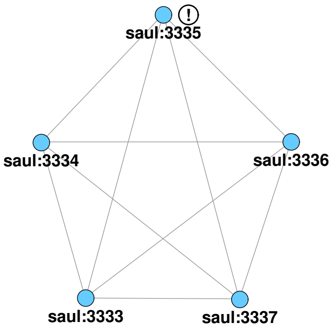





```
{p | (p,"G") <- K4$Ratings};
```

```
Signed(<Document = <QueryHash = ...,
                    Response = {}>,
        Signature = ...>)
```

In the final step, the responses are combined to answer the original question ('which web pages are ok?'):




```
{"www.yahoo.com","www.ietf.org"}
```

One further remark on the GUI is in order. Since the computation is distributed there is no guarantee, even when running all of the processes on the same machine as we did here, that the GUI will receive reports of events in the order in which they occured. When we set up this example, our runs sometimes showed events in the causal order that appears here in the paper and sometimes showed these events in other orders. Actual sequencing is ensured only for causally related events in a single node. This makes it more complicated to understand the QCM movie, of course; further work on the GUI and QCM would be necessary to aid better ordering. A simple approach to this problem is to use our threaded version of QCM, which runs in one process; when the GUI monitors this process, causally related events will be reported in the expected sequence.

5.3 Variations and Analysis.

We now consider some variations on the computation above and analyze a few design decisions.

Suppose one or more of the QCM servers does not respond. This would not be at all unusual in a distributed context since overloaded servers and machine outages could cause this behavior. Recall that QCM attempts a best effort response, so failures of this kind cause less information to be reported. For our example above, with query `OK` submitted to to saul:3335, the response would be the empty set, { }, if any of the servers saul:3333, saul:3336, saul:3337 were not available. However, if saul:3334 is not available, then the ultimate response will not be affected because it turns out that saul:3334 had no information about appropriately-rated pages. This approach has the advantage that QCM is tolerant of some lack of information, and the disadvantage that the information it returns must be viewed as partial in such cases.

QCM has been set up to make limited use of supplied certificates. For example, if we supplied a certificate from `K6` along with the original query, then queries to saul:3336 would have been short-circuited. If the certificate said that the keys of `"alice"` were `K3` and `K4`, then the computation would have proceeded as before, but without queries to saul:3336 and saul:3337. If, on the other hand, this certificate contained *different* information, then saul:3335 would rely on it and possibly obtain a different answer. For instance, suppose the GUI supplied a certificate signed by `K6` with no keys for `"alice"`. Such a certificate might have been obtained at an earlier time from saul:3336 when saul:3337 was unavailable. Then saul:3335 would have replied to the GUI with the empty set because it would assume that saul:3336 does not provide any keys for `"alice"`. This is unfortunate, but the alternative would be to have QCM query saul:3336 anyway, and this would defeat the point of supplying certificates in the first place. Applications must supply certificates only if they wish them to be used in place of a query to the principal that signed a response. This is safe because the worst consequence is that less information may be given.

QCM nodes can also be configured to run in a verify-only mode. If saul:3333, saul:3334, or saul:3337 were configured this way there would be no effect on the computation. If, however, saul:3335 or saul:3336 were configured in this way then the application would need to supply certificates in order to get a response. In verify-only mode, the verify will not automatically retrieve certificates, leaving that to the application. For instance if saul:3335 is in verify-only mode then the application would need to consult all of the other processes to obtain certificates and then submit them to saul:3335. If all of the processes are in verify-retrieve mode except saul:3336, then the application will need to obtain a certificate for the keys of `"alice"` signed by `K6` (not `K7`!) in order to get the desired information.

An important thing to observe about each of these computations is that the data/programs of the QCM servers themselves are never changed. After the query is answered, the information held at all of the nodes is the same as before, and another query will cause the same computation to take place. The situation is similar to that of web pages, where the page is viewed but not changed by a hit. QCM could cache information, but, as always, there is a problem with stale data. Our current implementation does not attempt to do caching; however, an application could cache certificates to speed responses. This is safe since QCM will check the validity ranges on the certificates.

6 Conclusions

We have described a system, QCM, that combines security policy verification with automatic policy-directed certificate retrieval. QCM is able to accept certificates supplied by applications, combine them with local certificates, and determine which remote certificates are needed. It is able to retrieve remote certificates on a best effort basis while maintaining data integrity. The supplied, local, and retrieved certificates can then be combined to determine policy compliance. The system is able to handle offline and online modes. It is able to deal with nodes that will retrieve certificates on behalf of a requester and those that do not. The system has been formally specified and implemented to run on a variety of platforms. We have explored many kinds of policies and retrieval techniques to show that the system is able to express policies simply and carry out retrieval in a reasonable way. We believe that these steps have shown that policy-directed certificate retrieval is feasible and desirable.

There are a few systems relevant to QCM and policy-directed certificate retrieval. DNSSEC [1], the secure extension of the Domain Name System proposed by the IETF, is one such. DNSSEC is similar to QCM in that it is a hierarchical, distributed database system for public data. The fact that DNS is such a widespread service is a good indication that QCM will be scalable to larger examples than we have shown here. We think that QCM has two advantages over DNSSEC: first, DNS has not been designed as a verifier, for example, it does not accept certificates 'pushed' at it along with queries; and second, DNS has been specifically targeted to the needs of the network infrastructure. It can be adapted to other applications, via the mechanism of text resource records, but this is not central to its design. QCM has been designed with application support as a first priority. A very recent example of a policy-directed system is PICSRules [8], a language proposed by the World Wide Web Consortium for expressing policies about web content filtering. PICSRules policies specify what PICS labels are required for viewing a web page, and can specify that labels can be obtained from remote servers ("label bureaus"). PICS labels are not yet signed, although this is planned.

It seems certain that certificates will be used by more and more applications in the coming years. Some of these applications will present challenges to automatic, policy-directed retrieval systems that we have not yet considered.

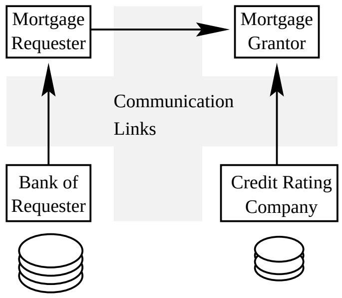


Figure 9: Certificates in a Mortgage Approval

Consider, for example, the documents involved in the pre-approval of a mortgage. Today, these documents are passed along by mail, fax, computer network, orally over the telephone, through personal contact, and so on. The authentication of documents generally relies on letterheads and security of communication channels like the telephone. What would it take to make this process more electronic and more *automatic*? It is important to note that certificate retrieval is a key element of the verification process. The person requesting the mortage and the one granting it are engaged in a mutual effort to create a proof: a proof that a mortgage should be pre-approved. To do this various "axioms" must be established, like how much money the requester has in the bank, and it must be proven that certain policies are respected, such as bank rules about the size of a loan compared to the value of a property. This proof entails the retrieval of a number of certificates. For instance, the requester may be asked to supply recent bank statements and permission for the grantor to check the requester's credit rating (see Figure 9). This process is reflective of many of the themes of this paper. Certificates must be 'pushed' to the grantor by the requester while the grantor uses information from the requester to carry out information retrieval from a credit rating company. However, this story also carries a number of complexities. For instance, the grantor may need to get a certificate (possibly just oral permission) before accessing the credit record of the requester. In an electronic version of this scenario such permission should probably be passed as a certificate and checked by the credit rating company before releasing information. This causes the retrieval problem to be intermingled with access control. Our system would need to be extended to deal automatically with this added complexity. However, it is likely that much of the process can indeed be automated.

Another aspect of the scenario of Figure 9 is the likely existence of large data repositories controlled by one or more of the principals. The bank and credit rating company almost certainly have large databases that are used to generate the certificates involved in this exchange. A system for automating this process will need to have a convenient interface to these systems. We believe that this is a strength of QCM, which was designed to make this interface easier, but there is still work to do if QCM is to match queries in its comprehension syntax to standards like SQL.

Acknowledgements We would like to acknowledge encouragement and assistance from Joan Feigenbaum, whose work on Trust Management inspired us to pursue this investigation. We received valuable assistance on the database aspects of the work from Rona Machlin, Arnaud Sahuguet, Dan Suciu, and Val Tannen. We also appreciated advice from Martin Abadi, Alex Aiken, Mike Hicks, Bob Harper, Luke Hornof, Peter Lee, and Jonathan Smith.

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] 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.

[3] Matt Blaze, Joan Feigenbaum, Paul Resnick, and Martin Strauss. Managing trust in an information-labeling system. *European Transactions on Telecommunications*, 1997.

[4] Caml home page. Http://pauillac.inria.fr/caml/index-eng.html.

[5] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. SPKI certificate theory. Internet Draft, March 1998.

[6] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. SPKI examples. Internet Draft, March 1998.

[7] Ensemble home page. Http://simon.cs.cornell.edu/Info/Projects/Ensemble.

[8] Christopher Evans, Clive D.W. Feather, Alex Hopmann, Martin Presler-Marshall, and Paul Resnick. PICSRules 1.1. Http://www.w3.org/TR/REC-PICSRules, 1997.

[9] Mike Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *Proceedings of the International Conference on Functional Programming Languages*. ACM, 1998. Http://www.cis.upenn.edu/˜switchware/papers/plan.ps.

[10] Mike Hicks, Jonathan T. Moore, Scott Alexander, Carl A. Gunter, and Scott Nettles. PLANet: An active internetwork. `www.cis.upenn.edu/~switchware/papers/planet.ps`, 1998.

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

[12] François Louaix. A web navigator with applets in Caml. In *Fifth WWW Conference*, 1996.

[13] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. In *7th USENIX Security Symposium*, 1998.

[14] SwitchWare Home Page. Http://www.cis.upenn.edu/˜switchware.

[15] Paul Resnick. Filtering information on the Internet. *Scientific American*, pages 106–108, March 1997.

[16] Paul Resnick and James Miller. PICS: Internet access controls without censorship. *Communications of the ACM*, 39(10):87–93, October 1996.

[17] W. Stallings. The PGP web of trust. *BYTE*, 1995.

[18] P. R. Zimmerman. *The Official PGP User's Guide*. MIT Press, 1995.