Authenticated Data Distribution using Query Certificate Managers

Carl A. Gunter, Trevor Jim, and Bow-Yaw Wang*
University of Pennsylvania

Abstract

QCM is a special-purpose programming language for authenticated data distribution on the Internet. QCM programs maintain a database distributed among the nodes of a loosely-coupled network and interact by exchanging digitally-signed database queries and responses. The database primitives of QCM naturally support essential security functions such as the definition, distribution, and use of access control lists and public key directories. QCM has a formal semantics based on structural operational semantics and I/O automata which may be used to prove correctness properties of QCM programs.

1 Introduction

We have designed a special-purpose programming language, *Query Certificate Manager (QCM)*, for the authenticated distribution of data in an untrusted network such as the Internet. Target applications range from basic network services, such as the Domain Name System (DNS), to high-level applications, such as certificate-based content filtering for web browsers. While general-purpose languages can be used to implement these functions, it is more convenient and significantly less error-prone to address them in a special-purpose language. By tuning the language to the problem domain, we can eliminate entire classes of errors while making programs easier to write, understand, and prove correct. Moreover, as we will illustrate in this paper, our special-purpose language can be given a precise semantics with clear correctness properties.

The central feature of QCM is the automatic management of *certificates* and *queries* to provide authenticated data distribution. In QCM, programmers do not write message-sending code to obtain remote data; the necessary messages are sent automatically by the query management system. To ensure authenticity, queries and replies are digitally-signed certificates; but programmers do not write code to verify signatures. This is done automatically by the certificate management system. Furthermore, QCM automatically checks the *consistency* of data contained in verified certificates. Each of these tasks—exchanging

*`http://www.cis.upenn.edu/~{gunter,tjim,bywang}`

messages, verifying signatures, and interpreting the contents of certificates—is complicated, error-prone, and should not be the responsibility of the typical programmer. Making them the responsibility of QCM leads to more robust programs. In particular, the correctness of these operations can be established once for QCM rather than for each program that requires authenticated data distribution. A relevant parallel is automatic memory management. It makes programs easier to write and understand by relieving the programmer of the responsibility of explicit allocation and deallocation. It eliminates dangling pointer bugs, a class of errors that are particularly hard to diagnose, and it makes programs easier to prove correct because the correctness of memory management depends only on the garbage collector.

QCM can be used either by itself, or in combination with a general-purpose language. Services such as the WHOIS database, DNS, and public key directories are simply distributed databases, and, as such, can be programmed entirely in QCM. Other services require more than just authenticated data distribution. For example, standards such as IPsec, the Secure Sockets Layer, and Secure HTTP are used to establish private and authenticated communication over the network. Here, QCM could be used to provide authentication, while communication and privacy could be implemented in another language. We have resisted making QCM a universal language. It has sufficient power for its task, and its programs are simpler and easier to understand as a result.

All of the services just mentioned are currently being designed or redesigned to accomodate authentication, on an ad hoc basis. This leads to duplicated effort and probably bugs that could be avoided by using QCM as a common basis for authentication. Moreover, even if all of these network-layer services were bug-free, they do not begin to address all of the authentication needs of higher-level applications. As a simple example, web browsers are now being refitted with certificate-based content filtering. The central problem here is the authenticated transmission of content ratings from rating services to the browser, typically via rating certificates attached to the web pages they rate. Network-layer authentication can ensure that a rating certificate is not corrupted during transmission, but does not help the application verify the signature of the certificate, or interpret its contents; it has no knowledge of rating certificates at all. QCM is appropriate for this problem, as well as lower-level problems.

Our work is about the authenticated distribution of data in general, and not about security-related data in particular. However, security infrastructures and trust management are areas in which authenticated data distribution is especially needed. This has influenced our choice of examples in this paper, which include the formation of access control lists (ACLs), certificate-based data filtering, and public key infrastructure.

There have been many other recent proposals for managing certificates, including PolicyMaker [2], SDSI [7], and SPKI [5]. These systems include only limited support for distributed database operations (to form groups, define access control lists, provide for queries about identity certificates, and so on). But they will need to interface with existing database systems anyway if they are to support some of their intended applications. By designing QCM as a database language, we directly support database integration, we avoid redundancy with technology already developed by the database community, and we can take advantage of new algorithms developed for databases on the web.

We make two contributions in this paper: (1) we show how to adapt new and existing techniques of distributed databases to provide authenticated data distribution over the Internet; and (2) we show how QCM's formal semantics provides basic correctness guarantees.

In addition to the usual challenges faced in distributed databases, a distributed family of QCM-mediated databases needs to provide support for manipulating certificates and maintaining the integrity of queries and answers. These new problems call for new ideas about algorithms and languages beyond those encompassed by existing work on query optimization and semantic data control for distributed databases. At the same time, we seek to build on existing database standards in order to provide for practical deployment.

The statement of correctness for QCM, like that of many other systems intended to provide security guarantees, is an interesting problem. There is a presumption that the semantics of the basic database constructs in QCM like union, join, and so on coincides with their usual meanings. However, one must take care in describing security guarantees: QCM is designed to facilitate *deferred trust* but cannot itself guarantee that trusted parties operate correctly. Nevertheless one can expect QCM to detect and deal sensibly with certain kinds of inconsistencies that may arise if data integrity fails.

This abstract is divided into five sections, the second of which provides an informal description of QCM programming with sample applications. A fuller definition is provided in Section 3. Section 4 states the correctness guarantees for QCM programs. Many design decisions are involved in the formulation of QCM messages and programs; we explore the design space and related work in a concluding section.

2 Query Certificate Managers

A QCM program defines a database and resides at a node in a computer network. Each program is associated with a unique *principal*; following standard practice [2, 7], a QCM principal is actually a public key. When a QCM program is queried, it will sign its reply with the secret key corresponding to its principal. The principal of a QCM program also serves as a name for the database defined by the program. By referring to remote principals, a QCM database can be defined in terms of remote QCM databases. Principals are thus used for both authentication and data distribution in QCM. The following QCM program illustrates these ideas.

$$\mathbf{friends} = \{p, q\} \cup p.\mathbf{friends}$$

This is a QCM definition of a database, **friends**, that includes the principals p, q, and the principals in p.**friends**. The database p.**friends** is maintained at a remote site (p's computer) and is distinct from the local **friends** (every principal maintains its own namespace). Sets of principals like **friends** are a particularly important application of QCM. They can be used as groups or access control lists to protect resources in the network (e.g., do not grant a request unless it is signed by the secret key of a principal in the group).

The program is quiescent until it receives a query addressed to its principal. For example, we might be asked whether a principal r is a member of **friends**. The QCM computation begins by comparing r to p and to q. If either matches, we can respond immediately. Otherwise, we must determine whether r is a member of p.**friends**.

One way to proceed involves an exchange of messages. QCM can send a message to p asking whether $r \in p$.**friends**, and wait for a reply signed by the secret key corresponding to p. Note that p.**friends** may itself be defined in terms of remote databases, so our query may in turn cause further queries to be sent out by p. The decision of what messages to exchange with whom, and what to do with responses, is made automatically by the QCM implementation. It is derived from the definition of **friends**, above, and is a simple application of existing database technology.

A second computation is possible if QCM is presented from the start with a certificate, signed by p, stating that $r \in p$.**friends**. In this case, QCM can deduce that $r \in$ **friends** without sending a message to p. Again, the use of the certificate is automatic; the programmer writes the same code for both cases, and does not mention messages or certificates. The ability to incorporate certificates into the database computation is one of the features that distinguish QCM from existing database languages.

We feel it is vital to support both of these computations. Using certificates to short-circuit messages, as in the second computation, places less burden on the maintainer of **friends** and can lead to a quicker response. It typically happens when r itself is making the initial request. In this case the certificate acts like a *capability* that r submits whenever it needs to access a resource protected by p.**friends**. On the other hand, p may not be willing to give r a certificate. The fact that r is a member of p.**friends** may be a closely guarded secret, to be shared only with the maintainer of **friends**. In this case the second computation is not possible, and the first computation must be used. A real-life example of this is the Secure Electronic Transactions protocol, where certificates may contain financial information that credit card companies will share with banks, but not with customers.

Notice that we did not give a program for p.**friends**. In fact, p.**friends** need not be implemented by a QCM program; for example, it could be a proprietary database. All that is required is that p understand the message format of QCM database queries and responses. Although complete databases can be programmed entirely in QCM, we think that the authenticated integration of existing databases into the network will be one of the most important services provided by QCM.

For example, VeriSign [12] issues *digital IDs*, certificates that associate a public key and an e-mail address with a name. There are several classes of IDs, and the class of an ID reflects how much trust should be put in the binding of key to name. For example, Class 3 IDs must be requested in person, with proper identification, while Class 1 IDs can be requested by e-mail. To integrate a database of digital IDs using QCM, we simply regard it as a relation with a particular schema, e.g.,

$$\mathit{VeriSign}.\mathbf{IDs}(\text{name}, \text{key}, \text{e-mail}, \text{class})$$

A user can define a public key directory as the union of some locally-known bindings, as well as trustworthy entries from VeriSign's database of IDs:

$$\begin{aligned}\mathbf{pkd} = \{&\langle \text{name} : \text{Bob}, \text{key} : K_{\text{Bob}}, \text{e-mail} : \text{bob@bob.com}\rangle\} \\ &\cup \{\langle \text{name}, \text{key}, \text{e-mail}\rangle \mid \langle \text{name}, \text{key}, \text{e-mail}, \text{class}\rangle \leftarrow \mathit{VeriSign}.\mathbf{IDs}, \text{class} = 3\}\end{aligned}$$

Informally, this equation defines **pkd** to be a table with columns labeled name, key, and e-mail. The expression $\{\langle \text{name} : \text{Bob}, \text{key} : K_{\text{Bob}}, \text{e-mail} : \text{bob@bob.com}\rangle\}$ defines an entry (row) in the table for Bob's key and e-mail address. The expression

$$\{\langle \text{name}, \text{key}, \text{e-mail}\rangle \mid \langle \text{name}, \text{key}, \text{e-mail}, \text{class}\rangle \leftarrow \mathit{VeriSign}.\mathbf{IDs}, \text{class} = 3\}$$

is a *set comprehension* that adds additional entries to **pkd** based on the contents of Verisign's database. Informally, each entry in Verisign's database of class 3 is used to produce an entry in **pkd**, by retaining the name, key, and e-mail fields, and discarding the class field.

The user can query the local **pkd** for keys by sending QCM an expression to evaluate. For example, to obtain Alice's keys (she may have more than one), the user asks the query

$$\{k \mid \langle \text{name} = \text{Alice}, \text{key} = k, \text{e-mail}\rangle \leftarrow \mathbf{pkd}\}.$$

By combining this query with the definition of **pkd**, and using algebraic reasoning, QCM derives a query to ask VeriSign for the appropriate certificates:

$$\{k \mid \langle \text{name} = \text{Alice}, \text{key} = k, \text{e-mail}, \text{class}\rangle \leftarrow \mathit{VeriSign}.\mathbf{IDs}, \text{class} = 3\}.$$

As a final example, we show how QCM can be used for certificate-based filtering of web pages. The idea is that a principal, r, maintains a database assigning ratings (G, PG, etc.) to web pages. This is simply a relation between the URL or cryptographic hash of a page and its rating:

$$r.\mathbf{ratings}(\text{page}, \text{rating})$$

Suppose a browser is configured to display only G-rated pages, that is, pages in the set

$$\mathbf{ok} = \{p \mid \langle \text{page} = p, \text{rating} = \text{G}\rangle \leftarrow r.\mathbf{ratings}\}.$$

Before displaying a page with hash h, the browser will use QCM to evaluate the query $\{p \mid p \leftarrow \mathbf{ok}, p = h\}$. It evaluates to the singleton set $\{h\}$ if h is G-rated;

otherwise, it evaluates to the empty set, and the browser will not display the page.

We could proceed as in the last example, and derive a query to send to r, but then we would have to make two requests for each page: one for the page and one for the rating. Instead, the page provider could store a *rating certificate*, issued by r, with the page itself. In QCM, such a certificate is written

$$r \ \textbf{says} \ r.\textbf{ratings} \supseteq \{\langle \text{rating} = \text{G}, \text{page} = h \rangle\}.$$

This is a document, signed by r, stating that h is G-rated; the actual signature is implicit, and is verified automatically by QCM. To filter pages, then, the browser obtains the page h together with the rating certificate, and submits the certificate to QCM along with the query $\{p \mid p \leftarrow \textbf{ok}, p = h\}$. Using the certificate and the definition of **ok**, QCM will evaluate the query to $\{h\}$, without sending a query to r.

Although bundling the page and the rating certificate together reduces network traffic, it requires the cooperation of the page provider and the rating service. This may be unrealistic: a page provider may not be willing to carry ratings unfavorable to its pages. But because QCM programs do not mention messages or certificates, this does not matter. If an appropriate certificate is provided with the page, it will be used; if not, QCM will ask the rating service for one. Other filtering proposals, such as PICS [10], also specify that both ways of obtaining ratings should be possible.

3 Formal Definitions

We now describe the syntax and operational semantics of QCM programs.

3.1 Syntax

The basis of QCM is the language of *set comprehensions* [3]. Our notation is summarized in Table 1.

We distinguish two kinds of variables: *basic variables* x, y, z will range over non-set values, and *set variables* S, T will range over sets. Different QCM programs may define and export the same set variables differently, hence we qualify set variables by principals: $p.S$ is considered a different variable than $q.S$. We drop the qualification p from $p.S$ when it can be recovered (p can refer to $p.S$ as simply S, but q must use $p.S$).

We use a, b to range over constants; constants include principals, and we use p, q, r to range over constants that happen to be principals. We use A, B to range over a countable set of *attributes* (field names). The *expressions* include constants, variables, labeled and unlabeled products, set union and intersection, and comprehensions. We write $\varnothing$ for the empty set. The attributes of a record $\langle A_1 = M_1, \ldots, A_n = M_n \rangle$ are required to be distinct. Attributes and basic variables may overlap, and we use $\langle \ldots, A, \ldots \rangle$ to abbreviate $\langle \ldots, A = A, \ldots \rangle$.

Basic variables	x, y, z
Set variables	S, T
Principals	p, q, r
Constants	a, b
Attributes	A, B
Expressions	$M, N ::= x \mid a \mid (M_1, \ldots, M_n) \mid \langle A_1 = M_1, \ldots, A_n = M_n \rangle \mid$ $p.S \mid x.S \mid \{M_1, \ldots, M_n\} \mid (M \cup N) \mid (M \cap N) \mid$ $\{\ M \mid G_1, \ldots, G_n\ \}$
Generators, guards	$G ::= P \leftarrow M \mid P = P'$
Patterns	$P ::= x \mid a \mid (P_1, \ldots, P_n) \mid \langle A_1 = P_1, \ldots, A_n = P_n \rangle$
Definitions	$D ::= p.S = M$
Queries	$Q ::= (p$ **asks** M **noting** $R_1, \ldots, R_n)$
Responses	$R ::= (p$ **says** $M \supseteq N)$
Basic values	$v ::= a \mid (v_1, \ldots, v_n) \mid \langle A_1 = v_1, \ldots, A_n = v_n \rangle$
Set values	$V ::= \{v_1, \ldots, v_n\}$
Messages	e
Jobs	$j ::= \langle M, p, N, \{R_1, \ldots, R_n\} \rangle$

Table 1: QCM syntax

We require that comprehensions have no free basic variables (a basic variable is free in $M = \{N \mid G_1, \ldots, G_n\}$ if it appears in M, and does not appear in a generator $P \leftarrow p.S$ in N).

A (standard) type system for QCM is given in Table 2; it rules out expressions like the selection of a field from a relation that does not possess the field in question. We assume that every constant has a known unique, closed basic type. We use s, t to range over *type variables*, and σ, τ to range over *basic types*. Basic types include datatypes, products, and labeled products. If σ is a basic type, then $\{\sigma\}$ is a *set type*. A *context* Γ associates basic types with basic variables, and set types with set variables. A *basic context* has only basic variables in its domain, and similarly, a *set context* has only set variables in its domain.

A *definition* has the form $p.S = M$. A *QCM program for principal* p consists of a set context Γ_p (giving types to set variables not qualified by p) and a sequence $p.S_1 = M_1, \ldots, p.S = M_n$ of definitions. We require that the definitions be typable under Γ_p. The secret key corresponding to p is implicitly available to the program, and is used to sign outgoing messages. We assume that if a principal q appears in one of p's definitions, then p knows how to send messages to q (e.g., p has q's network address). Note that two principals need not agree on the type of a given set variable; contexts represent local knowledge and need not be consistent from node to node. If Γ_p and Γ_q are not consistent, dynamic errors can occur when p and q communicate.

QCM programs interact by exchanging database queries and responses. A *query*, Q, is a certificate of the form $(p$ **asks** M **noting** $R_1, \ldots, R_n)$. The

Type variables	s, t
Basic types	$\sigma, \tau ::= \mathbf{int} \mid \mathbf{prn} \mid \cdots \mid t \mid (\tau_1, \cdots, \tau_n) \mid \langle \ell_1 : \tau_1, \ldots, \ell_n : \tau_n \rangle$
Set types	$\{\sigma\}$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : \sigma \mid \Gamma, p.S : \{\sigma\}$

$$\Gamma \vdash x : \Gamma(x)$$

$$\Gamma \vdash c : \tau \quad \text{(where } \tau \text{ is the type of } c\text{)}$$

$$\frac{\Gamma \vdash M_1 : \sigma_1,\ \ldots,\ \Gamma \vdash M_n : \sigma_n}{\Gamma \vdash (M_1, \ldots, M_n) : (\sigma_1, \cdots, \sigma_n)}$$

$$\frac{\Gamma \vdash M_1 : \sigma_1,\ \ldots,\ \Gamma \vdash M_n : \sigma_n}{\Gamma \vdash \langle \ell_1 = M_1, \ldots, \ell_n = M_n \rangle : \langle \ell_1 : \sigma_1, \ldots, \ell_n : \sigma_n \rangle}$$

$$\Gamma \vdash \varnothing : \{\tau\} \quad \text{(for any } \tau\text{)}$$

$$\Gamma \vdash p.S : \Gamma(p.S)$$

$$\frac{\Gamma \vdash x : \mathbf{prn}}{\Gamma \vdash x.S : \Gamma(x.S)}$$

$$\frac{\Gamma \vdash M : \{\sigma\},\ \Gamma \vdash N : \{\sigma\}}{\Gamma \vdash M \cup N : \{\sigma\}}$$

$$\frac{\Gamma \vdash M : \{\sigma\},\ \Gamma \vdash N : \{\sigma\}}{\Gamma \vdash M \cap N : \{\sigma\}}$$

$$\frac{\Gamma, \Gamma' \vdash G_1,\ \ldots,\ \Gamma, \Gamma' \vdash G_n,\ \Gamma, \Gamma' \vdash M : \sigma}{\Gamma \vdash \{\ M \mid G_1, \ldots, G_n\ \} : \{\sigma\}} \quad (\Gamma' \text{ is a basic context})$$

$$\frac{\Gamma \vdash P_1 : \sigma,\ \Gamma \vdash P_2 : \sigma}{\Gamma \vdash P_1 = P_2}$$

$$\frac{\Gamma \vdash M : \{\sigma\},\ \Gamma \vdash P : \sigma}{\Gamma \vdash P \leftarrow M}$$

$$\frac{\Gamma \vdash M : \{\sigma\},\ \Gamma, p.S{:}\{\sigma\} \vdash D_1, \ldots, D_n}{\Gamma \vdash p.S = M, D_1, \ldots, D_n}$$

Table 2: The type system of QCM

expected *response* is a certificate, R, of the form (q **says** $M \supseteq N$). That is, q returns an *approximation* N to the query M. Often, it will be the case that in fact $N = M$; however, it will be technically convenient to work with approximations, as we discuss below.

A query certificate (p **asks** M **noting** $R_1, \ldots, R_n$) implicitly includes a valid signature of p on M and $R_1, \ldots, R_n$; and response certificates similarly have implicit, valid signatures. We never refer to a certificate R or Q unless its signature has been verified. However, a node may receive messages that are not well typed, or that have invalid signatures. Therefore we introduce *messages*, e, which are the basic unit of transmitted data. Every QCM node p has available a state function $unmarshal_p$ which converts messages to certificates, performing signature checking and returning **Error** if any problem is found. For example, if $unmarshal_p(e) = Q = (p$ **asks** M **noting** $R_1, \ldots, R_n)$, it is guaranteed that the implicit signature of Q is valid, that M is well typed under Γ_p, and that all of the certificates $R_1, \ldots, R_n$ are also well-formed. Every QCM node p also has a function $marshal_p$ that converts certificates to messages.

3.2 Semantics

Conceptually, the semantics of query evaluation can be divided into two parts: the semantics of message exchange in a collection of nodes, and the semantics of computation at a single node (including deciding what messages to send). We model message exchange using automata theory, including an explicit formulation of the network itself as an automaton. We use standard techniques from database query evaluation and the operational semantics of programming languages to describe the evaluation of a QCM program on a single node. This semantics is not included here because we prefer to describe instead how supplied certificates can be integrated into local information to determine what further queries must be made; for this we use some new techniques that require explanation, including soundness and completeness criteria.

Inter-node Computation. We use I/O automata [9] to describe how nodes exchange messages in a network. I/O automata are essentially process algebras with state; they are a well-established formalism for the specification of distributed systems. Table 3 gives an I/O automaton that specifies the message-sending behavior of a node controlled by a QCM program. The automaton has three state variables: a set of *jobs* for queries remaining to be answered; a set of inclusions representing its current accepted facts about local and non-local relations; and a set of expressions that the node has asked other nodes to evaluate. The job for a query (q **asks** M **noting** $R_1, \ldots, R_n$) has the form $\langle N, q, M, \{R_1, \ldots, R_n\}\rangle$, where the expression N is the current approximation for M calculated by the node. The approximation will be refined step by step as the node consults local information, sends out queries, and collates responses. Information gathered from responses is checked for acceptability in a sense we will define in the section below on correctness, and, if acceptable, is added to

State

pending$_p$, a set of jobs, initially empty;
  jobs of the queries remaining to be answered by p

accepted$_p$, a set of inclusions, initially containing p's definitions;
  the set of inclusions believed by p

requested$_p$, a set of Γ_p expressions, initially empty;
  the set of all queries ever asked by p of other principals

Actions

Input $\mathit{receive}_p(e)$
Pre: $\mathit{unmarshal}_p(e) = R$
Eff: $\mathit{accepted}_p := \mathit{accepted}_p \cup \mathit{acceptable}_p(R)$

Input $\mathit{receive}_p(e)$
Pre: $\mathit{unmarshal}_p(e) = (q \text{ } \mathbf{asks} \text{ } M \text{ } \mathbf{noting} \text{ } R_1, \ldots, R_n)$
  $j = \mathbf{InitJob}(q \text{ } \mathbf{asks} \text{ } M \text{ } \mathbf{noting} \text{ } R_1, \ldots, R_n)$
Eff: $\mathit{pending}_p := \mathit{pending}_p \cup \{j\}$
  $\mathit{accepted}_p := \mathit{accepted}_p \cup \mathit{acceptable}_p(R_1, \ldots, R_n)$

Input $\mathit{receive}_p(e)$
Pre: $\mathit{unmarshal}_p(e) = \mathbf{Error}$
Eff: none (p discards the message)

Output $\mathit{send}_{p,q}(e)$
Pre: $j \in \mathit{pending}_p$
  $\mathit{step}_p(j) = \mathbf{Ask}(Y, q, e)$
Eff: $\mathit{requested}_p := \mathit{requested}_p \cup \{Y\}$

Output $\mathit{send}_{p,q}(e)$
Pre: $j \in \mathit{pending}_p$
  $\mathit{step}_p(j) = \mathbf{Ans}(q, e)$
Eff: $\mathit{pending}_p := \mathit{pending}_p - \{j\}$

Internal $\mathit{compute}_p$
Pre: $j \in \mathit{pending}_p$
  $\mathit{step}_p(j) = \mathbf{Job}(j')$
Eff: $\mathit{pending}_p := \mathit{pending}_p - \{j\} \cup \{j'\}$

Table 3: An I/O automaton for the QCM program of principal p

State
inbox$_p$ for each principal p, a set of messages, initially empty;
the messages sent to p but not yet read

Actions

Input $send_{p,q}(e)$
Eff: $inbox_q := inbox_q \cup \{e\}$

Output $receive_p(e)$
Pre: $e \in inbox_p$
Eff: $inbox_p := inbox_p - \{e\}$

Table 4: An I/O automaton for a store-and-forward network

a acceptance set that supplements the local data. A node can have many jobs queued at a time; evaluation proceeds in a nondeterministic, interleaved fashion.

The automaton moves from state to state by taking one of three different kinds of actions.

- The input action $receive_p(e)$ is taken when p reads a message e from the network.
- The output action $send_{p,q}(e)$ is taken when p asks the network to forward a message e to q.
- The internal action $compute_p$ is taken when p performs local computation that does not involve communication with the network.

Transitions are specified in a precondition/side-effect style. For example, the first clause for the action $receive_p(e)$ says that if the incoming message, e, is a response, R, then R should be examined and added to the local acceptance set if it is acceptable. The second clause says that if e is a query Q, then a job for Q should be added to the set of pending jobs, and any acceptable certificates supplied by Q should be added to the local acceptance set. The third clause says that if e is not a valid query or response, then there is no effect on the state—the message is effectively discarded. The remaining clauses describe how the node evaluates queries, using a state function, $step_p$, that defines intra-node computation.

The important thing to note here is that p does not 'accept' every certificate that is sent to it. We will study this more fully in the section below on correctness and defer the exact definition of the relation $acceptable_p$ to there. While p may *hope* to get good answers to its queries, whether p *does* get them is partly up to the unreliable network over which p communicates. We assume only that the signatures are true: if a message has p's signature on it, then it was in fact signed by p. If this assumption is violated because someone has learned how to sign as p, then our system will be misled. A simple automaton describing the behavior of a network is given in Table 4. Notice that the actions of the network automaton are the opposite of the QCM automata: from the network's point of view, $receive_p$ is an output action, and $send_{p,q}$ is an input action. When the network automaton and a collection of QCM automata are composed, the automata synchronize on their same-named actions. For example, in the composed system, a QCM automaton for principal p can take a $receive_p$ action only (and exactly) when the network simultaneously takes a $receive_p$ action. Two distinct QCM automata never share actions, so they never synchronize with each other; QCM programs can only communicate via the network, not directly.

Our network accepts messages directed to principals and forwards them nondeterministically. For simplicity we permit out-of-order messages, and there is no bound on the time it takes for a message to be delivered, but it is easy to design automata for more sophisticated networks. It is also possible to design a network automaton that acts as an adversary, for example by: intercepting and examining messages; misdirecting, replaying, or dropping messages; or forging messages. Such a network can be used in a study of the security properties of the system [8, 11].

Intra-node Computation. At each node, a complex calculation is needed to determine how to answer incoming queries and use incoming responses. The formal description of expression evaluation is in the full paper using a structural operational semantics. Instead of presenting details of the semantics, we focus on the most novel part of the problem which is how supplied certificates are used. Extensive research on efficient distributed query evaluation is available [13], but our work has focused primarily on the treatment of supplied certificates since this technology is comparatively new.

How should we evaluate a query M in light of an inclusion $N \supseteq V$ provided by some certificate? In the best case, M and N are syntactically identical, and we can immediately return V as the result. However, syntactic equality will not help us evaluate a query $\{x \mid (x, x, z) \leftarrow S\}$ when given the inclusion $\{x \mid (x, y, z) \leftarrow S, x = z, y = z\} \supseteq V$. We want to do better: in this example, we could return V as an approximation to the query. In fact, we have an algorithm that we can prove does as well as possible. We will describe the algorithm by example.

Suppose S is a relation with attributes A, B, and C, and we are given the inclusion

$$\{\langle A = x\rangle \mid \langle A = x, B = x, C = 3\rangle \leftarrow S\} \supseteq V.$$

Now, V is a table with a single column, A. We know that each row in V is derived from a row in S by simply discarding the B and C columns. Although these columns are not present in V, note that the inclusion provides us with enough information to reconstruct a large part of S using just V: it tells us that every entry in V was derived from an entry in S with a C value of 3 and a B value equal to the A value. Therefore, the following inclusion holds:

$$S \supseteq \{\langle A = x, B = x, C = 3\rangle \mid \langle A = x\rangle \leftarrow V\}.$$

In other words, we simply duplicate the A column from V to get the B column, and we assign 3 to the C column of every row. Our algorithm uses these ideas to take an inclusion, $M \supseteq V$, and recover as much information as possible about relation symbols appearing in M using just the table V. It has the following input/output behavior.

Input: An expression M not containing union ($\cup$); and a set variable S appearing in M.

Output: An expression M_S over a single set variable, S_V; M_S is a "best approximation" to S when $M \supseteq V = S_V$.

To give a more formal statement of the soundness and completeness of our algorithm, we need the following concepts. A database *instance* over a context Γ is a function $\mathcal{I}$ mapping each set variable $p.S$ in Γ to a set $\mathcal{I}(p.S)$ with type $\Gamma(p.S)$. Instances can be extended to map general expressions (not just set variables) to sets in the obvious way. We say that an instance $\mathcal{I}$ *satisfies* an inclusion $M \supseteq N$ if $\mathcal{I}(M)$ is a superset of $\mathcal{I}(N)$. A standard model can then be defined as follows.

Definition: If Γ is a context and I is a set of inclusions well typed under Γ, then we write $\Gamma, I \models M \supseteq N$ iff for all Γ instances $\mathcal{I}$, whenever $\mathcal{I}$ satisfies every inclusion in I, $\mathcal{I}$ also satisfies the inclusion $M \supseteq N$.

Then soundness and completeness can be stated formally as follows.

Theorem 1 *Let M be an expression not including union, and let S be a set variable appearing in M.*

1. $\Gamma, M \supseteq V = S_V \models S \supseteq M_S$.
2. *If L is any expression whose only set variable is S_V, and* $\Gamma, M \supseteq V = S_V \models S \supseteq L$, *then* $\Gamma, M \supseteq V = S_V \models M_S \supseteq L$.

The first part of the theorem says that M_S is an approximation to S, and the second part says that it contains any other approximation of S.

4 Correctness

As we mentioned in the introduction, there is some subtlety involved in formulating correctness properties for QCM. Recall the example QCM program:

$$\mathbf{friends} = \{p, q\} \cup p.\mathbf{friends}.$$

We expect that if this program is asked whether p is a friend, it should answer 'yes'. In particular, the database operations in the expression should have their expected meanings. If p considers r to be a friend, and *provides a certificate saying this*, then the program above should answer 'yes' when asked if r is a friend. If, on the other hand, a certificate asserting that r is in $p.$**friends** is received, but it is not signed by p, then it seems reasonable to ignore it. Finally, if p does certify that r is a friend, but somehow r is not 'really' a friend because he or she doesn't act like a friend, then this may be a problem with delegation to p and not something that QCM can be expected to determine. Drawing the line between what QCM should be able to detect and what must be viewed as a programming error, a trust management mistake, or a semantic control problem at another level, is the challenge we wish to address.

Correctness can first of all be judged in relation to the standard model of relational databases (described in the previous section).

Theorem 2 *Every reply sent by a QCM node is sound with respect to the acceptance set of the node: if p issues a certificate,* $(p \textbf{ says } M \supseteq N)$*, then* $\Gamma_p, accepted_p \models M \supseteq N$.

Note that soundness is cheap if the inclusions $accepted_p$ that we rely on are inconsistent (i.e., they have no model). Our definition implies that if $accepted_p$ is inconsistent, then $\Gamma_p, accepted_p \models M \supseteq N$ for *any* well typed $M \supseteq N$. So, while our semantics satisfies the soundness Theorem above, the system is trivialized if we rely on inconsistent data. We must take care not to believe everything we are told.

In fact, the following example, based on SDSI 1.1, shows that carelessness in this matter can lead to a security breach.[1] We define two groups, **school** and **employees**:

$$\begin{array}{l}\textbf{school} = \textbf{teachers} \cup \textbf{admin} \cup \textbf{students},\\ \textbf{employees} = \textbf{school} - \textbf{students}.\end{array}$$

Suppose **students** is maintained by a remote principal, p. In order to decide whether or not a principal q is a member of **school** or **employees**, we require a document, signed by p, stating whether or not q is a member of **students**: that is, a certificate $(p \textbf{ says } q \in \textbf{students})$, or $(p \textbf{ says } q \notin \textbf{students})$. 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 $q \in \textbf{students}$, by the first definition we have $q \in \textbf{school}$. And then since $q \notin \textbf{students}$, we have $q \in \textbf{employees}$—even though q was never a teacher or administrator!

Although one might hope that such inconsistencies never arise, the nature of security research forces us to confront the issue. What will the behavior of the system be when supplied with such certificates? The problem is not that the system cannot proceed, but that it might proceed in a number of ways.

- The system might take a conservative strategy and simply refuse access.
- The system might choose one of the certificates to use (like the first one it received).
- The system might discard both certificates and contact p directly to obtain a fresh certificate.

[1] This problem does not arise in QCM because QCM does not include negation. Negation has also been dropped in SDSI 2.0.

- The system might not notice the contradiction, and proceed unpredictably.

Without a well-defined semantics, different implementations might proceed in different ways, and the same implementation might proceed differently at different times. Any notion of correctness would be impossible. For QCM, we decided to see whether it is possible to detect all such inconsistencies. Experience with an implementation will be useful in determining what can be done with acceptable performance.

In general it is not easy to know what to believe. Common sense tells us not to believe a certificate (p **says** $q.S \supseteq N$), since p does not control $q.S$. But this is not enough: p can make well typed but nonsensical statements about its own relation, for example, (p **says** $\{x \mid x \leftarrow p.S, x = 1\} \supseteq \{2\}$. Furthermore, p can assert an inclusion that is satisfiable by itself, but contradicts an assertion we may have already obtained from another source, or even p itself. For example, the following assertions are separately consistent, but cannot simultaneously hold.

$$\begin{array}{ll} p \textbf{ says } \{x \mid x \leftarrow p.S, x = 1\} \supseteq p.S & (p.S \text{ must be } \varnothing \text{ or } \{1\}) \\ p \textbf{ says } p.S \supseteq \{2\} & (\text{contradiction}) \end{array}$$

To address this problem we define an acceptance criterion for certificates.

Definition: A certificate (p **says** $M \supseteq N$) is *acceptable* if it satisfies the following conditions:

1. Every relation symbol in M is qualified by p.
2. N is a value.
3. $M \supseteq N$ is consistent.

(It is not necessary to test that $M \supseteq N$ is well typed, since otherwise they could not appear in one of our certificates.) The first two conditions are clearly decidable, and algorithms for deciding the final condition are well known [1].

Our acceptance criterion guarantees the following invariant.

Theorem 3 *Evaluation never produces an inconsistent $accepted_p$ set.*

The key step in the proof of the invariant follows from this next Lemma, which explains our restrictions on the contents of certificates:

Lemma 4 *Let I and I' be sets of inclusions of the form $M \supseteq V$. If I and I' are consistent under Γ, then $I \cup I'$ is consistent under Γ.*

The Lemma fails if inclusions are replaced by equalities. Hence our use of inclusions makes it easier to keep $accepted_p$ consistent.

5 Conclusions

We have shown how a special-purpose system for authenticated data distribution can be designed to support a variety of applications, how such a system can be precisely specified, and how guarantees about the system can be proved. The proposed system lacks some desirable features, such as negation and the handling of looping references. The formal analysis could also be extended to include a more direct statement of QCM security against traditional communication attacks, like replay or 'man-in-the-middle'. We briefly comment on each of these issues and provide some hint of how we think QCM can be implemented.

If a system of QCM nodes grows linearly, with each new node being defined in terms of previously-existing nodes, then there will be no recursive loops that could cause infinite exchanges of messages. However, there is a danger of such loops being introduced when changes occur to programs at nodes already referenced by other nodes. Our current system views this as a programming and maintenance issue, to be avoided by writing or altering QCM programs so that this does not happen. A more automatic approach might be possible using some form of loop detection, perhaps relying on ideas from distributed Datalog evaluation [6]. We will consider this in future work.

The *revocation list* is a common security paradigm that QCM does not support, because we prohibit negation. For example, a revocation list might specify credit card accounts that have been compromised, and to which privileges should be denied. Negation would complicate evaluation, and consistency checking for expressions including negation is undecidable. We are searching for tractable restrictions of negation sufficient to handle revocation lists.

There is much more we could do with our formal analysis and we view the results in this paper as a first step. Network insecurity is mainly modelled here as unreliable nodes that might send inconsistent or ill-formed information. Our theorems did not assume that the information was sent from a node running a QCM program, so anything is possible. This implies that attacks like replay and 'man-in-the-middle' cannot affect these invariants; nevertheless, it would be preferable to model this using the network automaton. Stronger theorems might be obtained if assumptions are made of the network or sets of nodes on it, such as assuming that there is a subnetwork of QCM nodes (where message handling must respect the QCM semantics). We are hoping to explore the possibility of a result we think of as analogous to the Kahn-MacQueen Theorem for dataflow networks; such a result might says that a family of QCM nodes on a secure network (e.g., an intranetwork) can be viewed as a single QCM program.

We have worked on the implementation of a QCM interpreter in Java, using Java RMI to model QCM messages as remote method invokations and JDBC to mediate between QCM and relational databases. Progress on this effort will be reported in a future paper.

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.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[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] Peter Buneman, Leonid Libkin, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[4] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS-97*, May 1997.

[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] Allen Van Gelder. A message passing framework for logical query evaluation. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 155–165, Washington, D.C., 28–30 May 1986. *SIGMOD Record* 15(2), June 1986.

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

[8] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems, Second International Workshop, TACAS '96*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, March 1996.

[9] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.

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

[11] Steve Schneider. Security properties and CSP. In *Proceedings of the 17th Symposium on Security and Privacy*, pages 174–187. IEEE Computer Society Press, 1996.

[12] VeriSign, Inc. home page: www.verisign.com.

[13] Clement T. Yu and C. C. Chang. Distributed query processing. *ACM Computing Surveys*, 16(4):399–433, December 1984.