Design of an Application-Level Security Infrastructure

Carl A. Gunter and Trevor Jim*
University of Pennsylvania

Abstract

We propose a security infrastructure based on *authenticated data distribution*, implemented via the automatic management of queries and certificates. This approach is appropriate for an infrastructure to be used by application programmers, because they are not experts in cryptographic algorithms or security protocols. Our query certificate managers hide the use of cryptography and message sending from the programmer, and hence prevent programmer errors that could lead to security failures. The system has a formal semantics and correctness guarantees.

1 Introduction

Conventional wisdom says that security infrastructures should include primitives such as encryption and decryption, signing and signature verification, and message sending and receiving. This may be appropriate for security protocol designers or operating systems programmers, but we believe it is not appropriate for application programmers. When the resources, needs, and abilities of application programmers are taken into account, we arrive at a very different infrastructure.

Our proposed infrastructure is strongly influenced by ideas from programming language research. In our view, security protocols are simply programs, and security breaches are the result of program bugs. Security can thus be enhanced by incorporating techniques from programming languages that eliminate entire classes of bugs from programs. For example, *automatic memory management* is a language feature that relieves programmers from explicitly allocating and deallocating data: allocation and deallocation instructions are inserted by the language implementation. Not only does this make programs easier to write and understand, it also eliminates dangling pointer bugs. Such bugs are common in languages with explicit memory management, and can lead to security failures; Rivest reports that they were the

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

largest class of bugs encountered in implementing the security infrastructure SDSI [personal communication].

If memory management is hard to get right, cryptography and protocol design are even harder. Many cryptographic protocols designed by experts have turned out to be flawed, and application programmers are not experts in cryptographic algorithms, security protocols, and formal methods. Even when provided with formally verified, bug-free cryptography and protocols, they still build insecure systems [2, 3]. Taking automatic memory management as a guide, we seek to eliminate security bugs by hiding the use of cryptography, protocols, and message exchange from the programmer. Instead, the basic primitive we provide is *authenticated data distribution*, and we implement it by the automatic management of *queries* and *certificates*. Hence, the system is called the *Query Certificate Manager*, or *QCM*.

Common examples of authenticated data distribution include obtaining public keys or access control information from remote servers. These examples can be implemented in more than one way; for example: (1) by sending queries and responses over an authenticated channel between client and server; or (2) by means of certificates signed by the remote server and submitted to the client. The important thing to note is that the application programmer does not care *how* the remote information is obtained; he only cares that the *correct* information is obtained. Since the means of obtaining the data—cryptography and message exchange in (1), and careful examination of signatures in (2)—are complicated and error-prone, they should not be the responsibility of the typical programmer.

In our system, 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. And if certificates from remote servers are submitted to the system, messages to the remote servers are short-circuited, again automatically.

A second way in which we have been inspired by programming languages is in our use of formal methods. Like many programming languages, our system has a formal semantics, based on automata theory and a model of sets and relations. Having a semantics has proven invaluable in *designing* QCM, and places no burden on the *users* of QCM. For users, the semantics insures that all programs written in QCM have a sensible meaning: they define sets and relations. Thus, programmers do not need to understand the complicated semantics of message exchange and signature verification; they are justified in thinking of their programs as defining sets. This is in sharp contrast with security infrastructures based on message exchange and cryptography, where the users of the infrastructure typically need to use formal methods to have any assurance of security in their protocols.

The semantics has helped our design in several ways. First, a semantics is necessary in order to state and prove properties of the system, and so the semantics is fundamental to the correctness of our system. Second, in stating our correctness properties, we discovered some subtleties involving the *contents* of certificates, which could never be caught by a formal proof of correctness for the cryptographic protocols; see §3. This led us to an automatic analysis, *consistency checking*, that catches certain bugs that could lead to security failures, and that are likely to be overlooked by the application programmer.

Notice that our correctness properties are proved once for the QCM system. They then apply to any program written in QCM; no further correctness proof is needed. Essentially, we implement a QCM program by a protocol of message exchange and signature verification, and we have proven the correctness of a large *class* of protocols, namely, the protocols corresponding to all QCM programs.

There have been many other recent proposals for managing certificates, including PolicyMaker [4], 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 we think that application programmers will need to interface with existing database systems—for example, databases of customers—in order to support some of their intended applications. Therefore, we have designed QCM as a database language, and we directly support database integration. This avoids redundancy with technology already developed by the database community, and we can take advantage of new algorithms developed for databases on the web.

The remainder of the paper presents an informal overview of QCM, omitting most technical details. A complete definition can be found in a separate paper [6]. In §2 we give some examples of QCM programming. In §3 we discuss our semantics and correctness guarantees, and we conclude in §4.

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*; principals are traditionally the entities that take part in protocols by signing and exchanging messages. Conceptually, the principal of a QCM program accepts queries from other principals, supplies the queries as input to the QCM program, and returns the result after signing it. Following standard practice [4, 7], a QCM principal is actually a public key. A principal signs documents using its associated secret key, and the signature can be verified using only the signed document and the principal itself.

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. Our first example, *content filtering*, illustrates these ideas.

Content filtering prevents web pages with objectionable content from being displayed on a web browser. We show how QCM can implement content filtering based on page ratings provided by rating services. There are four principals: a page server, s; two rating services, $r1$ and $r2$; and a browser, b. The rating services maintain databases associating a rating (G, PG, etc.) with the cryptographic hash of a page. Conceptually, these databases are tables:

$r1$.**ratings**

hash	rating
$h1$	G
$h2$	PG
$h3$	R
$h4$	PG
⋮	

$r2$.**ratings**

hash	rating
$h1$	G
$h2$	G
$h4$	PG
$h5$	PG
⋮	

The table $r1$.**ratings** is controlled by $r1$ and is completely independent of the second table, $r2$.**ratings**. For instance, $r1$ and $r2$ do not have to agree on the rating for a page, and do not have to rate the same pages.

The browser b can define a ratings database of its own by the following QCM program:

$$\mathbf{ratings} = r1.\mathbf{ratings} \cap r2.\mathbf{ratings}$$
$$\mathbf{ok} = \pi_{\text{hash}}(\sigma_{\text{rating}=G}\mathbf{ratings})$$

This defines two tables, b.**ratings** and b.**ok**, using relational algebra expressions, which are the basis of QCM. The table b.**ratings** is defined to be the intersection of the two remote ratings databases:

b.**ratings**

hash	rating
$h1$	G
$h4$	PG
⋮	

The second definition, for b.**ok**, is more complicated. The *selection* operator, $\sigma_{\text{rating}=G}$, constructs a table by starting with b.**ratings**, and discarding all rows not rated G. The *projection* operator, π_{hash}, discards all columns except for hash. The result is a table of hashes with G ratings, which the browser is willing to display:

b.**ok**

hash
$h1$
⋮

Now, the tables b.**ratings** and b.**ok** may be very large, because $r1$.**ratings** and $r2$.**ratings** may be large. Therefore, the actual tables are not materialized at b, and instead are fetched as needed. However, our semantics will guarantee that the application programmer can think of them as tables residing at b.

The QCM program can execute in a number of ways. In the first way, the browser obtains a page, h, from the server s. It then needs to check whether the page should be displayed, that is, whether h is in the table b.**ok**. It does this by asking QCM to evaluate a relational algebra query:

$$\sigma_{\text{hash}=h}\mathbf{ok}.$$

This will construct a new table consisting of the rows of **ok** in which the hash is h: either the table will be empty (and the browser will not display the page), or it will have a single row, h (and the browser will display the page). The execution of QCM can be explained by algebraic reasoning. Beginning with the query, we simply fill in the definition of **ok**, and then **ratings**:

$$\begin{aligned} \sigma_{\text{hash}=h}\mathbf{ok} &= \sigma_{\text{hash}=h}(\pi_{\text{hash}}(\sigma_{\text{rating}=G}\mathbf{ratings})) \\ &= \sigma_{\text{hash}=h}(\pi_{\text{hash}}(\sigma_{\text{rating}=G}(r1.\mathbf{ratings} \cap r2.\mathbf{ratings}))) \end{aligned}$$

At this point, QCM could request the entire $r1$.**ratings** and $r2$.**ratings** databases and finish evaluating the query. However, these databases are likely to be large, so further algebraic reasoning is used:

$$\begin{aligned} \sigma_{\text{hash}=h}\mathbf{ok} = \; &(\pi_{\text{hash}}(\sigma_{\text{hash}=h,\text{rating}=G}r1.\mathbf{ratings})) \\ &\cap (\pi_{\text{hash}}(\sigma_{\text{hash}=h,\text{rating}=G}r2.\mathbf{ratings})) \end{aligned}$$

That is, it is more efficient to throw away rows and columns at $r1$ and $r2$, rather than at b.

Now, QCM will automatically send the query $(\pi_{\text{hash}}(\sigma_{\text{hash}=h,\text{rating}=G}r1.\mathbf{ratings}))$ to $r1$; the response will be small, at most one row. The responses from $r1$ and $r2$ can be combined to form the answer to the original query in the obvious way.

What we have just described is standard distributed database query evaluation. But note that it is quite expensive: to view one page, three queries and three responses were required. For efficiency, the server s might arrange to have $r1$ and $r2$ give it *certificates* for its page h. We write such certificates in the form

$$\begin{aligned} &r1 \;\textbf{says}\; r1.\mathbf{ratings} \supseteq \{\langle \mathbf{hash} : h, \mathbf{rating} : G\rangle\} \\ &r2 \;\textbf{says}\; r2.\mathbf{ratings} \supseteq \{\langle \mathbf{hash} : h, \mathbf{rating} : G\rangle\} \end{aligned}$$

These are documents signed by $r1$ and $r2$, asserting that a row with hash h and rating G appears in their respective ratings databases. (The digital signature is implicit in our certificate notation.)

In this scenario, when the browser asks s for the page h, s will send along the two ratings certificates as well. The browser will submit the certificates to QCM along with its query, $\sigma_{\text{hash}=h}\mathbf{ok}$. In this case, QCM will carefully examine the certificates, and, if everything checks out, will be able to evaluate the query without exchanging messages with $r1$ and $r2$. This sort of computation is not standard in databases.

We feel it is vital to support both of these computations. Clearly the second computation can be more efficient, but it also requires cooperation between s, $r1$, and $r2$. If the rating services do not give s a good rating for its pages, it may not be willing to carry their certificates. For this reason, other filtering proposals, such as PICS [8], also specify that both ways of obtaining ratings should be possible. The advantage of our system is that it covers all possibilities, without forcing the programmer to write code for each case. If both ratings certificates are provided by s, neither $r1$ nor $r2$ is queried by QCM; if neither certificate is provided, both $r1$ and $r2$ will be queried; and if only $r1$'s certificate is provided, then only $r2$ will be queried. In each of these cases, the same QCM program (above) is used.

Notice that we did not give QCM programs for $r1.\mathbf{ratings}$ and $r2.\mathbf{ratings}$. In fact, they need not be implemented by QCM programs; for example, they may be stored in proprietary databases. All that is required is that $r1$ and $r2$ 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.

Take, for example, the problem of *public key distribution*. VeriSign [9] is a company that 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.,

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

Using QCM, a programmer 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:

$$\begin{aligned}\mathbf{pkd} = \{&\langle \text{name} : \text{Bob}, \text{key} : K_{\text{Bob}}, \text{e-mail} : \text{bob@bob.com}\rangle\} \\ &\cup \pi_{\text{name,key,e-mail}}(\sigma_{\text{class}>1}\, \mathit{VeriSign}.\mathbf{IDs})\end{aligned}$$

The expression on the right defines a relation containing an entry for Bob, as well as other entries derived from VeriSign's database. We use $\sigma_{\text{class}>1}$ to discard Class 1 IDs in VeriSign's database as untrustworthy, and $\pi_{\text{name,key,e-mail}}$ to discard the class column from the resulting table. Thus $\mathbf{pkd}$ is a table with columns labeled name, key, and e-mail.

The programmer can query the local **pkd** for keys. For example, to obtain Alice's keys (she may have more than one), the programmer asks $\pi_{\text{key}}(\sigma_{\text{name=Alice}}\mathbf{pkd})$. By combining this query with the definition of **pkd**, and using algebraic reasoning, QCM derives a query to ask VeriSign for the appropriate certificates:

$$\sigma_{\text{name=Alice,class}>1}\, \textit{VeriSign}.\mathbf{IDs}$$

Notice that we do not bother to request Class 1 IDs. VeriSign's response will be processed to construct the table of Alice's keys, just as we described in the filtering example. Alternately, QCM can be used to validate a purported ID for Alice: if the ID is submitted to QCM along with the same query, QCM will examine its signature and use it to evaluate the query, short-circuiting any message to VeriSign.

As a final example, *access control lists (ACLs)* can be programmed in QCM:

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

This is a QCM definition of a database, **friends**, that includes the principals p, q, and the principals in the remote database p.**friends**. Sets of principals like **friends** can be used as ACLs to protect resources in the network: requests will be granted only if signed by a principal in the ACL.

3 Semantics and correctness

We describe just enough of our semantics so that we can state our correctness properties. A full semantics can be found in a separate paper [6].

The basis of QCM is the named relational algebra, which can be found in any standard database textbook (e.g., [1]). A QCM program is a set of definitions,

$$R_1 = M_1, \ldots, R_n = M_n$$

where each R_i is a relation symbol and each M_i is a relational algebra expression. We require the definitions to typecheck, and we do not permit recursive definitions. We saw several examples of QCM programs in the last section.

QCM programs interact by exchanging database queries and responses. A response is a certificate

$$q \ \mathbf{says}\ M \supseteq N$$

where M is a relational algebra query and N is a table. A query is a certificate

$$p \ \mathbf{asks}\ M \ \mathbf{noting}\ C$$

where M is the relational algebra query, and C is a set of response certificates submitted along with the query.

The exchange of messages and verification of signatures is specified by an automaton, which we do not describe fully here; the examples of the last section show some of the details. The main task of the automaton is to gather a set of inclusions,

$$M_1 \supseteq N_1, \ldots, M_n \supseteq N_n,$$

that will be used to evaluate queries. These inclusions are gathered from responses to queries and also from certificates that are submitted to the automaton along with queries. The main correctness theorem states that the responses a principal issues in fact follow from these inclusions:

Theorem: If (p **says** $M \supseteq N$), then $M \supseteq N$ is implied by the inclusions of p's automaton.

There are two important consequences of the theorem. The first is that programmers are justified in thinking of their programs as defining sets—according to the theorem, the answer to a query follows from the inclusions, and does not depend on the sequence of message exchanges used to gather the inclusions.

Second, we must be very careful in what we believe: if the inclusions are *inconsistent*, then they imply *any* conclusion $M \supseteq N$. The following example, based on SDSI 1.1, shows that carelessness in this matter can lead to a security breach. We define two groups, **school** and **employees**:

$$\begin{aligned} &\mathbf{school} = \mathbf{teachers} \cup \mathbf{admin} \cup \mathbf{students}, \\ &\mathbf{employees} = \mathbf{school} - \mathbf{students}. \end{aligned}$$

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 **says** $q \in \mathbf{students}$), or (p **says** $q \notin \mathbf{students}$). 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. Since $q \in \mathbf{students}$, by the first definition we have $q \in \mathbf{school}$. And then since $q \notin \mathbf{students}$, we have $q \in \mathbf{employees}$—even though q was never a teacher or administrator!

In our full QCM paper, we show how this can be prevented by an automatic analysis that rejects certificates that could cause inconsistencies.

4 Conclusions

We have described a security infrastructure, QCM, that is appropriate for the application programmer. Our main point is that since cryptography and message exchange are complicated and error-prone, they should be managed automatically by the infrastructure rather than the programmer. Hence, the basic primitive of

QCM is authenticated data distribution, which is implemented by the automatic management of queries and certificates.

QCM has a formal semantics, and this has been invaluable in pointing out areas which could lead to security failures in QCM programs. In particular, it led us to develop an automatic analysis of certificate contents to prevent semantic inconsistencies.

We are working 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] Ross Anderson. Why cryptosystems fail. In *1st ACM Conference on Computer and Communications Security*, November 1993.

[3] Ross Anderson. Making smartcard systems robust. In *Cardis 94*, 1994.

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

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

[6] Carl A. Gunter, Trevor Jim, and Bow-Yaw Wang. Authenticated data distribution using query certificate managers. Submitted for publication, 1997.

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

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

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