A Reliable, Scalable General-purpose Certificate Store

1. Introduction

Traditionally X.509 certificates were intended to be stored in an X.500 directory, a semi-mythical beast only rarely seen. X.509 itself was originally designed as part of the access control mechanism for the directory, but this goal has since become inverted, with the X.500 directory becoming subservient to X.509. Directory access was handled via the Directory Access Protocol (DAP), an extremely complex, heavyweight, and difficult-to-implement protocol. The whole system is specified in about 2 inches of ISO standards which (when implemented) can run to 100,000+ lines of code.

Because of the difficulties, both technical and political, encountered in trying to implement the full X.500 directory, there is currently no standard certificate storage or management mechanism. One way in which security protocol designers have worked around this problem for the last decade or so is by including in any exchanged messages a full chain of certificates leading from the end entity up to some ultimately trusted CA root certificate which is typically hardcoded into the relying party's application. The problems inherent in relying on hardcoded certificates are ameliorated by either assuming that the user will upgrade the software periodically, thus replacing the current collection with a newer set of hardcoded certificates, or by giving the certificates extraordinarily long validity dates (20 years is popular [1]), with security concerns taking second place to the fear of adverse user reactions to dialog boxes warning of expired certificates.

This approach has a number of problems. When used for end-user certificate storage it doesn't scale well, adding and removing certificates is difficult and usually requires user intervention (in a perfect world the user would be expected to have some understanding of the issues involved in adding a new CA certificate, but in practice they'll just keep clicking "OK" until the dialogs go away), other applications can't access the certificate collection, different applications have mutually incompatible certificate collections (for example Netscape Navigator and MSIE differ in a number of their hardwired certificates), and the certificate management process requires using the (often very clunky) interface provided by the web browser (the Netscape interface has been described as "ugly dialog boxes that only a geek could love"[2]), making any type of automated certificate management (for example using a scripting language or rapid application development (RAD) tool such as Visual Basic or Perl) impossible.

When used by businesses and organisations, the problems are even more serious. Instead of a single, centrally managed certificate store with certain guarantees of performance, reliability, availability, and recovery, the certificate management process degenerates into a motley collection of random certificates scattered across hundreds or even thousands of machines, with no real control (or even knowledge) of what is held where and by whom.

The unavailability of a store of existing, known good certificates can also have serious performance implications. Instead of checking a digital signature by fetching a known good certificate from the store and using it to verify the signature, a relying party must extract the certificate chain from the data, walk up the chain verifying each certificate in turn, and if revocation checking is being used perform a revocation check on each certificate in the chain (this is usually turned off or not even implemented in end-user applications due to the heavy performance penalty paid in performing this check). Instead of taking a fraction of a second, architectures of this type can take tens of seconds or even minutes before all of the steps in the protocol have been completed.

This situation is clearly unsatisfactory. This paper analyses the requirements for, and presents the design of, a general-purpose certificate store which places few constraints on the underlying computer hardware or operating system used, provides a high degree of scalability (from single end users up to the corporate/CA level), and provides the level of reliability, availability, and error recovery required of such an application and stipulated in a number of standards which cover CA operation.

A companion paper "Certificate Management as Transaction Processing" will build on the foundation provided by this paper to examine the certificate management process as a form of transaction processing, leveraging the three decades of experience the industry has in this field to present a working solution to the problem of managing certificates in the real world.

1.1 Existing Work

Almost no information on the topic of reliable, scalable, general-purpose certificate stores is currently available. Although a sizeable amount of literature covers PKI theory (some of the more comprehensive examples being

[3][4][5][6]), much of it appears to be targeting slide projectors as the underlying hardware platform [7][8][9][10]. Despite extensive searching and checking with other researchers, the author was unable to identify any previously published research work in this area.

2. Requirements Analysis

Before we rush out to design a solution for the problem, we need to first determine what the problem actually is. This requires determining how the certificate store will be used, what environment it will need to function in, and what extra functionality beyond simple certificate storage it needs to provide (contrast this with the X.500 approach of starting by designing an extremely complex architecture and leaving details such as whether it can be implemented and what it will be used for and how for later).

2.1 Query Capabilities

A certificate storage mechanism will need to be able to respond to the following queries:

?? "Who signed this?" (fetch a certificate used in S/MIME, SSL, etc based on an identifier)

?? "Who issued this?" (fetch the certificate used to issue the certificate at hand)

?? "Is this valid/still valid?" (check whether the certificate at hand was ever officially issued, or was issued and has been revoked or un-issued in some manner).

In addition to these standard queries there is a somewhat vague "Get me X's certificate", where X is typically an email address or occasionally the domain name of a web server or host (in practice this isn't really true, most protocols which employ certificates carry with them all the certificates they need as part of a complete chain up to a trusted root, but the theory is that at some time in the future this won't be the case any more). For businesses, certificates may also be identified by values such as customer numbers or a credit card number or any one of a number of similar identifiers, so the solution we adopt needs to be fairly flexible in its ability to handle various forms of certificate identifier. Finally, there are assorted private-key-related queries such as "Which key/certificate should I use to sign this" which fall into the area of private-key operations and are beyond the scope of this work.

2.2 Operating Environment

Once the operational requirements are known, we need to look at the environment in which a certificate store will need to function. There are three general applications for a certificate store:

1. The end user, typically running Windows 95/98, but also Linux or a Macintosh, with a small collection of certificates used for email correspondence.
2. The business, typically running Windows NT or Unix, very occasionally systems such as IBM or Tandem mainframes. These will have a (possibly sizeable) collection of certificates for customers and/or trading partners (again, this currently isn't the case, but it's assumed that it will be at some point).
3. The CA, typically running Unix (although some will be using NT), with a large collection of certificates. CAs also need to store revocation information and often have strict logging requirements to allow the certificate issuing process to be audited.

In the case of the end user, the certificate store will be used by novices and can't require the installation of custom drivers, system services, or complex system reconfiguration. Ideally, it should employ some mechanism which is already present on the system. In the case of business users and to a lesser extent CAs, the solution employed should preferably be one which fits in with existing operations, in other words one which doesn't require major software upgrades and reconfiguration, new hardware, and extensive staff retraining to operate.

2.3 Further Requirements

Businesses and CAs have further requirements which go beyond those of end users. Unlike end-user systems where one user has complete control of the machine and, by extension, the certificate store, the systems used by businesses and CAs will typically have special auditing and security requirements which the certificate store will need to be able to handle. For example the technical guidelines for implementing the German digital signature law require that

certificate and CRL repositories be protected from unauthorised modification through mechanisms evaluated at ITSEC E2-high, and that all modifications to the certificate store be recorded on non-modifiable media [12]. Although few other digital signature specifications go to the exhaustive lengths of the German technical requirements, being instead phrased as very general "guidelines" and "recommendations", it is likely that in practice CAs will be required to record in a secure manner details of every certificate issued for auditing purposes, just as businesses already face certain auditing requirements for fraud-prevention and regulatory purposes. The Visa certificate management requirements require that the date and time, identity of the initiator of the event (for example a certificate issuance or revocation), origin of the request (if available) and details of the object affected be recorded in an audit log, that the exact use and functioning of the auditing mechanisms be carefully documented, and that the auditing information be retained for seven years [13]. Similarly, ISO 15782-1 requires the logging in an audit journal of all certificate management operations (for example receipt of certification requests, certificate issuance, revocation, and CRL generation), the identity of the person authorising the operation, and the date, time, and sequence number of the operation being performed [14]. This standard also includes requirements for data backup and business continuity planning, as does the ANSI standard which covers CA practices and policy [15].

The MasterCard certificate management requirements also require audit logging of every imaginable certificate management operation and the retention of the resulting audit records for a period of 10 years. Administrative-level access to the audit data (for example reinitialisation of the audit database) may only be carried out in the presence of a MasterCard observer. Finally, any problem with the auditing will result in the immediate shutdown of the CA. The motivation behind such stringent and detailed operational requirements is given elsewhere in the MasterCard specifications: "MasterCard has a particular interest in the operation of any certification authorities that can sign certificates that may have an impact on the processing of MasterCard transactions and, therefore, the trust and quality of the MasterCard Brand" [16].

Coupled with the strict auditing and logging requirements is a need for security features which ensure that unauthorised users can't disable auditing or alter the audit record after it has been created, for example the Visa requirements state that "sufficient mechanisms must exist to prevent the deletion or modification of the archived information" and the equivalent ISO standard requires that "automated audit journals must be protected from modification or substitution". Similar wording can be found in the ANSI standard. This generally requires that the auditing mechanism be isolated from the certificate store update mechanism, so that someone authorised to update the certificate store can't affect the auditing mechanism and someone authorised to manage the auditing mechanism can't affect the certificate store. A third level of security is reserved for general users, who can't modify the certificate store or access the auditing facilities. Isolating certificate store update from auditing provides a Clark-Wilson style separation of duty mechanism [17], which guarantees the integrity of the certificate store unless the different parties conspire (the real-world translation of the Clark-Wilson policy [18], IT security management standards such as BS 7799 [19], and any number of accounting and auditing firms can provide further guidance on improving the security of this technique).

These requirements are encapsulated in the ABA digital signature guidelines' concept of a trustworthy system (which is echoed in both the Visa and ISO requirements which followed the ABA guidelines), a collection of computer hardware, software, and operating procedures which "are reasonably secure from intrusion and misuse, provide a reasonably reliable level of availability, reliability and correct operation, are reasonably suited to performing their intended functions, and adhere to generally accepted security principles" [20]. These requirements are then amplified by listing further recommendations such as access restrictions and auditing to prevent single user from compromising the system, measures to reduce the effects of disasters both natural and man-made, provision for auditing operations, and so on. The ABA guidelines recognise that there is a balance to be maintained between the level of security provided and the feasibility and viability of implementing it, and that the cost and effort involved in providing high levels of security must be balanced against the seriousness of the risk incurred through forgoing the higher levels of assurance. For example storing certificates as flat files on a public disk would be regarded as unsatisfactory, using the security and auditing facilities of a commercial database would be sufficient, and resorting to a multilevel secure trusted RDBMS would probably be regarded as excessive.

3. Possible Approaches

There are several approaches used by current applications, a number of which can be dismissed out of hand. For example storing certificates in the Windows registry is inefficient, specific to one particular operating system, doesn't scale, is subject to catastrophic data loss when the registry becomes corrupted, and has myriad other problems. Alternative approaches such as using flat files and certificate containers like PKCS #15 soft-tokens [21] have similar

problems with efficiency and scalability (in all fairness it must be said that none of these were ever intended for use as general-purpose certificate stores).

This leaves two other possibilities, some form of database (either one which provides simple key-and-value lookup or one supporting a full relational model) and LDAP (nee X.500).

3.1 Berkeley DB

Berkeley DB and its various ancestors and cousins dbm, ndbm, and gdbm provide an efficient means of storing and retrieving records consisting of key/value pairs to and from disk. In effect, Berkeley DB provides a disk-based extensible hashing scheme [22]. Because of its simplicity, access is extremely fast, however this also severely limits its usefulness since the only operations it directly supports are 'add data where key = x' and 'select data where key = x' (newer versions also support range-based comparisons using B-trees [23]). Logging is present only at a rudimentary level used to recover from crashes, multiple keys are handled by pointing the associated data value for each key type to a record number (Recno) which points to the actual data value, there are no security or auditing facilities which would be required for a business or CA key store, and any query more complex than fetching a value based on a key isn't supported. Although Berkeley DB provides a fast and efficient means of managing key/value pairs, the amount of effort required to turn it into a full-featured certificate store is too great to make its use for this purpose feasible.

3.2 LDAP

LDAP (Lightweight Directory Access Protocol) was originally developed at the University of Michigan as a lightweight, TCP/IP-based version of X.500's DAP (specifically, it was DAP with almost everything stripped out to allow it to run on an 80286-based PC) [24]. Later updates to the protocol turned it into HDAP, although for historical reasons the original L prefix is retained. An LDAP client connects to a server to request information from the directory or submit information to be added to the directory. Unlike a relational database which stores information in rows of tables, an LDAP directory stores information in a hierarchical manner, although the underlying database engine converts it back into a flat format. Most of the commonly-used LDAP servers such as slapd/OpenLDAP, Netscape Directory Server, MessageDirect (formerly Isode), and others, use Berkeley DB as their underlying database engine. In order to implement the complex X.500 matching rules required by LDAP using Berkeley DB, the server maintains multiple indices which are used to handle exact matches, approximate matches, substring matches (with the number of indices required being proportional to the total length of the string), and so on.

Due to the fact that it represents relatively immature technology, LDAP has a number of problems, not helped by the fact that its being based on X.500 makes it extremely complex to use. The complexity issue has already been alluded to above, and leads to problems both for humans (LDAP queries are specified in a manner which is incomprehensible to most users) and computers (processing these queries is complex and time-consuming). Beyond the complexity problems lie a number of practical issues. Since much LDAP technology is still at the experimental stage, running and using an LDAP server presents a considerable challenge for organisations as they need to invest a sizeable amount of time, money, and personnel into installing and configuring the server, setting up schemas, decorating the directory with certificates, redefining the schemas and adding new object classes for all the certificates which don't fit the existing ones, adding more certificates, redefining everything again to account for further anomalies (these iterations typically go on for awhile, and often never settle down), handling issues like maintenance and data backup and recovery (one widely-used server's backup process consists of copying all files used into another directory; restoring from backups involves copying everything back again [25]) and so on.

Once all this is done, user machines need to have LDAP clients installed, LDAP traffic needs to be coaxed through firewalls (a member of the securities industry has estimated that it would take a years work before the required infrastructure was in place to start allowing incoming LDAP queries in through firewalls [26]), and a dozen other obstacles need to be overcome before anything can be made to function. Based on feedback from users who have trialled LDAP for use as a certificate and CRL store, the technology still has some way to go before it's useful, and this hasn't touched the question of whether it has the scalability and reliability which will be required of it [27].

Finally, the fact that an LDAP server's ability to store certificates is totally dependant on the internal details of the certificate means that it can never function as a general-purpose certificate store, since it can only store certificates which match the schema and DN and attribute structure expected by the server (some applications don't even store the certificate under the subject DN but instead use the DN of the person who "owns" the certificate). Since current industry practice is to stuff everything imaginable into a certificate [28], it's unlikely that an X.500-style directory will

be able to serve as a general-purpose certificate store unless the certificates have been specially crafted to work with it. This leads to serious interoperability problems across products from different vendors since the directory schema will end up being defined both by what the product or products being used can do, and by what they can't do (for example product A may use a particular attribute which isn't supported by product B). In one large-scale federated system which was constructed using an X.500-style directory the only solution to this problem was to force all participants to use a lowest-common-denominator fixed schema, a process described as "extraordinarily painful" by one of the participants.

3.3 Relational database

The final possibility for a certificate store is a full relational database. Relational databases (RDBMSs) have gone through decades of development and refinement, have a large and well-established support infrastructure (both in terms of personnel and hardware and software), and are available from a wide variety of sources. Using an RDBMS as a certificate store has a number of advantages over other alternatives. The most obvious advantage is their pervasiveness throughout the industry, with the effect that most businesses are already running some form of RDBMS as part of their day-to-day operation, so that extending it to function as a general-purpose certificate store merely involves providing some extra disk space for the required tables. In the rare case where an existing database isn't present, a high-performance, low-cost or (in many cases) free offering such as MySQL [29][30] can be downloaded from the Internet and installed with a minimum of fuss.

Most RDBMSs provide the auditing, logging, recovery, and security features required of a certificate store [31] (in fact one of the few surviving full X.500 directory products actually uses "an underlying RDBMS [Ingres] for its high integrity and indexed storage system" [32]), and the powerful query handling and report generation facilities provide more than enough support to extract information about the day-to-day operation of a CA or the certificate management process in a business.

Moving to smaller-scale deployment, the drivers and technology required for a relational database certificate store are already built into the most common platform in the form of Windows ODBC drivers, and therefore require zero installation and configuration effort for the majority of end users. A number of small-scale, free databases are available for other platforms, and through various ODBC porting efforts [33][34] will hopefully attain the pervasiveness of ODBC under Windows by being included with standard installs of the operating system.

Finally, the RDBMS has another major advantage over LDAP (shared to some extent by Berkeley DB): it is insensitive to the format of the data it is storing. That is, the certificates being stored may (and usually do) contain arbitrarily garbled and broken DN's and other information, but since they're viewed simply as a collection of keys to search on and an opaque blob, the RDBMS doesn't care what they contain.

4. Using an RDBMS as a Certificate Store

A RDBMS would appear to be an ideal match for the requirements identified earlier for a general-purpose certificate store. This section will cover the various issues involved in using an RDBMS for this purpose, with implementation details being covered in the following section.

4.1 Correctness

Certificate and CRL updates must conform to the correctness principle, which stipulates that if a transaction is initiated with the database in a consistent state then once the transaction has concluded the database is also in a consistent state. Correctly implemented transactions are said to have ACID properties (or occasionally to pass the ACID test), where ACID stands for atomicity (the transaction is executed in an all-or-nothing manner), consistency (the transaction preserves the consistency of the database), isolation (each transaction is executed as if no other transactions were executing at the same time), and most importantly of all durability (the effect of the transaction, once completed, cannot be lost) [35][36][37]. ACID corresponds directly to the Clark-Wilson concept of a well-formed transaction, one which is constrained in such a way that it preserves the integrity of the data [17]. In Clark-Wilson terms, the ACID properties of an RDBMS ensure that constrained data items (CDIs, the objects to be protected) are taken from an initially valid state into a final state which is also valid by a transformation procedure (TP) on the CDI.

4.2 Security

In order to provide a separation of duty mechanism, we require certain security features in the database which will prevent a single user from compromising the integrity of the data. Most RDBMS' provide this type of facility through the GRANT/REVOKE statements which allow a users rights to access data and execute anything other than a particular type of SQL statement to be selectively restricted.

There are three separate roles which need to be handled: users authorised to read certificates and CRLs (in other words to perform SELECT operations), users authorised to add certificates and CRLs (in other words to perform INSERT operations alongside SELECTs), and users authorised to manage auditing (but not to add certificates or CRLs). The least privileged users (the unwashed masses) are only allowed to read certificates or CRLs:

```
REVOKE ALL ON certificates FROM PUBLIC;
GRANT SELECT ON certificates TO PUBLIC;
```

Some RDBMS' provide an even finer degree of access, allowing selects only on a particular column so that only the certificate itself could be read. Since the other columns can be reconstructed from the data in the certificate, this isn't necessary in this case.

Users authorised to add certificates and CRLs are slightly more privileged, being granted INSERT as well as SELECT privileges. These users aren't however given UPDATE or DELETE privileges, so that once a record and its associated keys are successfully entered into the table, it's not possible for them to alter any part of it. Finally, users who handle auditing have CREATE TRIGGER and ALTER TABLE privileges but can't add data to the certificate or CRL tables. Since this is a one-off operation, it would probably be performed once by someone in an administrative capacity (with MasterCard observers present if required) and then locked down to prevent entries from being deleted from the audit table and to prevent the auditing capability itself from being disabled or altered.

These privileged operations can be further constrained to require extended user authentication, or to require local access to the system (so that attempts to add a certificate from a remote network connection would be disallowed). For example DB2 can tie into the IBM Resource Access Control Facility (RACF) to require that certain operations are made by appropriately authorised users with local system access or extra levels of authentication like RACF PassTickets (a one-time-password mechanism which cryptographically binds the user ID, requested application, RACF signon key, and request time into an authorisation ticket), and that requests originating through SNA or TCP/IP distributed data facility (DDF) mechanisms are disallowed [38].

These security measures satisfy the Clark-Wilson requirements for separation of duty and user authentication for users authorised to execute a TP, as well as meeting the requirements set out in certificate processing standards such as the ISO, MasterCard, and Visa ones.

4.3 Auditing

The most straightforward means of handling auditing is to use the RDBMS' built-in logging facilities to record all updates performed on the database. Logged events are usually written to a predefined table used to record logging information, and can be read and processed using standard database techniques.

The built-in logging facilities generally leave something to be desired in terms of the information they provide since they represent a one-size-fits-all logging facility intended to record everything which happens within the database. In practice it would be preferable to record only selected details of an event, typically the time and date of the update, the identity of the user performing the update and details of where they made the update from (if available), and perhaps a copy of the certificate or CRL which was added (although this should already be present in the table used to store certificate or CRL data).

We can perform this type of user-defined auditing by using triggers, functions which automatically fire when an operation such as an update takes place. Triggers are handled automatically by the database and can't be avoided or bypassed, so that an insertion into a table would always trigger the creation of an audit entry for it. This means that even if someone were to bypass the certificate management application and manipulate the database through some other means, the trigger would still fire and record the change in the audit log. This fine-grained level of auditing is generally used to provide financial auditing facilities (recording the data which was changed) while general logging is considered a security audit facility (recording the actions which were performed) [39]. An example of an audit trigger

is shown in Figure 1. This trigger will fire once for each successful certificate addition, writing the time of the update, the identity of the user performing the update, the common name in the certificate, and its key ID to a logging table.

```
CREATE TRIGGER log_cert_updates
AFTER INSERT ON certificates FOR EACH ROW
BEGIN
  INSERT INTO cert_log (date, user, commonName, keyID)
  VALUES (SYSDATE, USER, :commonName, :keyID);
END;
```

Figure 1: Database trigger used for auditing

(the syntax varies slightly from vendor to vendor, the example above is for Oracle [40]). This type of auditing facility satisfies the Clark-Wilson requirement that all TPs write to an append-only CDI (the logging table) all information necessary to permit the nature of the operation to be reconstructed. As a somewhat extreme case, the ability to log actions both before and after an update could even be used in the style of a double-entry bookkeeping system which requires that any modification of the books comprises two parts which balance each other. Any imbalance can be detected by balancing the books (comparing the pre- and post-update logs). Again, this level of auditing can be used to satisfy the requirements in various certificate processing standards.

5. Implementation Issues

There are several problems which need to be overcome when storing certificates in an RDBMS. The first problem is that certificates are binary blobs which can't be handled by some databases and their fields can't be indexed by most databases. Since all databases can handle ASCII text, this problem can be solved by encoding binary fields using base64 encoding if required. Further issues include how we can look up certificates in the store, how revocation checking is handled, and various efficiency considerations. These are examined in more detail below.

5.1 Search Keys

Once a mechanism for storing a certificate in a database exists, the problem of identifying the certificate arises. In the X.509 world, certificates are usually identified by their issuer or subject Distinguished Name (DN) and serial number, an identifier type which can't be easily applied in a useful manner. Complicating this is the fact that certificates have been spotted with names which consist of everything from full X.500 DNs, assorted portions of DNs and things which look like DNs but aren't, down to single partial DN components and email addresses. Luckily there's no need to actually do anything with any of these components, so the whole can be treated as an opaque, somewhat awkward blob used to identify a certificate. Due to ambiguities and implementation errors in DN encodings, the value we hash is the DN exactly as it appears encoded in the certificate, rather than any application-internal form which may canonicalise it and correct encoding errors. Since the blob can be of arbitrary size, we hash it down to a manageable size using SHA-1, and use that (in base64-encoded form) to look up the certificate. The use of a hash of the blob to identify a certificate has the additional benefit that the hash is virtually guaranteed to be unique[1], so that by specifying that a row entry must be unique when the database table is created we can have the database warn us of attempts to add duplicate data. In addition the fact that the hash values will be more or less evenly distributed leads to excellent lookup performance, a feature which is covered in more detail in the section on efficiency considerations.

Now that we have a basic strategy for turning the identifiers used in certificates into something usable, we need to look at the types of identification information which need to be stored. The DN hash or nameID is used to identify certificate issuers (that is, the issuer of a certificate would be located by looking up the nameID of the certificate's issuer DN), providing a means of handling the "Who issued this?" query. To complement this we also need a hash of a certificate's issuerAndSerialNumber, which is used in S/MIME and a number of related standards to identify certificates, satisfying the "Who signed this?" query. This type of identifier is the issuerID, and is also used to satisfy "Is this still valid?" queries as explained further on.

In addition to DN-based IDs, it is also possible to identify a certificate via its subjectKeyIdentifier, which is usually a hash of the encoded public key but may also have been generated in many other (mutually incompatible) ways.

[1] If we do ever find two different DN's which hash to the same value, we get a Crypto or Eurocrypt paper as a consolation prize.

Because of this, we again hash it to a uniform size before use. This value is the keyID, and may become useful at some point for chaining or certificate identification, although currently the nameID and issuerID make it redundant.

Another commonly-used ID is a hash of the entire certificate, generally called a thumbprint but more formally an out-of-band certificate identifier or oCID for short, after its use in the standard where it was first officially defined [41]). This value is a straight SHA-1 hash and is stored in the same manner as the other hashes.

Alongside these ID types there are various other ways in which the certificate may need to be identified. The most common one is the email address, however various bits and pieces of DN's and possibly some types of certificate alternative name (for example a customer number or credit card number) may also be useful in some circumstances, so we also allow these if required — the exact details of what to use as search keys is left as a local configuration issue, since it depends on the manner in which the certificate store will be employed and on the expected contents and usage of the altName fields.

Finally, we also store one or two additional items which will make certificate management easier. The most obvious one is the certificate's expiry date, since it allows us to issue queries to easily determine which certificates have expired, or to locate those which are about to expire so they can be renewed. For example:

```
SELECT certificate FROM certificates WHERE validTo < SYSDATE + ?
```

would return all certificates which are due to expire within the next ? time interval and therefore need to be renewed.

One complication which arises is the use to which the fetched certificate will be put. It's possible for CAs to issue certificates with identical identification information but different usage conditions, so that one may be valid for encryption, one for general signing, one for signing email (a subset of general signing), one for signing for authentication (a different subset), one for signing for nonrepudiation (a subset whose definition no-one can quite agree on), and so on and so forth. Since the type and number of variations of usage condition are open-ended, there's no easy way to plan for this contingency (if we allow for *n* possibilities, someone will define an *n*+1th one the day after we deploy the certificate store). Handling this situation presents a significant problem, since no lookup mechanism (whether it be specific to an RDBMS or for any other type of certificate store) can efficiently handle a query which can involve falling back across half a dozen possibilities in turn until we find something general enough to provide a match.

Current implementations of certificate stores handle this problem in a wide variety of ways including ignoring the issue entirely and taking the first certificate they find, presenting the user with a dialog and asking them to choose (effectively equivalent to choosing a certificate at random), or crashing. The ability to handle multiple certificates returned from an RDBMS query allows us to attempt to fix this problem by stepping through the query result set looking for the most appropriate match for a particular requested usage type (for example emailProtection for S/MIME email encryption, falling back to keyEncipherment, and then to no usage at all if nothing else is available). A better solution might be to use the PGP approach of identifying the key usage in the DN, allowing the correct certificate to be accessed via a single query operation.

5.2 Revocation Checking

Revocation checking is extremely easy. Each CRL entry is stored in a table containing revocation information by converting the CRL issuer DN and certificate serial number from the CRL entry into an issuerID and entering that into the CRL table. A revocation check consists of the following point query:

```
SELECT crlData FROM crls WHERE issuerID = issuerID
```

In the vast majority of cases the certificate will not have been revoked, so the query will return instantly without performing a disk access. The largest CRLs in existence today, after several years of operation by the issuing CAs, contain only a few thousand entries, so the entire CRL index will almost certainly be held in main memory. In combination with the use of cached queries via a slight redesign of the query presented above, a revocation check will be performed using a single in-memory hashed lookup or referral to a B-tree index, a vast improvement over importing and traversing a thousand-entry CRL every time a revocation check is required.

If the revocation check returns a match, further revocation details can be fetched from the CRL data held in the CRL table. In the example used above the data is used as the value to fetch in the query, although effectively it's just a dummy used as a presence check for a CRL entry. In theory since we're not (initially) interested in the data which is stored but only performing an existence check, we could use an alternative type of query such as a count on an index to return the number of entries found. Since the index will be held in memory, this type of presence check will be

performed without any data accesses [42]. In actual tests with a range of commercial RDBMS', fetching a single row entry proved to be significantly quicker than performing an index count, probably due to the fact that the count operation never took advantage of the fact that a `UNIQUE INDEX` can never return more than one entry, so the scan could be stopped after a match was found, or even better converted to a special-case point query which doesn't return any data. Instead, the database spends some time scanning for possible entries which don't exist, providing the somewhat counterintuitive result than an existence check which involves a data fetch is much quicker than one which doesn't.

Some RDBMS' provide a `LIMIT` clause to limit the number of rows fetched in a query, but using this didn't improve the performance since the problem was caused by attempts to find non-existent entries rather than reading too many rows as the result of a successful query. Other RDBMS' provide an `EXISTS` operator which specifically informs the database that what's being performed is an existence check, allowing it to optimise the query execution on this basis [43]. Unfortunately `EXISTS` can only be used to modify the results returned from a subquery, which isn't useful for our purposes.

5.3 Efficiency Considerations

Because the certificate store may at some point need to handle large numbers of entries, we need to take care to design and employ it in a manner which leads to the most efficient access possible. The extreme simplicity of the manner in which we're using it (fixed-size columns, very simple select statements, and so on) allows us to apply various optimisation and database tuning techniques in order to produce best-case performance. For example we can take advantage of the fact that the queries used are point queries (which return one entry from one row based on an equality selection) to configure the table indexing and access mechanisms for optimum performance. With proper indexing, execution time for point queries is mostly independent of table size [44], providing a high degree of scalability.

When designing the table structure, we can take advantage of the fact that most of the entries will have a fixed format to define fixed rather than variable-length fields, since storing variable-length records leads to inefficiencies due to the need to maintain complex length and offset information for the variable-length fields. In addition by moving fields such as the certificate blob to the end of the row, we can leave the more efficient fixed-length records at the start for faster access (some RDBMS' will perform this rearranging automatically). Other systems put blob management under the control of the user, for example Informix allows blobs to be stored in blobspace, a separate disk area reserved for blobs, with the main table holding only a descriptor for the blobspace entry [45]. This is a hybrid form of in-table storage and Postgres95's external storage which is described below, and is used to circumvent the problem that storing blobs directly in the main table requires (at least in the Informix implementation) interspersing blob data blocks with table row data blocks, increasing the size of the table [46]. Storing blobs in blobspace improves performance by separating the table rows (which are only used as search keys) from blobs (which represent the data actually being retrieved during queries).

Certificate blob fields have the potential to be particularly troublesome since they may lead to the record overflowing the basic allocation block used by the RDBMS. A somewhat primitive solution to this problem is to avoid storing the blob in the table and to only store a link to an external file, which is the approach taken by Postgres95 for providing support for large binary data objects [47]. This is somewhat unsatisfactory since we now lose the assurance that the RDBMS' ACID properties also cover the blob.

If the blob overflows a single allocation block, it will be stored as a spanned record over two or more blocks [48]. For example Sybase stores blobs separately from other table row data as a linked list of 2kB blocks [49]. Because of the relatively small (at least comp ared to block size) size of each certificate, it's unlikely that storing them directly in the database will lead to the creation of too many spanned records, and the overhead is negligible (the few blobs which result in the creation of spanned records will merely incur an extra read to retrieve both record fragments, and since most RDBMS' will read an entire string of blocks at once and disk subsystems perform read-ahead cacheing, the performance hit won't be anywhere near as serious as it is for objects like JPEGs and MPEGs which are more usually the cause of spanned records).

Once we've figured out how to arrange the data, we need to design an optimum query strategy to retrieve certificates from disk. An RDBMS' query processing consists of three steps, parsing the query into a parse tree, rewriting the parsed query into the form which (it is hoped) will minimise execution time (also known as query optimisation), and finally preparing a physical query plan to apply the processed query to the database [50]. Since we know that all queries will be point queries, we can ignore consideration of complexities such as views and joins, pitfalls like the use

of `DISTINCT` and `ORDER BY` (which entail a sort of the data), and anklebiters such as the fact that joining expressions with `OR` or the use of subqueries may cause the RDBMS to avoid using an index, a serious performance-killer for a certificate store [51]). In fact the query being used, `SELECT certificate FROM certificates WHERE` *`key = value`*, is so simple that there isn't really any optimisation which can be applied to it, and all the overhead lies in the parsing and rewriting steps [52]. If we can somehow avoid these steps, we can eliminate all processing overhead except that resulting from the actual execution of the query.

Many databases try to cache repeated queries to avoid having to reparse the same query over and over again, so with a little careful design we can ensure that, after the query is parsed and cached for the first time, it never needs to be parsed again. For example Oracle stores parsed queries in the shared system global area (SGA) and tries to locate repeated uses of the same query by performing a hashed lookup of the not-yet-parsed query text against the text which generated existing cached queries. If there's a match, it'll reuse the cached copy rather than parsing the newly-submitted query text [53].

We can ensure that a query always has the same text by making use of bound variables to contain changing data such as the contents of fields. A bound variable consists of a pointer to a location containing the non-constant data, so that instead of specifying (for example) `SELECT certificate FROM certificates WHERE nameID = eyAxOHCmuXpVFil`, we would use `SELECT certificate FROM certificates WHERE nameID = :key`, where `:key` is a reference to the location containing the key to search on. Since this text string is constant, it will result in the same pre-parsed cached query being used for each new query submitted, reducing the processing step to a simple hashed lookup. The resulting query path is shown in Figure 2.

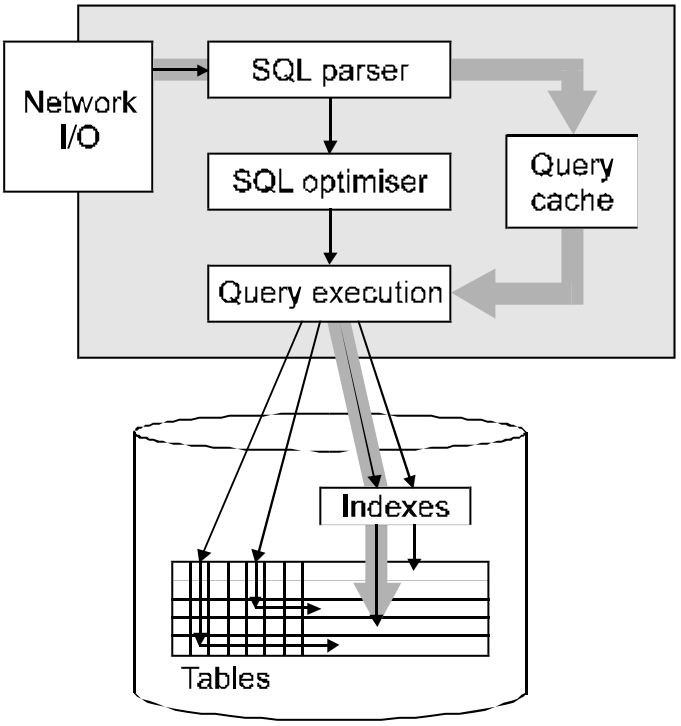

Figure 2: Optimised key database query path

The types of values being used as keys by the certificate store have both an advantage and a disadvantage. The advantage arises from the fact that they are unique and completely random, which leads to optimum performance in the data structures used for searching for records (even the most simplistic tree implementation will end up probabalistically balanced through the insertion of random entries). Because of this the columns being indexed have excellent selectivity, meaning that no two rows have the same value. If rows did contain the same value, the matching entries would be stored in linked lists in the index, possibly stretching across multiple memory pages. Searching such an index is rather inefficient since the RDBMS must lock all the affected index and data pages as it processes the list, slowing down not only the query itself but all other queries which reference the affected pages.

The disadvantage of the values being used as keys is that the size of the keys leads to some inefficiency because it reduces the number of key/pointer pairs which will fit into a tree node, which usually corresponds to a page of memory or 4kB (with a few exception like Sybase and MS Access, which use 2kB memory pages and disk blocks). This decreases the branching factor of the tree, leading to deeper trees which take longer to search [54]. This problem can be ameliorated by using not the full 160-bit (27 bytes once base64-encoded) hash of a key value but instead using only 128 or even 64 bits, for a 20% or 60% reduction in space used. An alternative solution takes advantage of some RDBMS' capabilities to use a hashed index instead of a B-tree index. A hashed index can resolve point queries of the type used in the certificate store in a single access provided there are no overflow chains, making it ideally suited to this application. RDBMS' which support this type of index allow its use to be specified when the index is created, for example Ingres allows it be given on creation with:

```
CREATE INDEX keyid_idx ON certificates(keyid) WITH STRUCTURE = HASH
```

which will create a hashed index instead of the default B-tree one [55]. Hashed indices are unaffected by key size (except for the fact that it takes fractionally longer to hash larger keys), so the certificate store should use hashed indices if at all possible. It does this through an RDBMS-specific rewrite rule which rewrites generic index-creation commands into ones which create hashed indices if the underlying database supports it.

In addition to the use of special-case optimisations such as hashed indices, we can also apply some generic improvements such as using a `UNIQUE INDEX` (whose benefits have been mentioned earlier) and specifying that the table columns are `NOT NULL`, which improves performance in all RDBMS types by eliminating the need to handle the special-case `NULL` value.

An efficiency consideration which only applies to the CRL table is that of clustering entries to allow for efficient table scans. The CRL table is the only one for which this type of access occurs when a CRL is assembled. Clustering allows for more efficient scanning by collocating entries with the same primary key (a table column designated by the user) [56]. By designating the CRL nameID as the primary key, we ensure that all CRL entries for a given CA are clustered in one location, so that assembling a CRL consists of a single linear read of the entire collection of entries.

6. RDBMS Interface

The RDBMS interface is implemented in three layers as shown in Figure 3. The first layer, common to all database types, translates a key management function call into SQL and performs base64 encoding of binary data if necessary. The output of this layer is an SQL query string. For example the hypothetical function:

```
getCertificateFromID( char *nameID, unsigned char *certificate );
```

might produce the SQL query:

```
SELECT certificate FROM certificates WHERE nameID = nameID
```

The second layer marshals the data to be sent to the RDBMS backend and unmarshals any data returned from the it. If the RDBMS is a local one which interacts directly with the application, this step is omitted.

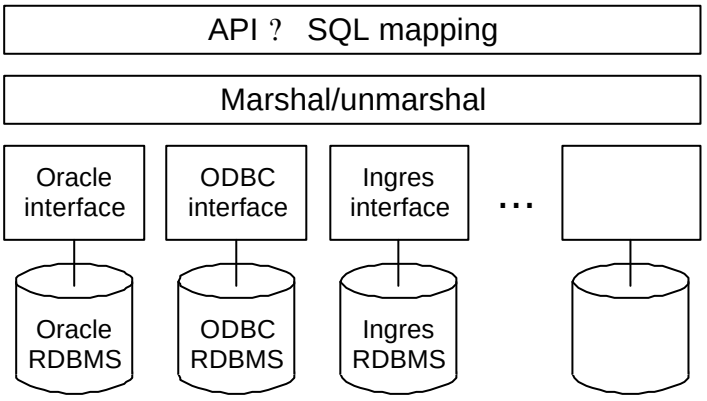

Figure 3: RDBMS interface layers

The final, third layer consists of plugins for the various database backends which consist of glue code which is specific to the database being used, but this is fairly easy to implement since the queries are extremely simple so that the glue routines typically comprise around 200 lines of code for each database type. Because of the simplicity of the queries, a relatively basic database is a lot easier to tie in than a complex, full-featured database. For example the MySQL interface took an afternoon to write, whereas the extremely complex (and rather quirky) ODBC interface took nearly a week to write and debug.

Besides the general glue code, the plugin layer also performs other functions such as providing a general SQL rewrite mechanism which performs any required translation from generic SQL into the RDBMS-specific format. For example MySQL has no `CREATE INDEX` command, so this is rewritten as `ALTER TABLE certificates ADD INDEX ...` (in newer versions of MySQL this re-mapping is performed by MySQL and is transparent to the caller [57]). A more commonly-encountered rewrite rule is the one for key blob data, since there is no generally standardised data type for handling blobs. For example MS SQL Server uses `LONG VARCHAR`, Oracle uses `LONG RAW`, Informix uses `BYTE`, and Sybase uses `IMAGE` (with the additional constraint that the data must be specified in hexadecimal rather than binary, requiring further mapping of the data), so the rewrite rules adjust the blob data type to match the underlying RDBMS.

The RDBMS plugins provide five services, of which the first two are `openDatabase` and `closeDatabase`, which open and close the connection to the underlying database server. The remaining three services submit a query to the database backend, with varying levels of functionality. For example the service which adds data to the database doesn't need to perform elaborate checking for returned data (in fact apart from checking for possible errors it doesn't need to retrieve any data at all), whereas the service which reads data from the database needs to go through a number of steps to determine whether the data was found and then read it through the client connection.

The first query service, `performCheck`, performs a transaction which checks for the existence of an object without actually returning any data, for example to check whether a certificate revocation is present in a database. The second and most complex service, `performQuery`, performs a transaction which returns data (the certificate or CRL entry being read). The third service, `performUpdate`, performs a transaction which updates the database without returning any data (to store a certificate). A typical data flow through the three layers starting with the high-level call `createDatabase("certificates")` might be as follows.

Layer 1: This layer rewrites the function call into a series of SQL statements, of which the main one is:

```
CREATE TABLE certificates ( commonName VARCHAR(64), email VARCHAR(64), [...] )
```

Layer 2: The query is marshalled and forwarded to the database-specific plugin.

Layer 3: In this case the RDBMS doesn't support VARCHAR, so the previous query is rewritten as:

```
CREATE TABLE certificates ( commonName CHAR(64), email CHAR(64), [...] )
```

Once this step is complete the query is submitted to the database and the result checked to make sure the query executed successfully. In this case we use the `performUpdate` service since we're performing an update transaction without returning any data. This performs the following (simplified) operations:

```
DBMSQuery( query );
if( DBMSGetResult() == NULL )
  return( ERROR );
DBMSFreeResult();
return( OK );
```

All other operations on the database are implemented in a similar manner.

7. Usage Experience

The design described here has been in use for a number of years as part of the cryptlib security toolkit [58], an open-source software package which runs on a wide variety of platforms. The backend store which is by far the most widely used is Windows' ODBC, which is present as a built-in component of all newer Windows releases. This has the advantage that for most end users, the certificate store facility works automatically without any need for user installation or configuration.

Usage information for larger-scale applications has been less easy to come by. Although simulations with very large datasets (millions of entries) were carried out to empirically evaluate the schemes presented in this paper, it won't be known for some time how well these tests will reflect actual practice, for example, how well the access patterns for any eventual large-scale PKI will reflect the random access model used for performance testing.

Real-life data provided by a number of large public CAs revealed that the certificate store of choice in each case was an RDBMS and involved standard industry practice for handling issues of reliability (use of redundant servers, UPSs with backup generators, a rigorous backup strategy, and so on in textbook fashion) and scalability (as one respondent put it, the standard solution to CA scalability is simply to "pile on more boxes and get more pipes"). The real-world load experienced by these systems can range from some hundreds to low thousands of new certificates issued each day (the real upper limit, at least for the higher assurance certificate levels, is determined by paper-shuffling rates, not server performance), with CRLs issued every few hours. A typical CRL production technique was to perform a straight database scan during periods of low activity, dumping the result to the next CRL to be issued. These real-world situations match fairly closely the certificate store strategy presented in this work.

The only research paper which touches on this area dates from 1995 and mentions MITRE Corporation's experience with certificate stores [59]. Their initial attempts to use an X.500-style directory for this purpose failed due to performance problems arising from the way information is represented in the directory, resulting in them falling back to their existing data warehouse, whose RDBMS facilities handled the task admirably.

8. Conclusion

This paper has presented a design for a reliable, scalable, general-purpose certificate store based on existing technology which has been proven in the field and which is independent of the underlying operating system and hardware platform. A companion paper "Certificate Management as Transaction Processing" will examine the application of another type of well-established technology to the problem of certificate management. A full implementation of the techniques described in these papers is available over the Internet [60].

9. References

[1] "Lifetime of certs now in circulation", Dan Geer, posting to c2 cryptography mailing list, message-ID `199901251953.AA09739@world.std.com`, 25 January 1999.

[2] "Web Security & Commerce", Simpson Garfinkel and Gene Spafford, O'Reilly & Associates, 1997.

[3] "Understanding the Public-Key Infrastructure", Carlisle Adams and Steve Lloyd, Macmillan Technical Publishing, 1999.

[4] "Internet X.509 Public Key Infrastructure PKIX Roadmap", Alfred Arsenault and Sean Turner, `draft-ietf-pkix-roadmap-04.txt`, 22 October 1999.

[5] "Digital Certificates: Applied Internet Security", Jalal Feghhi and Peter Williams, Addison-Wesley, 1998.

[6] "Secure Electronic Commerce: Building the Infrastructure for Digital Signatures and Encryption", Warwick Ford and Michael Baum, Prentice-Hall, 1997.

[7] "The 1997 RSA Data Security Conference Proceedings", RSA Data Security Inc, Redwood City, California, 1997.

[8] "The 1998 RSA Data Security Conference Proceedings", RSA Data Security Inc, Redwood City, California, 1998.

[9] "The 1999 RSA Data Security Conference Proceedings", RSA Data Security Inc, Redwood City, California, 1999.

[10] "The 2000 RSA Data Security Conference Proceedings", RSA Data Security Inc, Redwood City, California, 2000.

[12] "Maßnahmenkatalog für digitale Signaturen auf Grundlage von SigG und SigV", Bundesamt für Sicherheit in der Informationstechnik, 18 November 1997.

[13] "Visa Secure Electronic Commerce — Consumer Certificate Processor Requirements, Version 1.1", Visa International, November 1997.

[14] "ISO 15782-1:1999, Banking — Certificate Management Part 1: Public Key Certificates", International Organisation for Standardisation, 1999.

[15] "PKI Practices and Policy Framework", ANSI X9.79, American National Standards Institute, 2000.

[16] "Security Requirements for Certificate Authorities and Payment Gateways: General and SET Version 1.0 CAs, Version 1.0", MasterCard International Inc, January 1998.

[17] "A Comparison of Commecial and Military Computer Security Policies", David Clark and David Wilson, *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, April 1987, p.184.

[18] "Separation of Duties Audit Concerns and the Clark-Wilson Data Integrity Model", Robert Moeller, *Computer Security and Information Integrity*, Elseview Science Publishers, 1991, p.319.

[19] "A Code of Practice for Information Security Management", British Standard 7799-1, British Standards Institute, 1999.

[20] "Digital Signature Guidelines: Legal Infrastructure for Certification Authorities and Secure Electronic Commerce", Electronic Commerce and Information Technology Division, American Bar Association, 1 August 1996.

[21] "PKCS #15 v1.1: Cryptographic Token Information Syntax Standard", RSA Laboratories, 18 January 2000.

[22] "Berkeley DB", Michael Olson, Keith Bostic, and Margo Seltzer, *Proceedings of the 1999 Usenix Annual Technical Conference (Freenix Track)*, June 1999, p.183.

[23] "The Ubiquitous B-Tree", Douglas Comer, *ACM Computing Surveys*, **Vol.11**, **No.2** (June 1979), p.121.

[24] "A System Administrators View of LDAP", Bruce Markey, *;login*, **Vol.22**, **No.5** (October 1997), p.15.

[25] "Netscape Directory Server Administrator's Guide, Version 4.1", Netscape Communications Corporation, 1999.

[26] "Re: CRL Push over S/Mime", Dwight Arthur, posting to `ietf-pkix` list, message-ID `33C2A1FD.C0A9C4B@mindspring.com`, 8 July 1997.

[27] "PKI: First Contact", Steve Whitlock, Boeing Information Services, presented at "Trust and Confidence in the Global Infrastructure", The Open Group, 25th October 1999.

[28] "The X.509 Style Guide", Peter Gutmann, `http://www.cs.auckland.ac.nz/~pgut001/pubs/x509guide.txt`.

[29] "MySQL", `http://www.mysql.com`.

[30] "MySQL", Paul DuBois, New Riders, December 1999.

[31] "Database Security and Integrity", Eduardo Fernandez, Rita Summers, and Christopher Wood, Addison-Wesley Publishing Company, 1981.

[32] "Applying Relational Searching to Distributed Object Oriented Directory Systems", Alan Lloyd, Rick Harvey, and Andrew Hacking, OpenDirectory white paper, 25 February 1999.

[33] "iODBC Driver Manager", `http://www.iodbc.org/`.

[34] "unixODBC", `http://genix.net/unixODBC/`.

[35] "Transaction Processing: Concepts and Techniques" Jim Gray and Andreas Reuter, Morgan Kaufmann, 1993.

[36] "Atomic Transactions", Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete, Morgan Kaufmann, 1994.

[37] "Principles of Transaction Processing", Philip Bernstein and Eric Newcomer, Morgan Kaufman Series in Data Management Systems, January 1997.

[38] "DB2 for OS/390 Version 5 Administration Guide", IBM Corporation, 1997.

[39] "Oracle7 Server Application Developer's Guide", Oracle Corporation, February 1996.

[40] "Oracle7 Server Administrator's Guide", Oracle Corporation, February 1996.

[41] "Internet X.509 Public Key Infrastructure: Certificate Management Protocols", RFC 2510, Carlisle Adams and Stephen Farrell, March 1999.

[42] "Optimizing Performance in DB2 Software", William Inmon, Prentice-Hall Inc, 1988.

[43] "Sybase SQL Server Performance and Tuning Guide", Server Publications Group, Sybase Corporation, 7 February 1996.

[44] "The Benchmark Handbook for Database and Transaction Processing Systems (2nd ed)", Jim Gray, Morgan Kaufmann Series in Data Management Systems, 1993.

[45] "Informix Guide to SQL Reference", Informix Software Inc, February 1998.

[46] "Performance Guide for Informix Dynamic Server", Informix Software Inc, February 1998.

[47] "Postgres95 User Manual", Andrew Yu and Jolly Chen, Department of EECS, University of California at Berkeley, 5 September 1995.

[48] "File Organization for Database Design", Gio Wiederhold, McGraw-Hill, 1987.

[49] "Sybase SQL Server Reference Manual, Vol.1", Server Publications Group, Sybase Corporation, 24 January 1996.

[50] "Query evaluation techniques for large databases", Go etz Graefe, *ACM Computing Surveys*, **Vol.25**, **No.2** (June 1993), p.73.

[51] "Database Tuning: A Principled Approach", Dennis Shasta, Prentice Hall, 1992.

[52] "Database System Implementation", Hector Garcia-Molina, Jeffrey Ullman, and Jennifer Widom, Prentice-Hall, 2000.

[53] "Oracle7 Server Tuning", Oracle Corporation, June 1996.

[54] "Utilization of B-trees with Inserts, Deletes and Modifies", Theodore Johnson and Dennis Shasha, *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, March 1989, p.235.

[55] "Ingres II SQL Reference Guide", Computer Associates International, Inc., 1998.

[56] "Handbook of relational database design", Candace Fleming and Barbara von Halle, Addison-Wesley, 1989.

[57] "MySQL Reference Manual", TcX AB, Detron HB and Monty Program KB, 25 February 2000.

[58] "The Design of a Cryptographic Security Architecture", Peter Gutmann, *Proceedings of the 8th Usenix Security Symposium*, August 1999, p.153.

[59] "Developing and Deploying Corporate Cryptographic Systems", Diane Coe and Judith Furlong, *Proceedings of the First Usenix Workshop on Electronic Commerce*, July 1995.

[60] "cryptlib Security Toolkit", `http://www.cs.auckland.ac.nz/~pgut001/cryptlib/`.