Address Space PKI – Database Design

Mark Reynolds

June 20, 2007

Introduction

This document describes the design of a database system for managing a set of digital objects – certificates; certificate revocation lists (CRLs); and route origination authorizations (ROAs) – in support of a PKI of IP address space and AS# allocations. The core requirement for this system is to manage the collection of ROA objects in a way that makes it straightforward for the user to identify those ROAs that are valid, and convert the IP address and AS# information in those ROAs into a form that can be postprocessed into BGP filter specifications.

Functionally, this core requirement can be broken down into three top-level steps: obtain a set of ROAs and their associated digital objects; identify those ROAs that are valid; and extract the information needed for BGP filter generation from those valid ROAs. The first part of this process, the actual mechanism used to obtain the digital objects, is not directly within the scope of our responsibility, although our software does interact with this mechanism. The second component, the ROA validation component, contains the overwhelming majority of the effort. In order to validate a ROA one must not only verify that the ROA meets the ROA specification, as well as conforms to the profile for this PKI, it also implies that the ROA must be validated against the certificate that signed it. This, in turn, implies that this certificate itself must be validated, which leads to the need to trace up the chain from this certificate to its parent and further predecessors until one reaches a self-signed trust anchor certificate. At each point in this process the certificate in question must be validated, which involves a number of separate tests (Has it expired? Has it been revoked? Does it actually have a parent?). The third and final component, the generation of the ROA output, presupposes the existence of a ROA parser, which would have to exist in order to perform the validation step. Given that parser it is straightforward to generate any output format one wants from the IP address and AS# information.

The remainder of this document will describe the software design that has been created in an effort to meet these requirements. The description given will take a process oriented form. The software consists of a number of components that perform various aspects of the overall work required. The work performed by each component is designed to satisfy a subset of the requirements for the whole task. There is a logic flow of activity starting from the point at which a new digital object arrives, and proceeding through the point at which the filter file(s) are created. Some of the components must be executed in a specific order, while others can be executed out of order. Despite this, it is still meaningful to speak of an overall process flow, namely the set of activities that must be executed in a given time interval, nominally a single day.

This document will describe these software components, how they integrate with one another and with the external processes also taking place, and most specifically, how they conspire to realize the overall requirements of the database portion of the PKI.

Component Design

The Address Space PKI (APKI) database system manages digital objects. These objects are stored in files of three logical types: certificates, certificate revocation lists (CRLs) and route origination authorizations (ROAs). Individual users of the APKI system obtain their local files through a synchronization process with external sources of these files. The collection of local files is known as the local *repository cache*. In the general case, any process could be used for this synchronization. In the APKI database system, however, we explicitly **assume** that the process will be performed using the rsync utility. This utility is a standard component of Unix/Linux systems and was not developed or modified in our APKI system. The APKI system is closely integrated with rsync, however. Should the synchronization process change (to use http, for example), there are two dependencies with the APKI software that would need to be changed. These dependencies will be described below in subsections 2 and 6.

The APKI database software consists of the following components: (1) the rsync wrapper; (2) the rsync_aur program; (3) the rcli database (DB) client; (4) the query client; (5) the garbage collection client; (6) the chaser client; (7) the APKI database; (8) the database interface libraries; and (9) the ROA processing library. Each of these components will be described in turn.

1. The rsync wrapper

The rsync wrapper is a shell script that invokes (wraps) the rsync utility. It calls the rsync utility with a set of command line arguments that cause rsync to generate a logfile of all its actions. It also ensures that logfiles are rotated, i.e. that a new logfile is created each time rsync is run. This script can optionally specify a configuration file for rsync to use; if no such configuration file is provided, rsync uses a default configuration file. It is **assumed** that the user provides the default configuration file. The rsync wrapper may be invoked manually (by the user); it may also be invoked as a result of running the chaser client. After the rsync utility has done its work, the logfile will record the changes that rsync has made to the repository. Note carefully that the APKI query software never modifies the repository itself: it is a read-only consumer of the information in the repository.

2. The rsync_aur program

When the rsync wrapper (and, by implication rsync) has run to completion, a logfile of actions will have been generated. This file is in a format that is specific to rsync. The rsync_aur program is launched by the rsync wrapper as its last action. The rsync_aur program parses the logfile and generates a series of “events” corresponding to each action. This is one of the dependencies on the use of rsync stated above. Should another synchronization protocol be used, such as http, a different logfile format would be generated, and therefore that portion of rsync_aur that parses the logfile would have to change. Each event is actually a message sent across a socket corresponding to an action. A full description of the socket protocol is beyond the scope of this document (it can be found in the file "apkiaur.txt.") For our purposes, three events (messages) are significant. When a file is added to the repository, rsync_aur sends an "A" message to the socket. When a file is updated in the repository (new contents are stored into an existing file), rsync_aur sends a "U" message to the socket. Finally, when a file is removed from the repository, rsync_aur sends an "R" message to the socket. From a process standpoint, rsync_aur performs three actions: it opens a socket to a port specified on the command line; it parses the rsync logfile and generates a number of messages (typically A, U or R messages), and then it closes its end of the socket. This entire process is known as a socket session.

3. The rcli DB client

The rcli program is a database client. It is known as a "client" because it is a client (user) of the database. (Throughout this document "client" will always have this specific meaning.) This program can run in two modes. In a manual scenario, rcli can be invoked on the command line to add or delete objects from the database. In a deployed configuration, rcli runs as a daemon (detached process), waiting for socket connections from rsync_aur and processing the messages it receives over the socket in an attempt to bring the database up to date with respect to changes in the repository that have been caused by rsync and reported by rsync_aur. When rcli is run as a daemon it runs perpetually, processing socket sessions. Fundamentally, rcli performs only two actions: attempting to add something to the database and attempting to delete something from the database (updates are processed as a delete operation followed by an add operation).

In order for a digital object to be added to the database it must be valid at the time it is added. The rcli program therefore makes extremely heavy use of the database infrastructure libraries. In order to validate certificates and CRLs it also makes use of the OpenSSL library. In order to validate ROAs it makes use of the ROA processing library. The rcli program is thus the primary validation engine for everything in the PKI. It will not add any object known to be invalid to the database. Objects known to be valid or those whose validity cannot be determined (e.g. due to the object being loaded before its parent certificate) are both placed in the database; flags in the database are set to distinguish between these two types of objects.

The only assumption that the rcli program makes about the structure of the repository is that there is a single top-level directory that contains the trust anchor certificates whose name starts with the string "TRUST". This allows rcli to know which certificates are allowed to be self-signed.

The rcli program uses three top level algorithms from the database infrastructure libraries: retrieve_tdir(), add_object() and delete_object(). The retrieve_tdir() algorithm is an extremely simple algorithm that requests the value of the repository root directory from the metadata table of the database. (This value will have been stored into this table when the database was initialized.) This directory named is then checked against the initial portion of the fully qualified pathname of the object being added. If they do not match the user and this is an interactive invocation of rcli, the user is requested to verify that the add operation is to be performed. (In a non-interactive invocation of rcli this test is not performed.) This is a sanity check to ensure that the user truly wishes to add an object that is not stored within the directory tree rooted at the repository's top level directory. This situation might occur for the trust anchor certificates; it might also occasionally occur for objects received by an out-of-band mechanism, e.g. received by email rather than rsync.

The add_object() algorithm calls a number of subordinate algorithms in order to accomplish work. The body of add_object() is shown in the following pseudocode:

```
If  not isokfile() return error;
Typ = infer_filetype();
If ( typ is invalid or unrecognized ) return error;
If not findorcreatedir() return error;
If ( typ is certificate )
        Return add_cert();
Else if ( typ is CRL )
        Return add_crl();
Else if ( typ is ROA )
        Return add_roa();
```

The isokfile() algorithm determines if the file that is being added is a normal file, and not a socket, pipe, device file, or symbolic link. It also determines if the file is readable. The infer_filetype() algorithm infers the type of the file based on its extension. Files with extensions ".cer", ".crl" and ".roa" are assumed to be DER encoded binary files of type certificate, CRL and ROA, respectively. Note that CRLs must have an extension of ".crl" and ROAs must have an extension of ".roa". Certificates should have an extension of ".cer", but this is not mandatory. Any file without an extension, or with an unknown extension, will be processed as if it were a certificate. This means that if objects other than certificates are placed in the repository, an error will occur when those objects are being processed by the OpenSSL read routines. If the file has an additional extension of ".pem" it is instead assumed to be a PEM armored representation of a DER encoded file of the corresponding type. Note that this inferencing step only looks at the filename; it does not attempt to open the file and apply any kind of heuristic to the contents.

In general any error is logged. The log entry will contain the name of the object being processed, the operation that is being attempted, and a reason for the failure. This allows the users to examine the logfiles to determine why an operation on an object failed.

The next step is to associate the directory part of the pathname to a directory identifier (the reason why directory identifiers are used is explained below in the database design section). The directory is looked up in the directory table of the database. If it is already present, its id is returned. If it is not already present, a new entry is made for that directory, and its id is returned. A switch is now made based on the inferred file type, and one of three subordinate algorithms is called: add_cert(), add_crl() or add_roa().

The add_cert() algorithm performs seven steps. Its pseudocode looks like:

```
If not cert2fields() return error;
Set some flag bits based on cert fields and user input;
If not rescert_profile_chk() return error;
If not verify_cert() return error;
If cert_revoked() return error;
If not add_cert_internal() return error;
verify_children()
```

The subalgorithm cert2fields() uses parts of the OpenSSL library to extract the following fields from the certificate: the subject, the issuer, the serial number, the SKI, the AKI, the SIA, the AIA (optional for trust anchors), the CRLDP, the basic constraints extension, the RFC3779 extension field (as a binary blob) and the notBefore and notAfter fields. If this extraction fails an error is returned. (The extraction could fail because, for example, the putative certificate file is not a valid X509 certificate file, has bad ASN.1, has a missing field that is mandated by the profile, or any number of syntactic failure causes.) Note also that other fields from the certificate, such as the key usage extension and the certificate policies extension, are also extracted subsequently (during verify_cert()). The purpose of cert2fields() is solely to extract those fields that will be used to populate the database.

The next step in add_cert() is to possibly set bit fields in the flags field for the certificate. The only bit that can have already been set is the FLAG_CA bit, which would have been set in cert2fields() if the certificate was a CA certificate. Add_cert() compares the subject and issuer fields, and if they are identical it sets the FLAG_SS (self-signed) bit. Add_cert() then checks an argument that was passed in on the command line, and if this argument is present it sets the FLAG_TRUST bit, indicating that this is a trust anchor. It is an error to attempt to set this bit if the FLAG_SS bit is not also set. This convention implies that from the rcli command line one can declare that a certificate is a trust anchor. There is no corresponding mechanism for rsync_aur to make such an assertion. The implication is that it is **assumed** that the database has been manually seeded with the trust anchor certificates using the command line form of rcli, and that as new trust anchors are issued or deleted, the user must manually update the database with the new trust anchor information.

The next step is to call the profile validation algorithm rescert_profile_chk(). This subalgorithm checks a variety of aspects of the certificate to insure that it conforms to the resource certificate profile. Specifically, it checks to see that the basic constraints extension is present and marked critical; that the SKI is not marked critical; that the AKI is not marked critical, that its keyIdentifier is present except in trust anchors, that its authorityCertIssuer is not present, and that its authorityCertSerialNumber is not present;

that the key usage extension is present and marked critical and that it has only keyCertSign and CRLSign set for a CA certificate and only digitalSignature set for an EE certificate; that the CRLDP, SIA and AIA (if present) are marked non-critical; that the certificate policy extension is present and marked critical and that it does not have PolicyQualifiers and that the PolicyIdentifier OID has the value 1.3.6.1.5.5.7.14.2; and, finally, that the RFC3779 extension is marked as critical. If any test fails, an error is returned.

The next step is to call the path validation algorithm verify_cert(). This subalgorithm uses a combination of OpenSSL and database infrastructure routines in order to perform validation of the certificate. The pseudocode for this algorithm is as follows:

```
Initialize stack SU of untrusted certificates
Initialize stack ST of untrusted certificates
C = input certificate
P = parent_cert(C)
While ( P does not have trusted bit set )
    Push P onto SU
    P = parent_cert(P);
    If P is NULL return error
Push P onto ST
Ask OpenSSL to verify(SU, ST, C); if error return error
```

The subalgorithm parent_cert(C) is a simple algorithm. It uses database infrastructure routines to find the set of database entries in the certificate table that have a subject equal to the issuer of C, and, from within that set, the subset of entries that have an SKI equal to the AKI of C. All such certificates are also checked to ensure that they have the valid bit set, and are not included if they do not. If parent_cert() returns zero matches at any point in this algorithm, then validation is deferred, as explained below under verify_children(). If a complete chain of certificates is found that terminates at a trust anchor, then OpenSSL is asked to perform the final validation, and its status returned.

The cert_revoked() subalgorithm checks to see if there are any valid CRLs that are currently revoking this certificate. It does this by querying the database for any CRL with the same AKI and issuer as the certificate. For each such CRL, it checks to see if any serial number in its list matches the certificate's serial number, and if so concludes that the certificate is revoked.

If verify_cert() returns TRUE and also indicates that it found a complete chain to a trust anchor, the FLAG_VALID bit is set in the flags field; if a complete chain was not found then the FLAG_NOCHAIN bit is set instead. At this point the add_cert_internal() subalgorithm is called. This subalgorithm uses database infrastructure routines to construct the appropriate SQL statement that will insert the information for this certificate into the certificate table.

The verify_children() algorithm attempts to verify all those objects whose validity was previously undetermined because this certificate was previously not available and hence there was no chain to a trust anchor. First, it attempts to verify all ROAs for which this certificate is the parent. For each ROA, if it is verified, then the flag is set to indicate that it is valid; otherwise, it is deleted from the database. Next, it attempts to verify all CRLs that are children. If the CRL is not verified, it is deleted from the database. Otherwise, the valid flag is set, and the whole process by which all the specified certificates are revoked is carried out. For each revoked certificate, the subalgorithm revoke_cert_and_children(), which is described in the garbage collection section, is called in order to invalidate the certificate's descendants as appropriate. Finally, it attempts to verify the certificates that are children, deleting those that fail and setting the valid flag of those that succeed. For all newly valid certificates, the verify_children() procedure is performed recursively.

The add_crl() algorithm performs four steps. Its pseudocode looks like:

```
If not crl2fields() return error;
If not verify_crl() return error;
If not add_crl_internal() return error;
revoke_certs();
```

The subalgorithm crl2fields() uses parts of the OpenSSL library to extract the following fields from the CRL: the issuer, the AKI, the serial number list (as a binary blob), the CRL number, and the thisUpdate and nextUpdate fields. If this extraction fails an error is returned. The extraction could fail because, for example, the putative CRL file is not a valid X509 CRL file, has bad ASN.1, has a missing field that is mandated by the profile, or any number of syntactic failure causes.

The next step is to call the validation algorithm verify_crl(). This subalgorithm uses a combination of OpenSSL and database infrastructure routines in order to perform validation of the CRL. Essentially, once the signing cert has been fetched from the database, and is determined to be valid, OpenSSL routines are used to validate the CRL. It is important to note that in this context the word "valid" as applied to the signing certificate has a more relaxed meaning than it did in the case of the full path validation performed during add_cert(). Since every certificate in the database was fully validated at the time it was added, it is only necessary to ensure that the signing certificate is still valid. In this case a reduced form of verify_cert(), known as verify_cert2(), is called. This function verifies that the certificate is in the database, has not expired and has its valid bit set, and that each of its parents is also in the database, has not expired, has its valid bit set, and that this chain terminates in a trust anchor. (A note on terminology: when the word "parents" is applied to a certificate or ROA it refers to the complete upward chain of parent certificate, grandparent certificate, etc, until the terminal trust anchor certificate.)

If verify_crl() returns TRUE, the FLAG_VALID bit is set in the flags field of the CRL being added and the next subalgorithm in add_crl() is called. This is the add_crl_internal() subalgorithm. This subalgorithm uses database infrastructure routines to construct the appropriate SQL statement that will insert the information for this CRL into the CRL table.

The final step is to do revoke_certs(). For each serial number in the list of serial numbers, it attempts to find a certificate in the database with that serial number and the same AKI and issuer as the CRL. If any such are found, it invokes the subalgorithm revoke_cert_and_children(), which is described in the section on garbage collection, in order to delete the certificate and invalidate its descendants.

The add_roa() algorithm performs three steps. Its pseudocode looks like:

```
If not roa2fields() return error;
If not verify_roa() return error;
Return add_roa_internal()
```

The subalgorithm roa2fields() uses the ROA processing library to extract the following fields from the ROA: the SKI, the IP address information and the AS#. If this extraction fails an error is returned. The extraction could fail because, for example, the putative ROA file is not a valid ROA file, has bad ASN.1, has a missing field that is mandated by the profile, or any number of syntactic failure causes.

The next step is to call the validation algorithm verify_roa(). This subalgorithm uses a combination of ROA library, OpenSSL and database infrastructure routines in order to perform validation of the ROA. This process can be broken down into the following components. First, the signing certificate for the ROA is fetched from the database. If this certificate is not present, then the deferred validation algorithm, described above, will be used; the ROA is immediately entered into the database (using add_roa_internal(), below) and is validated later, as described above under verify_children(). If the signing certificate was found, the IP address information and AS# in the ROA is compared with the IP address information in the signing certificate. If this fails to match an error is returned. Next valid_cert2() is called on the signing certificate to ensure that it has remained valid.

If verify_roa() returns TRUE, the FLAG_VALID bit is set in the flags field of the ROA being added and the final subalgorithm in add_roa() is called. This is the add_roa_internal() subalgorithm. This subalgorithm uses database infrastructure routines to construct the appropriate SQL statement that will insert the information for this ROA into the ROA table.

The delete_object() algorithm is the last of the three top level algorithms used by the rsync client. This algorithm calls the infer_filetype() algorithm to infer the type of object whose deletion is requested. If this algorithm cannot identify the file type, then an error is returned. Next, a lookup is performed on the directory portion of the filename. If this directory is found in the directory table in the APKI database, its id is returned. If the directory is not found, the delete_object() function is not found, an error is returned.

Finally, the database lookup on the pair {filename, directory-id} is performed in the database table that corresponds to the file type. If a match is found, it is deleted using a database infrastructure library call. If no match is found an error is returned.

When a valid certificate is deleted, there is one additional step, which is a call to revoke_cert_and_children(), a subroutine described in the section on garbage collection. This invalidates the certificate's descendants.

4. The query client

The query client is a database client program. It performs two types of operations based on command line parameters supplied by the user. It is run manually to generate the ROA output file(s), or to ask generalized questions about the database. The former operation is known as the "comprehensive query." The query client uses the database infrastructure libraries to ask for the set of all ROAs in the database by calling the database infrastructure routine that enumerates the contents of the ROA table. For each ROA in this list, it then must revalidate that ROA using a process similar to the one used by verify_roa(), as described in section 3.

The query client, however, has different requirements than the rsync client. In particular, it must keep track of two bits in the flags field of each of the parents of this ROA, namely the valid bit and the unknown bit. Therefore it must use a different algorithm, verify_roa2(). This algorithm is almost identical to the algorithm used for verify_roa(), except this algorithm takes note if any certificate encountered during its path traversal had its unknown bit set. (The unknown bit is set during garbage collection, as described below.) The complete set of ROAs is thus subdivided into three distinct subsets: invalid ROAs, ROAs for which the unknown bit was not set anywhere in the traversal and ROAs for which the unknown bit was set somewhere in the traversal.

Invalid ROAs are discarded. This action is logged with a reason code so that users can subsequently examine the logfile to determine which ROAs were deemed invalid and why. The contents of the second subset, the unambiguously valid ROAs, are presented to a subalgorithm print_roa() from the ROA library, to be described shortly. The contents of the third subset (the "ambiguous ROAs") are processed conditionally based on command line arguments to the query client. The user may make one of three choices: process the ambiguous ROAs as if they were valid; process the ambiguous ROAs as if they were invalid, or process the ambiguous ROAs specially.

In the first case, the set of ambiguous ROAs is also presented to the print_roa() algorithm. In the second case, the set of ambiguous ROAs is discarded (and logged). In the third case, the set of ambiguous ROAs is presented to print_roa() but with an alternate output filename.

The print_roa() algorithm converts the contents of a ROA into human readable ASCII text. The intent of this routine is to provide a simple output format containing one or more lines of output for each ROA. Each line contains the AS# and one of the IP address blocks contained within the ROA. This implies that a single ROA can produce one or more lines of output. The print_roa() algorithm accepts an argument specifying the output file to which the ROA contents are sent. There are at most two such files. The nominal output file is always present and contains at least the display form of all the unambiguous ROAs. The secondary output file may be present if the user requested special handling for ambiguous ROAs. If present, it contains the display form for all the ambiguous ROAs.

The query client can also be used to ask the database simple questions. Questions such as "what objects are in the database with FIELD=X", where "FIELD" and "X" are both supplied on the command line, can be asked. The user can thus use the query client to determine the set of all certificates in the DB that have been issued by a particular issuer, the set of all CRLs whose nextUpdate field has passed, and other such structured queries. The database infrastructure libraries are used for this activity.

5. The garbage collection client

The garbage collection (GC) client is a database client program. This client is a daemon program that typically runs in the background and performs its actions without user intervention. The GC client has three responsibilities: certificate validity interval checking, certificate expiration processing, and CRL staleness checking.

Certificate validity interval checking uses the algorithm certificate_validity() from the database infrastructure libraries. This algorithm traverses the database and examines the validity interval dates on every certificate. There are three possible outcomes of this interval check: the certificate appears to be ok; the certificate has a notBefore date in the future, and the certificate has a notAfter date in the past, i.e. it has expired.

If the certificate's validity dates are valid, then the subalgorithm certmaybeok() is called. This extremely simple algorithm merely checks to see if the FALG_NOTYET bit is set in the certificate's flags field. If it is set, then it clears this flag and sets the valid flag.

If a certificate has somehow been entered into the database but is not yet valid, the subalgorithm certtoonew() is called. This subalgorithm clears the validity bit in the flags field of the certificate and sets its FLAG_NOTYET bit.

If a certificate has expired, the GC client calls the subalgorithm certtooold(). This subalgorithm immediately calls another subalgorithm, revoke_cert_and_children(). The revoke_cert_and_children() subalgorithm performs the following actions. First, it deletes the expired certificate from the database. Next it iterates (recursively) over all the children of that certificate. For each child it calls the subalgorithm countvalidparents() on the certificate. This subalgorithm is very similar to the parent_cert() algorithm described previously, except that instead of retrieving the parent certificate, it returns a count of the number of possible certificates that could be valid parents of the given certificate. If this returned count is greater than zero then revoke_cert_and_children() does nothing more to that child (or any of its descendents). The purpose of this test is to handle the case where a certificate expired, but one or more of the children of that certificate were reparented. This typically occurs when two CA certificates are issued with overlapping validity dates, have the same subject and issuer and contain the same key.

If countvalidparents() returns zero on a certificate, that certificate is deleted and revoke_cert_and_children() is called on all of its children. Thus, the impact is that whenever a certificate expires it is deleted, and all of its children (and grandchildren, etc) are also deleted unless they have been reparented. Note also that this recursive deletion operation not only applies to entries in the certificate table, it also applies to entries in the ROA table. If the certificate that signed a ROA has been deleted, the ROA must be checked to see if it has been reparented by another certificate; if not, it is deleted too.

Certificate revocation processing is primarily handled during initial loading of the data by the rcli client. As a safeguard, this is done again by the algorithm iterate_crl(). In this algorithm the GC client iterates over all CRLs in the database, calling a helper function for each CRL that is found and that has its valid bit set. This helper function extracts the issuer, AKI and serial number (SN) list of its CRL. For each triple {issuer, AKI, SN} it attempts to find a matching certificate in the database. If any such are found the revoke_cert_and_children() algorithm is then called on the matching certificate. As just described, this will delete the revoked certificate itself and then perform the reparenting check recursively on all its children (and grandchildren, etc) – both certificate children and ROA children.

CRL staleness checking is performed by the algorithm stale_crl(). In this algorithm the GC client again iterates over all CRLs in the database. If a CRL has a nextUpdate field that is in the past the CRL is said to be "stale." If a CRL is stale, then all certificates to which it might apply, that is all certificates that match its {issuer, AKI} fields, are marked as "unknown" by setting their FLAG_UNKNOWN bit. Note that this operation does not clear the valid bit on these certificates. Conversely, if a CRL is not stale, then all certificates to which it might apply are checked to see if they do have the unknown bit set. If so, this bit is cleared.

The GC client may also be invoked manually, on the command line, to perform an immediate garbage collection.

The GC client makes heavy use of the database infrastructure libraries.

6. The chaser client

The chaser client is a database client program. The purpose of the chaser client is to gather a set of URIs that rsync might not currently know about and tell rsync to also synchronize the repository with those URIs in addition to the ones it is already using for synchronization. The reason this needs to be done is twofold: to synchronize the local repository with newly discovered external repositories, and to refresh the set of CRLs from their distribution points with the intent of eliminating stale CRLs in the DB.

The chaser client uses the following series of steps. It first gathers the values of all non-empty SIA, AIA and CRLDP fields from the certificate table in the DB. In then sorts this list of URIs to eliminate any that are not rsync URIs (this is the second dependency on rsync that was mentioned earlier). It then culls this list to eliminate duplicates. It then examines the set of URIs that it obtained from CRLDPs. If any such URIs refer to a directory, a pattern "/*.crl" is appended to ensure that only CRL files within that directory are fetched. It next further culls this list to eliminate file or directory URIs that are implied by a URI that defines an initial segment. For example, if both rsync://a/b/c and rsync://a/b/c/d/e are on the list the latter is deleted since it will inevitably be fetched by rsync when the former is fetched. This final culled list is then compared against the default set of rsync URIs, stored in rsync's default configuration file (or an alternate configuration file provided on the command line). The chaser client then generates an output file containing those URIs that are on its internally generated list but are not on rsync's list. The chaser client then invokes the rsync wrapper with this new (delta) configuration file as a command line argument.

Note that it is currently possible for the chaser client to fetch the same object more than once. This can occur if the same object is stored in more than one remote repository, and if those objects end up being stored to different destination directories in the local repository. Note that this is not an error, but is inefficient.

7. The APKI database

The APKI database is a MySQL database that holds pertinent information about managed digital objects. Any object will have been validated before being inserted into the database. Note that the database does not hold the entire contents of any object. Instead, it holds abstracted information about those objects together with information on where the actual underlying object (file) may be found in the repository. A design decision was made to not put the entire object in the database for the sake of efficiency, since a substantial part of each object will not be used by any of the DB clients once that object has actually been entered into the DB.

The database consists of five tables: the certificate table, the CRL table, the ROA table, the directory table and the metadata table. The certificate table contains various fields derived directly from the contents of the certificate, including the subject, issuer, serial number, SKI, AKI, SIA, AIA, CRLDP and the notBefore and notAfter fields. It also contains a binary field (known as a blob) containing the RFC3779 information block extracted from the certificate. It also contains other fields that describe the certificate, including a {filename, directory-id} pair that can be used to locate the certificate file itself; a flags field that contains the valid, nochain, unknown, CA, SS and notyet bits; a unique local identifier that is generated when the certificate is first inserted into the DB; and a timestamp that is also set when the certificate is first inserted.

The CRL table contains the issuer, AKI, thisUpdate, nextUpdate and CRL number fields from the CRL. It also contains a binary blob containing the serial number list from the CRL. It also contains a {filename, directory-id} pair of fields used to locate the CRL file; a unique local identifier field; and a flags field. In the case of a CRL only the valid bit in the flags field is used.

The ROA table contains the SKI and AS# fields from the ROA. It also contains a {filename, directory-id} pair of fields used to locate the ROA file; a unique identifier field and a flags field. In the case of a ROA only the valid bit in the flags field is used.

The directory table contains directory-name and directory-id fields. This table is used in combination with any of the three previously described tables. Its presence is based on efficiency considerations. Any digital object must be associated with exactly one file in the repository. This file has a unique fully qualified pathname consisting of a directory and a filename. It may be desirable to search in the database based on the full pathname; however SQL searches based on long strings are not efficient. It is more efficient, therefore, to separate the full pathname into a filename and directory-identifier pair, and put the mapping between directory-identifiers and directory names in a separate table. It might seem counterintuitive that this is more efficient, since it would appear that two table lookups (and a string concatenation) are required in order to derive the full pathname, but in fact in SQL this type of indirect lookup can be directly supported in a single query.

The metadata table contains information about the APKI database itself. It contains information about when each of the clients was last run; the name of the top level repository directory; and the maximum values of the unique identifiers used in each of the three main tables.

A complete definition of the database tables in SQL syntax can be found in the file "scmmain.h".

8. The database infrastructure libraries

The APKI database is accessed via SQL statements. A large part of SQL programming involves a fairly repetitive set of operations that have a significant amount of common code. In addition, SQL programming through a standard interface library, a so-called Open DataBase Connector (ODBC), is a fairly specialized knowledge domain. For all these reasons it was deemed good programming practice to encapsulate access to the database through a set of interface abstractions. As a result, two database libraries were created that all four clients use. The lower level library is known as sqcon and the higher level library is known as sqhl.

The sqcon library contains abstractions for common database operations. Thus, it contains functions that insert entries in tables, delete entries from tables based on matching criteria, search for entries based on matching criteria, and so forth. This layer also contains functions for manipulating the metadata associated with table entries, such as finding the next available unique id for a row that is about to be inserted, and also manipulating the flags field of an entry.

The sqhl library contains the implementations of the actual algorithms used by the clients. For example, all the algorithms contained within sections 3-6 of this document are contained in the sqhl library.

9. The ROA processing library

The ROA processing library is a collection of functions that performs three tasks: generation of ROAs from an ASCII configuration file; parsing and validation of ROAs; and generation of the ROA output format from a validated (or declared-to-be-valid) ROA.

The ROA generating functions take an ASCII input file similar in form to an OpenSSL configuration file, validate the contents of that file, and generate a ROA as output. ROA generation is primarily a test tool for the APKI.

The ROA parsing and validation functions must perform for ROAs the same set of operations that the OpenSSL library performs for certificates and CRLs. That is, these functions must verify that the information in the putative ROA file is valid ASN.1; that this ASN.1 conforms to the CMS definition of a ROA; that the signing certificate can be located in the DB based on the SKI found in the ROA (and therefore implicitly that this certificate is itself valid); that the ROA's signature is valid; and that the IP address and AS# in the ROA matches the corresponding information in the signing certificate. The interface to the ROA library was specified to be as independent of the DB infrastructure libraries as possible; some dependencies must exist, however, in order to fetch the signing certificate and perform signature verification.

The final task of the ROA processing library is to supply an output function that converts the IP address information and AS# information in a ROA into a user-friendly output format. At a very early stage it was decided not to support direct output in standard BGP filter syntax, as this syntax is somewhat complex. Instead, a decision was made to output the ROA information in a "lowest common denominator" format that each individual user installation could postprocess as it desired. The format chosen was ASCII text. Each ROA processed during a comprehensive query will result in one or more lines in one of the output file(s) generated by the query client. (Recall that the query client can be directed to generate either one or two output files – the "standard" output file for fully validated ROAs, and the "special" output file for ROAs that have certificates with the unknown bit in their validation path). Each line will contain an AS#, an IP address block, and an identifier for the ROA (currently, the SKI of the ROA).