Validation Algorithms for a Secure Internet Routing PKI

David Montana and Mark Reynolds

BBN Technologies
10 Moulton Street, Cambridge, MA 02138 USA
dmontana@bbn.com,mreynolds@bbn.com

Abstract. A PKI in support of secure Internet routing was first proposed in [1] and refined in later papers, e.g., [2]. In this "Resource" PKI (RPKI) the resources managed are IP address allocations and Autonomous System number assignments. In a typical PKI the validation problem for each relying party is fairly simple in principle, and is well defined in the standards, e.g. RFC 3280 [3]. The RPKI presents a very different challenge for relying parties with regard to efficient certificate validation. In the RPKI every relying party needs to validate every certificate at fairly frequent intervals (e.g., daily). In addition, certificates on the validation path may be acquired from multiple repositories in an arbitrary order. These dramatic differences motivated us to develop performance-optimized validation algorithms for the RPKI. This paper describes the software developed by BBN for the RPKI, with a special focus on this optimized validation approach.

Key words: Border Gateway Protocol, Resource PKI, Internet Routing PKI, Route Origination Attestation

1 Background

The Border Gateway Protocol (BGP) [4] is a critical routing protocol in the Internet. Routers exchange Autonomous System (AS) path information between themselves using BPG UPDATE messages. Unfortunately the current implementation of the BGP protocol does not provide any method for determining if such path information is valid. Path information may be invalid due to configuration errors, or, due to malicious BGP spoofing [5, 6].

Several proposed alterations to BGP provide for additional security to the path information [7, 8, 9, 10, 11]. All are predicated upon the existence of some form of PKI that binds AS# and IP-address block resources to the entities to which they have been allocated. These proposals have not been adopted due to the changes required to routers and the infrastructure requirements imposed. The most recent proposal for creating the requisite infrastructure is described in [1], an approach based on a new, digitally signed object, the Route Origination Attestation (ROA), together with a PKI to validate, manage and process such objects. Relying party software for use with this "Resource" PKI (RPKI) was

implemented by BBN, and is described in this paper. Of particular interest is the set of validation algorithms that were developed for the RPKI.

In a typical PKI [12] the validation problem for each relying party is fairly simple in concept, although it may be complex in practice. Typically a relying party receives an End Entity (EE) certificate which must be validated prior to verifying the signature on an object. The relying party may be provided with additional Certificate Authority (CA) certificates needed to complete the certificate path to one or more Trust Anchors (TA) employed by that relying party. The validation of a certification path from a TA to a EE certificate, including processing of revocation status data, is well defined specified in standards. The non-standard part of the process is the discovery of a suitable certificate path.

Given this typical task for a relying party, strategies for optimizing the performance of certificate validation have been developed. They are based on the assumption that a relying party will, within a reasonable time interval (say, 24 hours), validate only a very small fraction of all the certificates issued in the context of the specific PKI. This is a reasonable assumption for most PKI applications, e.g., secure email provided via S-MIME, VPN security using IPsec, or secure web access via TLS/SSL [13].

The RPKI presents a very different challenge for relying parties with regard to certificate validation. In this RPKI it is anticipated that every relying party (e.g., every participating ISP) will need to validate every certificate within (roughly) a 24 hour interval. This dramatic difference in validation methodology motivated the development of a novel performance-optimized approach to certificate validation. This paper focuses on the validation algorithms developed as part of our work on implemented the RPKI. Our approach to validation makes use of a relational database containing only validated signed objects (e.g., certificates) to avoid duplicative validation processing.

2 RPKI Software

The goal of the BBN RPKI (relying party) software is to process data from repositories operated by the five RIRs (and their subordinate certification authorities) in order to create a single set of text files that can be used to generate BGP filters. These filters describe which Autonomous Systems are authorized to originate routes for specified IP address prefixes. The BBN RPKI software system finds all the available ROAs, and their associated certificates and certificate revocation lists (CRLs), verifies the ROAs using these certificates and CRLs, and creates simple text files containing AS#/IP prefix pairs that can be used to create BGP filters.

As noted above, this process is different from the traditional use of a PKI to support applications. Previously, an application received one or a small number of signed objects that required validation. The application would gather the certificates required to form a certificate chain to a Trust Anchor and then perform the appropriate certificate path checks. Our application needs to determine the validity of all the available ROAs, certificates and CRLs across multiple repos-

itories. To do this quickly and with efficient use of system resources requires a different approach than that of PKI software supporting a traditional application, since it will require processing the same certificates and CRLs many times. Therefore, our software extracts relevant information from local copies of certificates and CRLs, and creates relational database records containing that information. This approach provides a simple, efficient, and highly structured way to access the applicable information for each object without having to repeatedly acquire and process certificate and CRL data. We first discusses how the data is stored locally, and then describes the different programs that operate on the data and transform it, eventually resulting in output files that can be used to create BGP filters.

Our software suite consists of a set of programs that typically run continuously, in the background, rather than a set of programs that are executed once from the command line. This is done for the sake of efficiency; as the ROA's validation state changes over time; our software updates files incrementally, so as not to require reprocessing all the data.

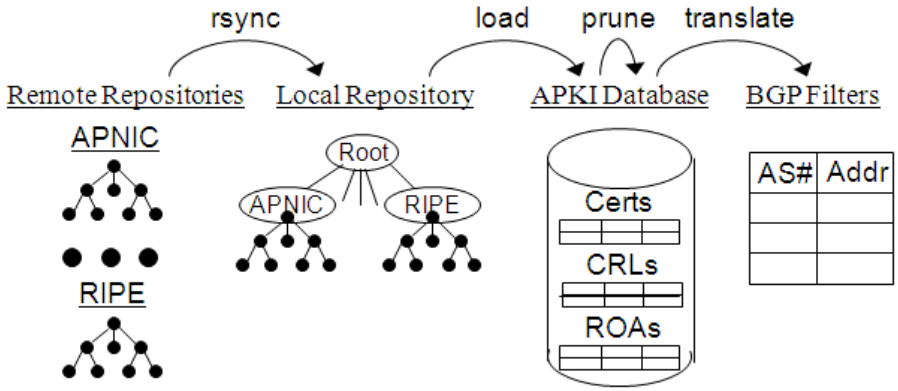

Fig. 1. Transforming the data

The different forms of the data and the transformations between them are shown in Figure 1. We now describe each component in the process.

Remote Repositories - The raw data (certificates, CRLs, and ROAs) are stored as files on a distributed system of servers provided by the CAs that comprise the RPKI. Each of the five RIRs will attempt to cache data from its subordinate CAs on its repository server, so it is expected that our software will be able to find most of the data it needs on the RIR servers. However, there will be some cases where the data is not cached at a RIR and the application will need to retrieve data directly from a subordinate CA repository. Data from these repositories can be located using the Subject Information Access (SIA) extensions present in RPKI certificates [14].

Local Repository - Our application retrieves files from repository servers and caches them locally. It uses rsync, an existing freely available utility, to keep the local repository synchronized with the remote repositories in a directory structure that mirrors the originals. Because most of the files retrieved from the remote repositories will need to be accessed many times as part of ongoing processing, it is more efficient to copy these files once (particularly with the efficient copying of rsync) rather than to continually access the remote versions. Furthermore, rsync outputs a record of what changes occurred in which files, which allows our application to process only the modifications, and thereby minimize the work done with each incremental data update.

Relational Database - MySQL [15] was used as the relational database underpinning our relying party software. The database has three main tables, one for each type of object of interest: certificates, CRLs and ROAs. Each object type has a different set of data fields in addition to its signature. The database includes only those fields required to search for and/or identify the different objects/rows in the tables. (It is easier and more efficient to leave seldom-used information in the corresponding file.) For example, some of the fields included in the certificate table are: subject key identifier (SKI), subject, authority key identifier (AKI), and issuer. This supports an SQL query requesting the certificate with a particular SKI and subject, or a query requesting all certificates with a particular AKI and issuer. (The significance of these queries is discussed in the section on validation algorithms.) Such database queries allow rapid location of objects of interest.

The database does not include all objects. If an object has been determined to be invalid, that object is either not loaded into the database, or is deleted from the database if it had previously been valid or awaiting validation. Note that the underlying file remains in the local repository until its remote copy has been deleted. (Henceforth we will always use the term "repository" to denote the local collection of files, unless otherwise noted.) An object can be determined invalid for a variety of reasons including expiration, revocation or a signature that does not verify. An object will be placed into the database if it is valid, or if there is not yet enough information to determine whether or not it is valid. The most common reason for an inability to determine an objects validity is when there is a missing link (ancestor) in the certificate path to a Trust Anchor. An object of undetermined validity stays in the database until it expires (and hence is know to be invalid) or it is deleted from its remote repository (and hence also from the local repository).

There is one column in each object table containing the validation state. This field can have three possible values: validated, awaiting-validation, and CRL-stale. Validated and awaiting-validation represent the obvious meanings; CRL-stale is explained next. Each CRL has a field next-update that tells the latest time to expect an updated version of the CRL, which may enumerate a potentially different set of revoked certificates. When the current time becomes later than the next-update time for a CRL, then any *sibling* certificate of this CRL (i.e., any certificate that shares the same issuer and hence could be revoked,

as discussed below) enters an uncertain state. This uncertain validity extends to all its *descendants*, i.e., those certificates and ROAs whose certification path includes this certificate. All sibling certificates and descendants have their validation state set to CRL-stale until the expected update to the CRL arrives.

BGP output files - The goal of the application is to derive this set of files, which are just text files consisting of IP address prefixes and their associated AS numbers. These pairs are specified in the ROAs and our software accumulates the pairs from all validated ROAs. An issue is what to do with those ROAs in the CRL-stale validation state. The application provides the user with three options: include them, exclude them, or put their pairs in a separate table/file where a human can decide how to handle them. (This last option is the default.)

The software is composed of a set of application programs that operate on the data. There is a natural sequential order to these programs which mostly follows the sequence of data transformations shown in Figure 1. Typically, one would expect them to be run in sequence at least once a day. However, there may be times when the programs need to be run separately, which is also possible. The program components are described next.

Synchronizer - This program executes rsync for each of the five top-level RIR repositories in order to create a local repository copy that contains the same data as the remote repositories. The program rsync is an open-source utility that efficiently synchronizes files and directories between two systems. The local repository has at least five parallel subtrees, one for each of the remote repositories. The synchronizer can synchronize with remote repositories other than these five main ones and generate additional subtrees if it is known beforehand (or discovered subsequently) that the data from some RPKI CAs is not cached at one of the five RIRs. The rsync program outputs messages about everything it does, including all files that it has added, updated or removed; the synchronizer saves this output to a log file to tell the loader which files need to be handled.

Loader - This program looks at the logs generated by the synchronizer and loads the data from all new or modified files into the database; it also deletes the data corresponding to those files that have been removed. Before the loader puts a record corresponding to a new object into the database it checks whether this object is invalid and refrains from loading such files. If the object can be validated, it sets the validation flag for that object, and then determines which other objects already in the database need to have their validation state changed, or need to be deleted, as a consequence. This process of validating an object and then determining the effects of this change on other objects in the database is discussed in detail in Section 3. When there are multiple remote repositories, the loader is executed in parallel with the synchronizer. After the synchronizer is done updating from one repository and has proceeded to a second repository, the loader can be loading the data from the first repository. Because both the synchronizer and loader spend much of their time doing input/output, parallel execution provides true speedup.

Pruner - This program looks for changes in the state of objects due to the passage of time, in particular certificates and CRLs that have expired (but which were valid at the time they were loaded). Finding expired objects in the database is easy, requiring only two database queries, one for certificates and one for CRLs. Propagating the effects of an expired certificate follows the process described in Section 3.

Chaser - This program is invoked when not all the certificates are cached at the RIR repositories, and hence accumulating all the required data necessitates retrieving data from lower-level CA repositories. Each certificate has a field that points to the location of its parent (the Authority Information Access or AIA), one pointing to sibling certificates (the Subject Information Access or SIA), and one pointing the CRL in which the certificate will be listed if revoked (the CRL Distribution Points or CRLDP). The chaser accumulates a list of alternative repositories from which to retrieve data and executes the synchronizer and loader on these repositories in the same way they are initially executed on the RIR repositories. Currently there is no way to check which objects from these lower-level repositories have already been cached by an RIR. In this case those objects will already have been copied and processed locally. When the duplicates are copied from a lower level CA repository, the fact that it they are duplicates will be noted, and they will not be entered into the database twice.

Translator - This utility creates the BGP filter files. The primary content of a ROA is a mapping from IP address prefixes to AS numbers; the translator combines the data from all the validated ROAs into a single large list. Note that the translator is part of a more general query tool for extracting information from the database, which allows easy visibility into the database for the expert user.

BBN has developed, integrated, tested, and released this software as a free, open source project. It may be downloaded from [16].

Work on the software is ongoing. In particular, the evolution of the resource certificate project [17] has grown to encompass a new type of object: a file manifest. BBN is currently working on extensions to the RPKI software to process manifests, as well as integrating them into the overall database architecture. (A manifest is a list signed by an EE representing a CA, enumerating all of the files currently published under that CA. The manifest will be used to detect unauthorized deletion or substitution of (older, valid) files in a repository.) BBN also plans to continue work on improving the efficiency of the software suite. As described below, there are cases in which caching a previously computed validation result can result in a significant performance improvement. In addition, the chaser (and the entire rsync transfer process) can be made more efficient by adding a mechanism for determining which objects in which repositories have already been retrieved, so that duplicate files are transferred less frequently. A number of other performance optimizations are also being considered. Finally, as standards for secure internet routing continue to evolve [18], we anticipate that the RPKI software will continue to be improved to track those changes.

3 Validation Algorithms

The process of keeping the (local) database current with respect to the validation state of all objects (certificates, CRLs, and ROAs) has two main parts. The first is validating or invalidating individual objects. The second is propagating the consequences of an object's change of validation state throughout the database. We discuss each of these steps, in turn, after providing a quick overview of the various objects relationships.

The primary relationship between objects is the parent-child relationship. The parent object is always a certificate. If the child is a certificate or a CRL, the parent is the certificate of the CA that signed the child. For a ROA or a manifest, the parent is an EE certificate. Hence, a parent certificate is required to validate an object. For the signature to be acceptable, the parent must itself be validated, and so a certification path is required back to a Trust Anchor. A Trust Anchor is a self-signed certificate that is inherently trusted; the default Trust Anchors in the RPKI are self-signed certificates issued by the RIRs and by IANA.

The parent-child relationship between two objects is dictated by two simple rules:

- A certificate is the parent of another certificate or of a CRL if the parent's SKI equals the child's AKI and the parent's subject equals the child's issuer.
- A certificate is the parent of a ROA (or a manifest) if the two objects have the same SKI.

The sibling relationship between CRLs and certificates is also important, because a CRL can revoke any certificate that is its sibling, i.e., has the same parent. A certificate is the sibling of a CRL if the two objects have the same AKI. A validated CRL revokes a sibling certificate if the serial number of the certificate is in the list of serial numbers of the CRL.

Having a relational database makes it easy and fast to find objects that satisfy these relationships. For example, the following single SQL query produces all the certificates that are the children of a certificate with SKI=foo and subject=bar:

SELECT id, ski, subject FROM certificates WHERE aki="foo" AND issuer="bar";

We now discuss the different ways that an object can change its state as part of single object validation. These are changes in validation state that are not the result of some other object changing state. Such initial events can potentially start a chain reaction of validation state updates.

When the loader acquires a new or modified object, because the synchronizer has added or modified a file in the local repository, it performs a set of checks to determine if it should create an object corresponding to that file and add it to the database. The loader performs the following checks:

- If the syntax of the object does not follow the specifications for that type of object, the object is not added to the database.

- If the object is a certificate or ROA and has expired, it is not added to the database.
- If a certificate is revoked by a validated CRL already present in the database, then the certificate is not added to the database.

Before loading an object into the database, the loader queries the database to determine if a validated parent of the object is present. If no such parent exists, the object is written to the database and put in the awaiting-validation state, for subsequent processing by the *deferred validation algorithm*, described below. If a validated parent does exist, then the object can be tested against a potential certification path. If the parent's key validates the object's signature, then the object is added to the database and marked as validated, potentially starting a chain of validation updates as described in the next section. If the parent's public key is inconsistent with the object's signature, then the object is not added to the database.

If the synchronizer removes a file that corresponds to an object that is still in the database, the loader removes the corresponding object from the database. This can possibly start a chain of validation updates. If the pruner determines that a ROA or certificate has expired, it deletes the object from the database. When a certificate is deleted, this can potentially start a chain of validation updates. If a CRL expires, then the CRL is placed in the CRL-stale validation state, and this can also start a chain of updates.

Note that once a potential path to a Trust Anchor exists, the actual validation of certificates and CRLs is done using existing publicly available software (OpenSSL [19] and cryptlib [20]) that performs the certification path validation checks and verifies the signatures all the way back to the Trust Anchor. An approach that cached the validation state could offer a potential performance improvement, since all but the final link in the chain has already been validated at that point. Note that this potential inefficiency does not exist for ROAs, since we had to write custom code to validate them.

When an object changes validation state this change can propagate through the database. Because of the need for certification path validation, a new certificate being validated or invalidated can ripple to its descendants, i.e. all those objects that use the certificate as part of the path used to determine its own validation state. The algorithms for propagating validation state efficiently and incrementally, including the deferred validation algorithm, are described below.

There are four conditions that can lead to propagation of validation state through the database. These conditions along with the actions they require are:

1. If a certificate moves from the awaiting-validation state to the validated state, then all of its children are tested to see whether they are now valid. (Note that the certificate cannot even reach the awaiting-validation state unless syntax checks on the certificate have already been successfully performed; therefore the checking being refer to here has to do with signature verification only.) Generally, these children will be in the awaiting-validation state until the missing parent is validated. Testing the signature of a child

against the newly valid parent's key either proves the child valid, and hence changes its state to validated, or proves it invalid, and causes it to be removed from the database

2. If a previously valid certificate is deleted or invalidated, then each of its child objects is declared invalid unless the child object has been re-parented by another certificate (during key rollover, for example). If the parent is deleted or its new state is awaiting-validation, then the state of each child is modified to be awaiting-validation. If the parent's state is CRL-stale, then the state of each child is modified to be CRL-stale.
3. If a CRL is validated, then each of its sibling certificates is tested to see if its serial number is in the CRL's list of revoked certificates, and if so, the certificate is revoked. If the newly valid CRL replaces one that is stale, then each of its sibling certificates should be removed from the CRL-stale state.
4. If a validated CRL becomes stale, its sibling certificates are placed in the CRL-stale state.

Conditions 1 and 2 constitute the core of the deferred validation algorithm. In a typical PKI path discovery propagates *upward*, from a child object to its parent objects. In the RPKI, path discovery propagates *downward*, from a parent object to its children, when new objects arrive. This type of validation is essential for the operation of the RPKI, since ultimately all objects must either be validated or invalidated, even though their order of arrival in the local repository is completely arbitrary. These operations are complicated by the fact that objects can interact with one another. We provide an example scenario of such propagation below.

Type	**ID**	**SKI**	**AKI**	**Serial #**	**Expires**	**Validation State**
Cert	1	AB:00	AB:00	123	31-DEC	Validated
Cert	2	13:B5	C1:8D	2	01-FEB	Awaiting
Cert	3	EE:23	C1:8D	345	01-FEB	Awaiting
Cert	4	7D:62	13:B5	3	01-FEB	Awaiting
Cert	5	28:2C	EE:23	7	15-JAN	Awaiting
ROA	1	EE:23			01-FEB	Awaiting
ROA	2	28:2C			01-FEB	Awaiting

Table 1. Initial state of example data on 01-JAN

The following scenario contains far fewer objects than would be in a real system, with the quantity of objects limited for the purposes of illustration. However, it is otherwise realistic, and is similar to scenarios we used for initial tests of our software. The initial state of the database is provided in Table 1. It does not show all the database fields, just those critical to determining validation state propagation. Certificate 1 is the single Trust Anchor and initially is also the only validated object, since the others do not yet have a certification path back to the Trust Anchor.

Type	ID	SKI	AKI	Serial #	Expires	Validation State
Cert	1	AB:00	AB:00	123	31-DEC	Validated
Cert	2	13:B5	C1:8D	2	01-FEB	Validated
Cert	3	EE:23	C1:8D	345	01-FEB	Validated
Cert	4	7D:62	13:B5	3	01-FEB	Validated
Cert	5	28:2C	EE:23	7	15-JAN	Validated
ROA	1	EE:23			01-FEB	Validated
ROA	2	28:2C			01-FEB	Validated
Cert	6	C1:8D	AB:00	23	01-FEB	Validated

Table 2. Missing link in trust chains arrives on 02-JAN

Table 2 shows the state after a new certificate arrives. This certificate provides the missing link in the certification paths for all the objects. Using the deferred validation algorithm described earlier, the validation state propagates from the new certificate first to its children and then to their children and so on.

Type	ID	SKI	AKI	Serial #	Expires	Validation State
Cert	1	AB:00	AB:00	123	31-DEC	Validated
Cert	2	13:B5	C1:8D	2	01-FEB	Validated
Cert	3	EE:23	C1:8D	345	01-FEB	Validated
Cert	4	7D:62	13:B5	3	01-FEB	Validated
ROA	1	EE:23			01-FEB	Validated
ROA	2	28:2C			01-FEB	Awaiting
Cert	6	C1:8D	AB:00	23	01-FEB	Validated

Table 3. Certificate 5 expires and its child ROA is invalidated on 15-JAN

Table 3 shows the state after certificate 5 expires. Certificate 5 is deleted from the database, and its child, ROA 2, is invalidated and placed in the awaiting-validation state.

Table 4 shows the state after a CRL arrives. This causes one of its sibling certificates, certificate 2, to be revoked, and its child, certificate 4, to be placed in the awaiting-validation state.

Table 5 shows the state after CRL 1 expires. Certificate 3 is a sibling of CRL 1 and is therefore placed in the CRL-stale state. This propagates to its child, ROA 1, which is also placed in this state.

4 Testing and Experimentation

While large-scale testing and experimentation with the application has been limited, we have been able to do some testing with real data. The RIRs have

Type	ID	SKI	AKI	Serial #	Expires	Validation State
Cert	1	AB:00	AB:00	123	31-DEC	Validated
Cert	3	EE:23	C1:8D	345	01-FEB	Validated
Cert	4	7D:62	13:B5	3	01-FEB	Awaiting
ROA	1	EE:23			01-FEB	Validated
ROA	2	28:2C			01-FEB	Awaiting
Cert	6	C1:8D	AB:00	23	01-FEB	Validated
CRL	1		C1:8D	2,33	20-JAN	Validated

Table 4. CRL 1 arrives on 16-JAN, revoking certificate 2 and invalidating its child.

Type	ID	SKI	AKI	Serial #	Expires	Validation State
Cert	1	AB:00	AB:00	123	31-DEC	Validated
Cert	3	EE:23	C1:8D	345	01-FEB	CRL-stale
Cert	4	7D:62	13:B5	3	01-FEB	Awaiting
ROA	1	EE:23			01-FEB	CRL-stale
ROA	2	28:2C			01-FEB	Awaiting
Cert	6	C1:8D	AB:00	23	01-FEB	Validated
CRL	1		C1:8D	2,33	20-JAN	CRL-stale

Table 5. CRL 1 expires on 20-JAN, causing other objects to enter the CRL-stale state.

cooperated to supply data in order to test our software. All certificates and CRLs from the five RIRs and their subordinates have been cached on a single server, with the cache updated intermittently. Noticeably absent from this data are any ROAs, since ROAs are objects that have only recently been defined and therefore are not currently used by ISPs as a means to express origin AS authorization. Therefore we have generated our own ROAs in compliance with the current specification [21].

There are two different scenarios of interest from the viewpoint of performance. The first is the initial synchronization and loading, when the software starts from a clean state and does a full read of all the data. The second is an incremental update, where the software starts with a local repository and database that reflects the state at the time of last execution, and then reads only the changes to the current state. The amount of work required for an incremental update depends on the number of objects added, modified, or deleted since the previous update, which in turn depends in large part on the time since the previous update. We anticipate that an update will be performed roughly once a day; since the number of objects updated in this time period is typically only a fraction of the total number of objects, we focused on the initial synchronization and load as the performance bottleneck. We also evaluated the performance of the load operation when cryptographic validation is omitted, in order to give us an estimate of how much time was spent in validation.

Our test for the initial synchronization and load for the non-ROA objects involved 22,633 certificates and 10,528 CRLs. The total time required was 20 minutes and 2 seconds. This is divided into the time for synchronization with the remote repository, which required 426 seconds (about 3.1 minutes), and the time for the load, which required 776 seconds (12.9 minutes). The synchronization time depends strongly on the network throughput, while the load time depends on the speed of the local computer. Note that if we had broken the data into five separate remote repositories, 80% of the time for synchronization could have been executed in parallel with the load, hence reducing the time by around 340 seconds (about 5.6 minutes). We anticipate that an incremental synchronization and load would be well under a minute, although we were not able to test this because the repository was not being updated when we performed our experiments. Without cryptographic validation, the time for the load was 226 seconds (about 3.7 minutes); since the average validation path was three or four certificates long, we could have saved roughly 400 seconds (about 6.6 minutes) by validating only the last link in the chain (knowing that the remainder of the chain has already been validated).

We generated 10,000 ROAs to be loaded (but not synchronized because we stored them locally). We do not know how many ROAs will be in real system if this approach is adopted, but this should be within an order of magnitude of the actual number. In a real system, after the RIR and occasional NIR tier, one would expect each CA certificate to be accompanied by at least one EE certificate and on ROA. The number 10,000 is consistent with our 22k certificates and 10k CRLs, on the basis of this reasoning. Loading these ROAs required 281 seconds (about 4.7 minutes).

The remaining question then is how long it takes to generate a file containing the BGP output values specified by these 10,000 ROAs, since this operation will need to be performed every day. The answer is 297 seconds (about 5 minutes). We have already identified a potential modification to the ROA validation step (involving putting more of the ROA data fields in the database), however, and anticipate that this time can be dramatically reduced. Initial experiments indicate that if this proposed modification is implemented, the total amount of time to process all 10,000 ROAs could be reduced to less than ten seconds.

It is worthwhile to note that an alternate implementation of RPKI validation has been carried out at ISC [22]. This code contains an implementation of validation as an integrated part of rsync, their combined version being called "rcynic". It also contains a variety of Python scripts for manipulating certificate and CRL information in a database. While this might appear similar to our RPKI software, BBN's focus has been on generating an end-to-end solution that is optimized for ROA validation and BGP output file generation, and thus we believe that the two are not directly comparable.

5 Conclusion

This paper has described BBN's implementation of a software suite for a resource PKI in which the resources are certificates, CRLs and, most importantly ROAs. The RPKI software performs all the syntactic and semantic validation steps necessary in order to arrive at a set of trusted AS# to IP-address block assignments that can be used to generate BGP filters. In the course of creating the RPKI software, a novel deferred validation algorithm was developed. The algorithm was optimized for the "validate everything" paradigm of the RPKI. Performance testing indicates that even for very large repositories it will be possible to perform a complete filter generation run on a daily basis. The RPKI continues to evolve as aspects of the RPKI itself evolve.

Acknowledgements

The authors would like to thank Dr. Stephen Kent of BBN, who developed the RPKI concept, and whose guidance was critical to the accomplishment of this project. The authors would also like to thank George Michaelson of APNIC for hosting the BBN RPKI software.

The BBN RPKI software was written by Charlie Gardiner, Joshua Gruenspecht, Peiter "mudge" Zatko, Marla Shepard, and the authors, under the management of Karen Seo.

The development of BBN's RPKI software was partially funded under U.S. Government contract FA8750-07-C-0006. However, the opinions stated in this paper are solely the responsibility of the authors.

References

1. Kent, S., Lynn, C., Seo, K.: Design and Analysis of the Secure Border Gateway Protocol (S-BGP). In: IEEE DISCEX Conference, 2000.
2. Kent, S.: An Infrastructure Supporting Secure Internet Routing. In: EuroPKI 2006.
3. Housley, R., Polk, W., Ford, W., Solo, D.: Internet X.509 Public Key Infrastructure - Certificate and Certificate Revocation List (CRL) Profile. RFC3280 (2002)
4. Rekhter, Y. Li, T.: A Border Gateway Protocol (BGP). RFC4271 (2006)
5. Murphy, S.: BGP Security Vulnerability Analysis. RFC4272 (2006)
6. Kent, S., Lynn, C., Seo, K.: Secure Border Gateway Protocol (S-BGP).: IEEE Journal on Selected Areas in Communications, Vol. 18, No. 4, 582–592 (2000)
7. Goodell, G., Aiello, W., Griffin, T., Ioannidis, J., McDaniel, P., Rubin, A.: Working Around BGP: An Incremental Approach to Improving Security and Accuracy for Interdomain Routing. In: Network and Distributed System Security Symposium, 73-85 (2003)
8. Hu, Y.-C., Perrig, A., Johnson, D.: Efficient Security Mechanisms for Routing Protocols. In: Network and Distributed System Security Symposium, 57–73 (2003)
9. Wan, T., Kranakis, E., van Oorschot, P.C.: Pretty Secure BGP (psBGP). In: Network and Distributed System Security Symposium (2005).
10. Kent, S.: Securing BGP: S-BGP. In: The Internet Protocol Journal, Vol. 6-3, 2–14 (2003)

11. White, R.: Securing BGP: soBGP. In: The Internet Protocol Journal, Vol. 6-3, 15–22 (2003)
12. Housley, R., Polk, T.: Planning for PKI, Wiley Computer Publishers (2001)
13. Opplinger, R.: Secure Messaging with PGP and S/MIME, Artech House Publishers (2000)
14. http://ietfreport.isoc.org/idref/draft-ietf-sidr-res-certs
15. http://www.mysql.com
16. http://mirin.apnic.net/bbn-svn/BBN_RPKI_software/trunk
17. http://mirin.apnic.net/resourcecerts/wiki
18. draft-ietf-sidr-arch-01.txt, available from http://ietfreport.isoc.org
19. http://www.openssl.org
20. http://www.cs.auckland.ac.nz/ pgut001/cryptlib
21. http://ietfreport.isoc.org/idref/draft-ietf-sidr-roa-format
22. http://subvert-rpki.hactrn.net