NIST Technical Note 2060

BGP Secure Routing Extension (BGP-SRx):

Reference Implementation and Test Tools for Emerging BGP Security Standards

Oliver Borchert
Kyehwan Lee
Kotikalapudi Sriram
Doug Montgomery
Patrick Gleichmann
Mehmet Adalier

This publication is available free of charge from:
https://doi.org/10.6028/NIST.TN.2060

National Institute of Standards and Technology
U.S. Department of Commerce

NIST Technical Note 2060

BGP Secure Routing Extension (BGP-SRx):

Reference Implementation and Test Tools for Emerging BGP Security Standards

Oliver Borchert
Kyehwan Lee
Kotikalapudi Sriram
Doug Montgomery
Advanced Networks Technology Division
Information Technology Laboratory

Patrick Gleichmann
Applications Systems Division
Office of Information Systems Management

Mehmet Adalier
Antara Teknik LLC

This publication is available free of charge from:
https://doi.org/10.6028/NIST.TN.2060

September 2021

U.S. Department of Commerce
Gina M. Raimondo, Secretary

National Institute of Standards and Technology
James K. Olthoff, Performing the Non-Exclusive Functions and Duties of the Undersecretary of Commerce for Standards and Technology & Director, National Institute of Standards and Technology

National Institute of Standards and Technology Technical Note 2060
Natl. Inst. Stand. Technol. Tech. Note 2060, 161 pages (September 2021)
CODEN: NTNOEF

This publication is available free of charge from:
https://doi.org/10.6028/NIST.TN.2060

Foreword

In the early 2000s, the IETF formed the Secure Inter-Domain Routing (SIDR) working group which was tasked with developing a security model for the Border Gateway Protocol (BGP) with the intent to eliminate or reduce the rate of successful BGP hijacks and other attacks against the core routing infrastructure. The result was the development of a two-stage security approach, one based on the prefix (IP address range) origination of an autonomous system's (AS) announcement and the other one dealing with the validation of the path such an announcement traversed on. The first stage is called the Resource Public Key Infrastructure (RPKI) and has been in its deployment stage since early 2013 and the second one is called BGPsec and includes a modification to the BGP specification RFC 4721. BGPsec became an RFC standard in late 2017. During that time, NIST actively participated in the development of the necessary RFCs and developed in parallel a reference implementation that addresses both tiers of the developed security model.

Abstract

In this paper, we first describe the problem space. Following that, we describe the design and implementation of the NIST reference implementation for RPKI-based route origin validation (BGP-OV) and BGPsec path validation (BGP-PV) within a BGP router. The system we developed is called BGP Secure Routing Extension (BGP-SRx).

We describe the system design, explain the design choices, describe communications between all components, and present the performance measurements obtained during the implementation stages. This paper is organized so that it first explains the high-level system design with a brief explanation of all components and how they interact. We will explain why we chose this design and provide a discussion of its benefits as well as shortcomings. Furthermore, we show which open-source components we chose and how we extended them for this project.

The BGP-SRx implementation is a reference implementation for BGP-OV with all its router side components as specified in RFC 6811, RFC 6810, and RFC 8210 as well as for BGPsec path validation as specified in RFC 8205 and RFC 8608. The implementation allowed early identification of issues while the specifications were still under development, hence provided important feedback to the development of the different IETF RFCs.

Key words

Border Gateway Protocol (BGP) security, BGP origin validation (BGP-OV), BGP path validation (BGP-PV), BGPsec, Internet infrastructure security, Resource Public Key Infrastructure (RPKI), Routing security, and robustness.

Table of Contents

List of Tables

List of Flow Charts

List of Figures

List of Listings

Glossary

AFRINIC	The Regional Internet Registry for Africa. AFRINIC is responsible for the allocation and management of Internet numbers according to their community-backed policy.
APNIC	The Regional Internet Registry for the Asia Pacific region that allocates and registers Internet resources in its region.
ARIN	The American Registry for Internet Numbers for Canada, the United States of America, and many Caribbean and North Atlantic Islands.
AS	An Autonomous System specifies a network, mostly an organization that can own or announce network addresses to the Internet.
BGP	The Border Gateway Protocol is a protocol designed by the Internet Engineering Task Force (IETF) to exchange routing information between autonomous systems.
BGPsec	Security extension to the BGP protocol. It allows to digitally sign path information that can be independently verified and therefore eliminate the possibility to alter path information without notice.
BGP-PV	BGP Path Validation specifies a mechanism described in RFC 8205 that allows verifying a path an UPDATE traversed by verifying the validity of the signatures over the relevant data.
BGP-OV	BGP Origin Validation specifies a mechanism described in RFC 6811 where a BGP router can verify if a particular prefix was allowed to be announced by the route's originator.
BGP-SRx	BGP Secure Routing Extension consisting of multiple software modules that implement Route Origin Validation as well as BGPsec Path Validation.
BIO	BGPsec Input-Output (BIO) enables the generation and storage of pre-computed reproducible BGPsec traffic for testing purposes.
IETF	The Internet Engineering Task Force is the premier Internet standards body that develops open Internet standards.
LACNIC	The Internet Address Registry for Latin America and the Caribbean, responsible for assigning and managing Internet number resources for their region.
RFC	A Request For Comments is a formal standards-track document developed in working groups within the Internet Engineering Task Force (IETF).
RIPE NCC	Regional Internet Registry for Europe, the Middle East, and parts of Central Asia that allocates and registers blocks of Internet number resources to Internet service providers (ISPs) and other organizations.

ROA	A Route Origin Attestation is a cryptographically verifiable attestation that a given Internet prefix can be announced by an AS listed within the attestation.
RPKI	The Resource Public Key Infrastructure is a framework aimed to secure the Internet's routing infrastructure, in particular the routing information such as the IP address prefix and Originator mapping embedded in the BGP protocol. It provides certificates that are used to verify if the originating AS is permitted to publish the embedded IP address prefix(es).
RVC	RPKI Validation Cache provides Validated ROA Payload (VRP) and public router keys.
VRP	Validated ROA Payload contains {prefix, max length, origin AS} information from an X.509 validated ROA.

K	Kilo as Measuring Unit 10^3 = 1 000
KiB	Kibi Byte, Measuring Unit 2^{10} Bytes = 1 024 Bytes
MiB	Mebi Byte, Measuring Unit 2^{20} Bytes = 1 048 576 Bytes
GiB	Gibi Byte, Measuring Unit 2^{30} Bytes = 1 073 741 824 Bytes

1. Introduction

The Border Gateway Protocol (BGP), specified in RFC 4271 [7], is often called the glue that holds the Internet together. It is used by BGP speaking systems called Autonomous Systems (AS) to exchange network reachability information. This information contains the Internet Protocol (IP) address range called IP prefix and the path the information traversed from the prefix originator to the receiver. The problem with BGP is that it was designed without any form of security in mind. Each AS can announce any prefix regardless of if it has reachability to the prefix or not. Bad BGP announcements that steal traffic away from the intended destination are called prefix hijacks. Often these hijacks are due to operator errors and misconfigurations but increasingly these announcements are malicious with the intent to either disrupt connectivity, redirect traffic to allow eavesdropping, etc. Because of this and other BGP vulnerabilities, the Internet Engineering Task Force (IETF) chartered [8] in 2006 the Secure Inter-Domain Routing Working Group (SIDR WG) with the single purpose to reduce vulnerabilities in the inter-domain routing system. Two main security principles were identified to be addressed within the SIDR WG:

- Is an Autonomous System authorized to originate an IP prefix?
- Is the AS path represented in the route the same as the path through which the prefix traveled?

The first one seeks to address prefix hijacking and subprefix hijacking, where an organization or hostile actor erroneously announces that a certain prefix or subprefix is reachable through their network. The result of such a hijack can range from increased latency to a total loss of connectivity. The second principle concerns the path the update traversed. By removing AS numbers from the path, the total AS path length can be reduced, and therefore the possibility that the modified path may be chosen over another legitimate path is increased. Using this technique an attacker can redirect traffic to go through the attacker's network which gives the attacker the possibility to inspect the traffic and extract valuable information.

During the lifetime of the working group, a set of standard documents were developed which do address these two vulnerabilities – prefix hijacking and path manipulation. During this time, NIST started developing reference implementations of these standards to identify possible issues and provide the learnings back to the standards body.

In this paper, we describe the design and development of the NIST reference implementation for Resource Public Key Infrastructure (RPKI) origin validation that addresses the prefix origination problem space and BGPsec path validation. BGPsec is specified in RFC 8205 [5] and addresses the second part of the problem space. The system NIST developed is called BGP Secure Routing Extension (BGP-SRx).

Furthermore, we describe the system design, explain the design choices, communications between all components, and present the performance measurements done during the implementation stages. This paper is organized such that it first explains the high-level system design with a brief explanation of all components and how they interact. We will explain why we chose this design with its benefits as well as shortcomings.

Additionally, we show which open-source components we chose and how we extended them for this project. The BGP-SRx implementation is a reference implementation for BGP origin validation with all its router side components as specified in RFC 6811 [1], RFC 6810 [2], and RFC 8210 [4] as well as for BGPsec path validation as specified in RFC 8205 [5], RFC 8608 [6]. The implementation allowed early identification of issues while the specifications were still under development, hence provided important feedback to the development of the different IETF RFCs.

2. System Design and Requirements

After some study of the different BGP router platforms, we decided to base the reference implementation on Quagga 0.99.22 [11], an open-source router platform. In parallel, another group of researchers participating in the IETF standards development for BGPsec path validation (BGP-PV) chose to extend the BIRD Internet routing daemon [12], to independently incorporate BGP origin validation (BGP-OV) and BGPsec path validation[13].

The overall system is split into four main BGP-SRx components:

These BGP-SRx components are:

(1) The Quagga based BGP/BGPsec router (QuaggaSRx or QSRx)

(2) The SRx-Server validation engine (SRx-Server or SRxSnP)

(3) The SRx Crypto API (SCA) for BGPsec path validation and signing

(4) A BGP/BGPsec capable traffic generator that allows us to generate large loads of fully signed multi-hop BGPsec traffic.

Additionally, we developed an RPKI Validation Cache (RVC) Test Harness that simulates the functions of a real-life RVC. This implementation allows to script test objects that contain ROA and BGPsec Key information that are then send to the client using the RPKI to router protocol [4].

In this paper, we concentrate on the four BGP-SRx components, namely, the BGP/BGPsec router, the SRx-Server, SRx Crypto API, and the BGPsec Traffic Generator (e.g. BGPsec IO or BIO). The additional test harness is briefly discussed in the Appendix.

The most important design decision for BGP-SRx was to outsource the Resource Public Key Infrastructure (RPKI) and BGPsec related processing from the router. This provides us with the following set of advantages:

- Router modifications are reduced to a minimum.
- Less effort when replacing the underlying router platform.
- Sharing of route validation results within multiple router instances.
- The workload of validation and maintaining state is removed from the router.

To accomplish this task, we first identified the minimum functional requirements for the router.

- The router needs to provide a mechanism (configuration, policies) to make use of the BGP-OV results as well as BGP-PV results.
- For BGPsec, the router needs to be able to parse the BGPsec path attribute as specified in RFC 8205 to detect syntactical errors.

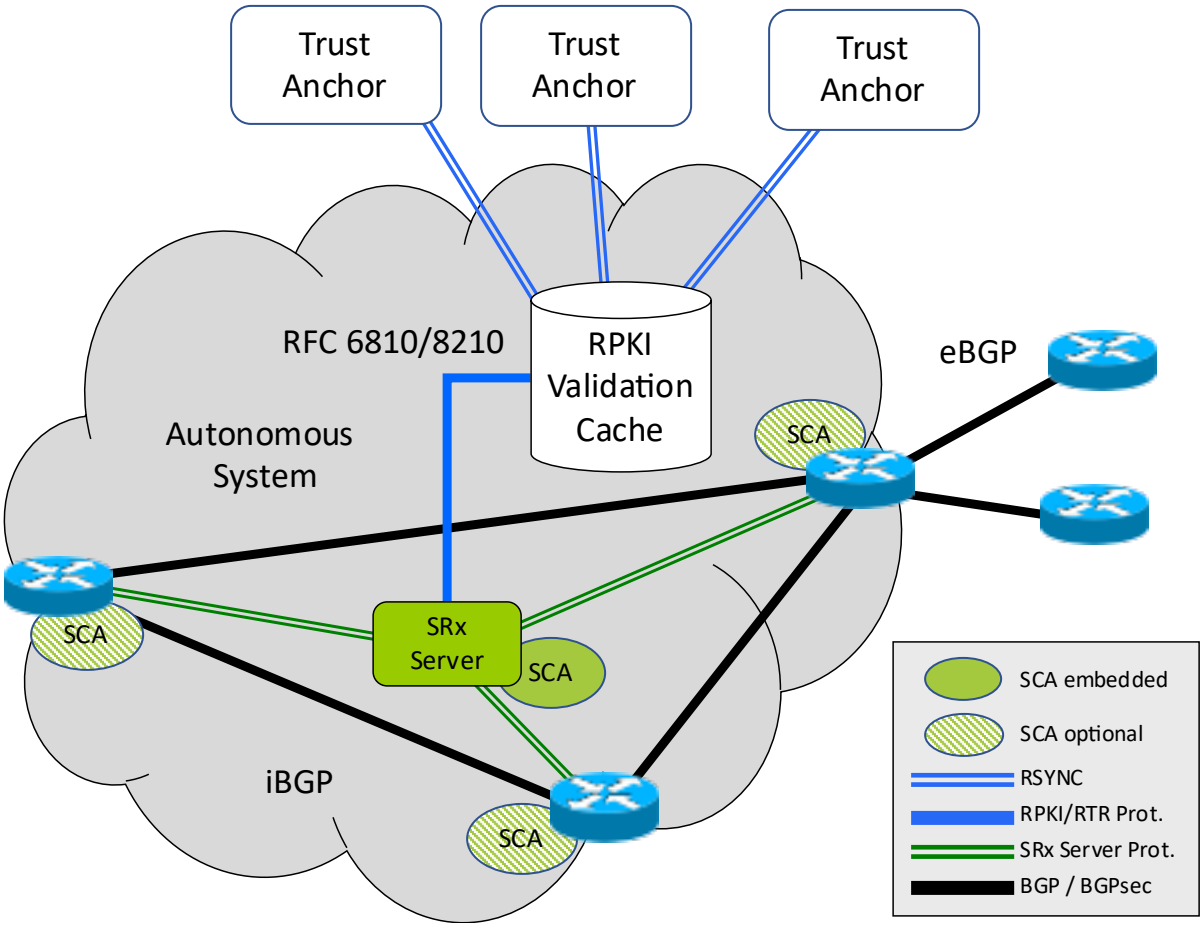


Figure 1 – BGP-SRx System Design

As shown in Figure 1, the RPKI Validation Cache (RVC) retrieves the certificates as well as BGPsec router keys from the Trust Anchor (TA) using RSYNC [14] or the RPKI Repository Delta Protocol (RRDP) as specified in RFC 8182 [15]. The most common TAs consist of the five regional registries, ARIN [16], RIPE [17], APNIC [18], LACNIC [19], and AFRINIC [20]. Once the RVC receives all RPKI certificates, it validates them and extracts the "Validated ROA Payload" (VRP). This payload will be sent to all registered router clients using the router to cache protocol (RPKI/RTR Prot.) as specified in RFC 6810 and RFC 8210.

The SRx-Server communicates with the RPKI validation cache via the router to cache protocol as specified in RFC 6810 (version 0) and RFC 8210 (version 1). Version 1 of the protocol is required to receive the BGPsec public router keys.

The BGP/BGPsec router communicates with the SRx-Server using the proprietary SRx-Server-Protocol [21]. To reduce the burden on the router, the SRx-Server provides an SRx-Proxy API which functions as a wrapper for bi-directional communication with the router. The SRx-Server provides both, BGPsec path validation (BGP-PV) and BGP origin validation (BGP-OV). The SRx-Server monitors changes within the RPKI and determines the effect on the validation result of prefix origin pairs and BGPsec secure path segments. Once a change in validation result is identified, the SRx-Server sends a notification to all routers registered with each affected update.

Furthermore, the system provides the SRxCryptoAPI (SCA) that introduces a high degree of flexibility regarding BGPsec cryptography. Since version 5.0, The BGP-SRx framework moved all BGPsec path processing into the SCA. The SCA functions as a wrapper for BGPsec algorithm implementations as well as a storage for public and private keys[1]. SCA is embedded in both the QuaggaSRx as well as the SRx-Server. SCA allows the handling of private router keys for QuaggaSRx locally for path signing and the processing-intensive validation for the SRx-Server. This removes the router's burden of handling thousands of expected public router keys. The number of public keys can range from one per Autonomous System (AS), of which there are currently approximately 70 000, to one key per router with tens of routers per AS which would easily exceed a million keys.

As shown in Figure 2, the SRx Server can run on the router platform, centralized within the organization, or on the same platform as the RPKI Validating Cache.

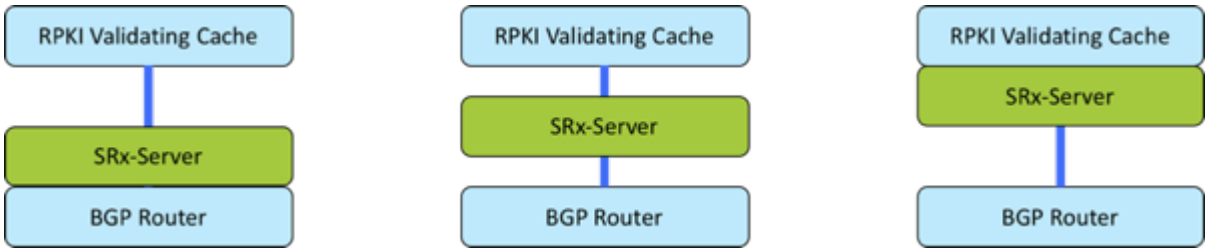


Figure 2 – Deployment options of SRx Server

[1] The SCA provides a simple non-side-channel-attack-safe key storage which can be replaced with any linked custom algorithm plugin.

2.1. BGP-SRx Test Framework

The BGP-SRx framework as shown in Figure 3 provides a set of test utilities such as a simple test harness that allows feeding RPKI objects to the router and a highly complex BGP / BGPsec traffic generator called 'BGPsec-IO (BIO)' that allows generating BGP-4 and BGPsec UPDATEs. These UPDATEs will be sent to a BGPsec speaker or passed to an SCA crypto module for BGPsec cryptographic performance testing. The test framework includes RPKI origin validation signaling as specified in RFC 8097 [3] as well as a reference implementation of RFC 8654 [10] with extended message support for BGP. BIO allows to pre-compute reproducible BGPsec traffic and stores it for later testing. This is especially useful for live BGPsec UPDATE traffic which is sent to routers to test their convergence times. In addition, these stored UPDATEs can also be used to feed into the SCA to allow reproducible tests for module implementations configured within the SRxCryptoAPI cryptography engine.

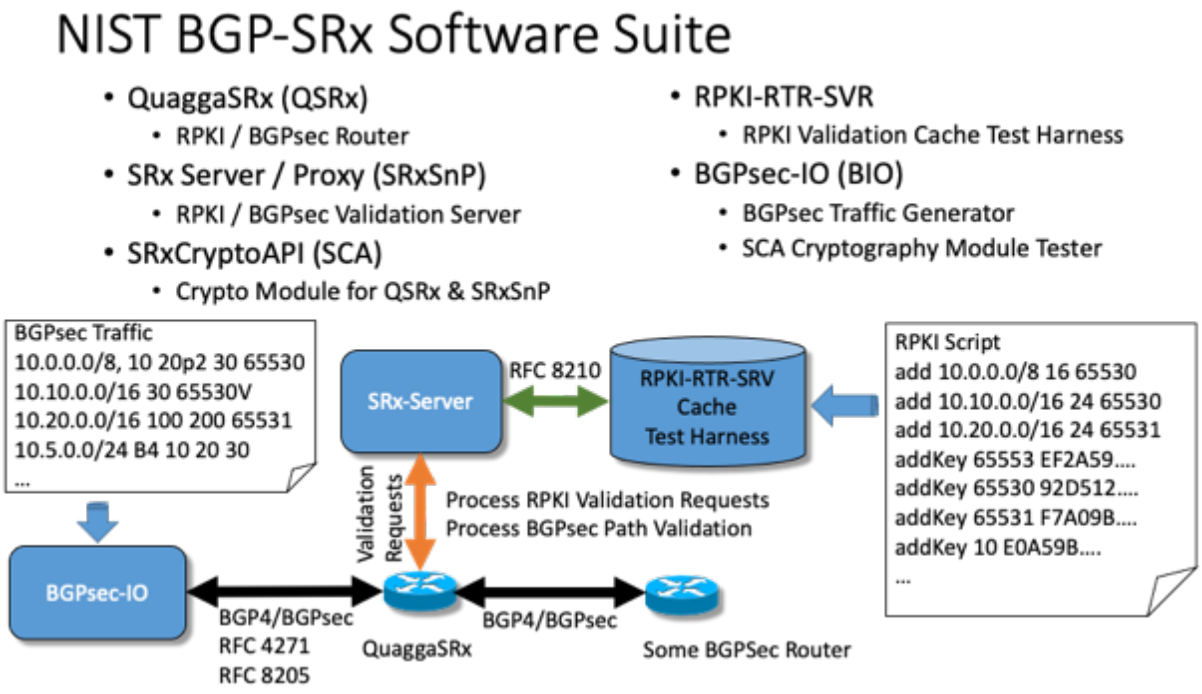


Figure 3 – NIST BGP-SRx Software Suite Test Setup

2.1.1. BGP-SRx Specific Validation States

We found it important to allow to identify situations where UPDATEs are not yet validated, or not enough data was available to validate – the latter case could be during synchronization with the RVC or if no RVC is available[2]. RFC 6811 which is used for BGP-OV does provide three validation states: Valid (V), Invalid (I), and Not-Found (N) and urges implementers to assign Not-Found to updates that were not validated:

"If validation is not performed on a Route, the implementation SHOULD initialize the validation state of such a route to "NotFound"".

RFC 8205 specifies two validation states: Valid (V) and Not Valid which we will refer to as Invalid (I) going forward. To allow identifying the situation where validation is not performed or not complete, we deemed it important to add Undefined using the symbol (?) as an additional validation state. This additional state implies that no validation was performed [22][23]. One possible reason could be that no connection to the validation engine could be established and therefore no validation could be performed. Another reason could be that not enough data was available to perform a validation, which could happen if the synchronization between SRx-Server and RVC did not complete yet or the communication between the QuaggaSRx router and the SRx-Server could not be established. Therefore, we believe it is important to allow acknowledging when no validation is performed with a state of "undefined/unverified" in both BGP-PV and BGP-OV validations.

2.2. Outsourcing Route Origin Validation and BGPsec Path Validation

As mentioned earlier, the validation for BGP-OV and BGP-PV is implemented by the BGP-SRx's SRx-Server. Detaching this validation from the router implementation not only relieves the router from performing additional work for the maintenance and validation of routes but also allows being more flexible in the selection of the underlying routing platform. This allows us to select additional routing platforms and extend them to be BGP-OV and BGP-PV capable.

Performing RVC lookups and handling failovers and how they affect the validation will become a significant issue (especially regarding RPKI maintenance) as the amount of data within the RPKI increases. This is even more of an issue regarding BGP-PV, as here RPKI not only provides ROA VRPs but also public router keys. Changes in the inventory of public keys through expiration, addition, key rollovers, etc. do have an immediate effect on routes. Signatures that previously passed validation might become invalid or vice versa. For each update, there is not only one data tuple [prefix, origin] to monitor. In BGP-PV, each path segment has to be monitored. With hundreds of thousands of keys, this can become a significant load increase on routers that could be dealt with in separate modules. Providing an external API that can be configured to use cryptographic plugins, allows to easily exchange BGP-PV crypto algorithms without the need of recompiling all the code. Further detailed information will be presented in Section 4.5.2 of this document.

The other important advantage of outsourcing BGP-OV and BGP-PV is monitoring modifications within the RPKI. Monitoring changes in BGP-OV are less costly than monitoring changes in BGP-PV. The current Routing Information Base (RIB), also called routing table, consists of approximately 1.1 million prefix origin pairs (IPv4 and IPv6 combined)[37][38]. Finding the correct prefix when stored in a trie is quick and with a smart

[2] In case no RPKI data is available due to RVC unreachability, no proper validation can be performed.

data structure such as the structure used in SRx-Server, validation changes of UPDATEs can be easily detected by just maintaining the data structure. BGP-PV though is different. With the current average AS path length of about four hops and each hop having its own Secure Path segment with each segment having its own signature, the data volume and processing does increase dramatically. This means that there can be 1.1 million (prefixes or routes) x 4 segments which amounts to 4.4 million signed BGPsec path segments (per peer). To map the keys to the appropriate path segment requires a key-segment manager. Finally, not all keys can be kept in memory as there are just too many of them. For all these reasons it is more than feasible to outsource the BGP-PV and its related management. Furthermore, a router performing BGP-PV with full deployment would have to perform up to 4.4 million signature verifications per peer for a full route exchange. This alone calls for the outsourcing of BGP-PV.

3. Router Implementation

For the router implementation, we decided to extend the Quagga[11] open source BGP routing platform. One major design decision was to keep the required modification of the codebase to a bare minimum to allow easier integration into additional platforms. Consequently, we decided that BGP-OV should not be performed on the router itself but on an external platform which we called SRx-Server. This frees the routers from the tasks of synchronizing with RVC's, and monitoring of prefix origin pairs and BGPsec secure path segments that would be affected by modifications within the RPKI. Furthermore, outsourcing the validation for both BGP-OV as well as BGP-PV allows a "lazy evaluation" by delaying evaluation when needed and therefore lowering the success of DOS attacks that are aimed at the validation engine.

This leaves the QuaggaSRx router with only a small set of new responsibilities:

- Add policies that include BGP-OV and BGP-PV results in the route selection process.
- Add BGPsec path attribute checking for syntactical correctness and error processing according to RFC 8205.
- Add BGPsec path generation according to RFC 8205 including the signing of the path.
- Use the SRx-Proxy interface for all remaining RPKI / BGPsec related functions.
- Implement a "lazy evaluation" mechanism.

As shown in Figure 4, upon receiving an UPDATE message, the router extracts the important information and sends it to the SRx-Server for validation. The SRx-Server then returns the BGP-OV and BGP-PV results to the router. The router then calculates an overall validation result and performs route selection. Once a route is selected and prepared to be forwarded to the peers, the router will sign the update in case BGPsec is enabled.

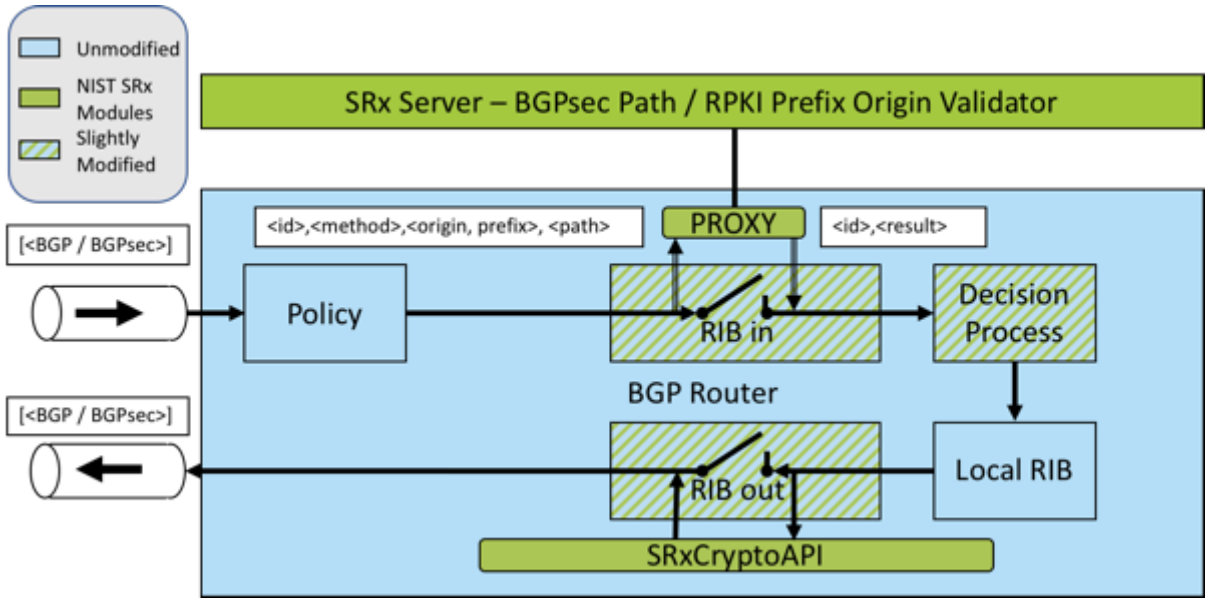


Figure 4 – Process Flow in QuaggaSRx

Please keep in mind that this is a simplified overview of a more complex algorithm. Going forward, when talking about an update we will be referring to the unique triplet [prefix, origin, path]. The path contains either the AS_PATH or the BGPsec_PATH attribute. This information is used to create a system-wide unique update ID which is used by the router and the SRx-Server to identify the update in future communication. To keep the memory usage to a minimum we decided to use an unsigned four-byte integer as ID. Furthermore, we also decided to perform lazy evaluation, which means the router can assign a pre-defined validation value to each update. Then the router initiates the validation request and proceeds with route selection. The SRx-Server in the meantime performs the update validation requests in parallel. The pre-defined validation value can either be configured or if available be provided within the received UPDATE as extended community attribute as specified in RFC 8097. By allowing the router to proceed with route selection while the SRx-Server performs the validation, the initial blocking of prefix processing due to validation calls will be reduced to a minimum.

Once the SRx-Server completes the validation and the validation result differs from the router-provided validation value, the SRx-Server sends a notification to the router containing the new corrected validation result (along with the update ID). Only then would the router redo the route selection process for the prefix in consideration.

Even though we envisioned not performing any BGPsec processing in the router other than checking for syntactical correctness as requested by RFC 8205, we implemented full BGPsec validation and signing processing within the router first. This was a simple decision because all BGPsec related validation processing is out-sourced into the SRxCryptoAPI (SCA). The only required modification of QuaggaSRx was adding the processing of the BGPsec path attribute itself, not the cryptographic processing. This allowed us to perform early BGPsec validation tests prior to extending the SRx-Server to perform BGPsec validation. The SCA allowed adding BGP-PV without adding the validation complexity and key management to the router itself, as the verification of BGPsec UPDATEs, as well as signing request and key management functionalities, are provided by the SCA. The only missing portion was the

dynamic retrieval of BGPsec public keys. Once early BGPsec experimentation with router-based validation was enabled, we extended the SRx-Server to perform BGP-PV as well as key retrieval by extending the router to cache protocol implementation from BGP-OV-only RPKI management as specified in RFC 6811 to add key handling as specified in RFC 8211.

3.1. Router and SRx-Server Synchronization

Once the router connects to the SRx-Server, the SRx Server sends a reset request. This request requires the router to send a validation request for each update in the router's RIB-IN table to the SRx Server. For each validation request, the router uses either the local update ID or the already received SRx Update ID (SUID) in case of a re-request. This assures that the SRx-Server will have complete knowledge of all routes including the routes the router received prior to connecting to the SRx-Server. This process is also performed if the connection between the router and SRx-Server got disrupted to synchronize the state between the router and SRx Server.

3.2. Processing Received BGP / BGPsec UPDATEs

As explained earlier, QuaggaSRx and the SRx-Server communicate using IDs to identify each specific update. As shown in Figure 1, the SRx-Server does not serve a single router. To assure unique IDs within the scope of the SRx-Server, the SRx-Server must generate the system-wide unique SUID and solve any collision conflicts. It is important to mention that if two router instances, both connected to the same SRx-Server, send the exact same update validation request, the SRx-Server must generate the exact same SUID. The same is true in case a router sends the same validation request twice. To allow proper initial communication between the router and the SRx-Server, the router will generate a temporary local ID that uniquely identifies the update within the router. QuaggaSRx uses a simple counter which is sufficient because the local ID has a very short lifespan. The local ID will be replaced once the SRx-Server responded with the system-wide unique SUID. The router sends the initial validation request using the locally generated ID. In addition to the local update ID, the router adds the update information consisting of the triplet [prefix, origin, path] and the assumed validation result. This can be either a pre-configured result or a result learned through the received UPDATE message in form of the extended community string as described in RFC 8097.

Once the validation request is initiated, the router will add the received update into the internal process queue. The SRx-Server will generate a receipt and return it immediately upon receiving the validation request. Contained in the receipt is the local ID[3] and the SRx-Server generated SUID together with the validation result. Once the router receives the validation receipt it will assign the validation results to the stored update in case they differ from the previously stored value.

More information on the generation of the system-wide unique SUID and the validation receipt is described in detail in Section 4.3.

In case the router does not have any connection with the SRx-Server, it generates the local ID and assigns the validation results (that it would have otherwise added to the validation request) to the update and proceeds with normal processing. As soon as the router connects to an SRx-

[3] The SRx-Server interprets the local update ID as request token to map the receipt to the request.

Server instance all updates will be sent to the SRx-Server. The SRx-Server then will generate new SUIDs and provide them to the router within the validation receipts.

When performing an operator-initiated update lookup, the type of update ID as shown in Listing 1 can be identified by the occurrence of a "+" right before the ID value. Local IDs are displayed using the "+" as a prefix. Once a router receives the SRx Server's update ID the "+" is not displayed. In case the update is originated from within the AS no update ID is allocated but represented as (--------) and labeled as valid. The field SRxLP specifies the local preference modification values as a result of the validation state. These values are used to modify the current local preference depending on the configuration. This is only used if local preference policies as described in Section 3.5.3 are employed.

```
bgpd# show ip bgp
BGP table version is 0, local router ID is 10.0.6.50
Status codes: s suppressed, d damped, h history, * valid, > best, i - internal,
              r RIB-failure, S Stale, R Removed
Validation:    v - valid, n - notfound, i - invalid, ? - undefined
SRx Status:    I - route ignored, D - SRx evaluation deactivated
SRxVal Format: validation result (origin validation, path validation)
Origin codes: i - IGP, e - EGP, ? - incomplete

   Ident    SRxVal SRxLP Status Network        Next Hop   Metric  LocPrf Weight Path
*> -------- v(v,v)              10.50.0.0/16   0.0.0.0         0          32768 i
*> +000001F ?(?,?)              10.60.0.0/16   10.60.0.1       0              0 60 ?
*  D7A9D0A8 i(i,v)          I   10.60.0.0/24   10.70.0.1       0              0 70 90 ?
*> 44AB9E12 v(v,v)              10.70.0.0/16   10.70.0.1       0              0 70 ?
*> 289682FD v(v,v)              10.80.0.0/16   10.80.0.1       0              0 80 ?
*  328B5F6E i(i,v)          I   10.80.0.0/24   10.70.0.1       0              0 70 90 ?
*> 9216E8E2 v(v,v)              10.90.0.0/18   10.70.0.1       0              0 70 90 ?
*> +0000020 ?(?,?)              172.16.2.0/24  10.60.0.1       0              0 60 2 ?
*> D4128FE7 v(v,v)              172.16.5.0/24  10.70.0.1       0              0 70 90 5 ?
*  A3267A30 i(i,i)          I   172.16.7.0/24  10.80.0.1       0              0 80 2 ?
*> 02B273E0 v(v,v)                             10.70.0.1       0              0 70 90 7 ?

Total number of prefixes 9
bgpd#
```

Listing 1 – QSRx Terminal – BGP RIB-IN Table

3.3. Processing Validation Notifications

QuaggaSRx does use the SRxProxy API which is the remote interface to the SRx-Server that is explained in detail in Section 4.2.2 of this document. QuaggaSRx does provide call-back functions that will be called by the SRxProxy API once validation results are received.

Once a validation result is received, QuaggaSRx first examines if the notification is a "receipt" to a validation request or a notification for validation state changes. A receipt differs from the regular notification by containing not only the SUID but also the router's internal local ID. This mechanism allows the router and the SRx-Server to synchronize their ID space. Once a receipt is received, the router queries for the update with the given local ID and replaces the local ID with the given SUID. In case the notification was not a receipt, no local ID is provided and instead, the router queries the internal tables for the update with the given SUID.

Once the update is retrieved (both during receipt and regular notification), QuaggaSRx verifies each validation result that was provided within the notification. If the final validation result (BGP-OV, BGP-PV) differs from the stored result, QuaggaSRx will temporarily store the overall validation result, replace the validation result with the newly provided result, and recalculate the overall validation result. Only if the overall validation result has changed will

QuaggaSRx restart the decision process. This prevents unnecessary processing within the router.

3.4. Calculating the Internal Validation Result

Even though RPKI origin validation and BGPsec path validation are treated separately, the QuaggaSRx implementation eventually generates one final state for each BGP update. As a clarification to start with, RFC 6811 defines the validation state Invalid where RFC 8205 declares the validation state Not Valid. Both mean the same, e.g. the validation result ended with an Invalid outcome. In the BGP-SRx Framework, we decided to use the notion of Invalid rather than Not Valid to allow simplification in the internal handling of the validation states.

QuaggaSRx assigns to each update two separate validation states, one for BGP-OV (S1) and one for BGP-PV (S2). The validation state for S1 can contain any of the values listed in Table 1 whereas all but "Not Found" can be assigned to S2.

Validation State		Description of validation state
Symbol	Name	
V	Valid	The received update is valid
I	Invalid	The received update is invalid
N	Not Found	No ROA information for the given BGP-4 update available
?	Undefined	Not verified yet or not enough data to compute a validation state

Table 1 – BGP-SRx validation state

Table 2 below illustrates the validation state results calculated within QuaggaSRx in case both validation algorithms BGP-OV and BGP-PV are enabled. As shown, the combination of both will only result in "Valid", "Invalid" (Not Valid per RFC 8205), or "Undefined / Unverified".

QuaggaSRx Overall Result ↘		BGPsec Path Validation – S2		
		Valid	Invalid	Undefined
BGP – Origin Validation – S1	Valid	V	I	?
	Not Found	I	I	I
	Invalid	I	I	I
	Undefined	?	I	?

Table 2 – QuaggaSRx validation state calculation table

In case only BGP-OV is enabled, all four states "valid", "invalid", "not-found", and "undefined" can be assigned. The operation mode depends strictly on the configured policy modes.

3.5. Validation Policies

QuaggaSRx operates in two distinct policy modes:

- origin_validation
- bgpsec

The first mode "*origin_validation*" only performs BGP-OV for route validation. In this mode, BGP-PV is disabled, and the overall validation can result in "*valid*", "*invalid*", "*not-found*",

and "*undefined*". In case the second mode "*bgpsec*" is enabled, the router will perform both, BGP-OV and BGP-PV. The overall validation state is computed using the validation state displayed in Table 2 and can only result in "*valid*", "*invalid*", and "*undefined*".

Policies within QuaggaSRx are applied to the overall validation result. QuaggaSRx provides three types of policies which can be combined. The below-specified policies are applied in the following order as specified.

- Ignore Policy (Prior to decision Process)
- Prefer Policy (After multi exit discriminator (MED))
- Modification Policy (After Prefer Policy)

3.5.1. Ignore Policy

Ignore policies are used to ignore received routes. The three ignore policies are "*Ignore Invalid*", "*Ignore Not-Found*", and "*Ignore-Undefined*". If at least one of the ignore policies is configured and a route update is evaluated and matches such a policy, the ignore flag is set and the route will not be further considered for route selection until the flag is cleared.

3.5.2. Prefer Policy

QuaggaSRx provides a single preference policy "*Prefer-Valid*" which is used to give preference of a valid route over any other route. The prefer-valid policy is added in the decision process on the second position right after the MED check. This policy has no effect on two competing routes that are both marked as valid.

3.5.3. Modification Policy

The modification policy does alter the local preference of a route. QuaggaSRx allows configuration of the modification for all validation states separately. Furthermore, the modification can be additive or finite. The finite modification overwrites the previous stored local preference with the configured one for the particular validation state. The additive modification allows increasing or decreasing the already specified local preference. For instance, the router is configured to increase routes with the validation state "not-found" by 10. Now the router compares two routes R1 and R2 with the same validation state "not-found" but different local preference LP(R1) = 100 and LP(R2) = 120. The router will recalculate prior to the route comparison the local preference from LP(R1) = 100 to LP(R1) = 110, and LP(R2) = 120 to LP(R2) = 130 and then compare both local preferences. In case the policy is not additive then the router will modify the assigned local preference to the one associated with the policy. Using the example from above, and a policy configured to set the local preference for "not-found" routes to 10, the local preferences would change from LP(R1) = 100 to LP(R1) = 10 and LP(R2) = 120 to LP(R2) = 10. In the latter example both routes would now have the same preference values whereas, in the first example with the additive formula, R2's preference would still be valued higher than the one of R1.

3.6. BGPsec Path Processing

The implementation of BGPsec required modification in packet processing. BGPsec introduces a new BGP attribute and modifies the BGP protocol in various ways. If the BGPsec_PATH attribute is provided, no AS_PATH attribute is allowed to be included in the UPDATE. Also, the route prefix must be encoded in MP_NLRI regardless of whether IPv4 or IPv6 is used and cannot be included as NLRI. BGPsec allows only a single BGPsec_PATH

and only one MP_NLRI prefix. The mechanism known as “Prefix Packing” (see Section 6.4.3) is not used in BGPsec.

Once QuaggaSRx receives a BGPsec UPDATE, it performs a syntax check of the BGPsec_PATH attribute and generates the corresponding AS_PATH attribute. Depending on the configuration, QuaggaSRx either initiates a validation call to SRx-Server or performs a local path validation call using the SRxCryptoAPI. This can be configured within the router configuration file. In the latter mode, the SRxCryptoAPI must be configured using a local key store. More on SCA cryptography and key storage is available in Section 5.

If the router is configured for remote validation using the SRx-Server, the BGPsec path validation will be performed by the SRx-Server as described later in Section 4.5. In this mode, the router performs a validation call in the same way as the BGP-OV validation call. Each validation call can be sent with a single validation request. In fact, the validation request for SRx-Server contains a flag specifying the mode of validation (BGP-OV, BGP-PV, or both).

Once a route prefix is selected and ready to be sent to the peer, QuaggaSRx uses the SCA locally for path signing. For this, the router configures the SCA with the required private key used for signing. For the signage, QuaggaSRx passes the received BGPsec_PATH attribute and forwarding data to the SCA with the keys Subject Key Identifier (SKI) to specify the key to be used. The SCA performs the signature generation. Key management and all cryptography related work are part of the SCA. In this implementation, we did choose a simple file-based key install rather than the mechanism described in section 8 of RFC 8635 [9].

3.7. The Command Line Interface (CLI)

Quagga-SRx provides a wide array of features added for BGP-OV and BGP-PV. All of them can be configured using the BGP configuration script. In addition to all the configuration settings, QSRx also provides CLI commands used for debugging and reconfiguration. A comprehensive list of all commands is available in Appendix A.1.

3.8. Future Considerations

As explained previously, the current implementation allows two modes, BGP-OV only and BGP-OV combined with BGP-PV. This resulted out of an early model in BGPsec, where BGP-OV and BGPsec were combined. Eventually, this model was obsoleted. For a future version, we will consider separating both validations.

The current implementation of SRx-Server does treat BGP-OV and BGP-PV separately though. This allows Quagga-SRx to separately specify policies for BGP-OV and BGP-PV. One could envision only manipulating the local preference regarding BGP-OV and ignoring updates with BGP-PV Invalid. This granular scripting of policies is not possible in the current model.

4. SRx-Server

The SRx-Server is the validation engine of the NIST BGP-SRx software suite. It provides the service of route origin validation (BGP-OV) and BGPsec path validation (BGP-PV). As shown in Figure 5 the SRx-Server is the "middleman" between the router and the RPKI validation cache (RVC). Once the SRx-Server is connected to the client, most likely a BGP router, it will perform all UPDATE validations requested by the client and start monitoring the UPDATE for validation state changes until the client removed the UPDATE from the router. In case changes to the updates regarding their validations are detected, the SRx-Server will immediately schedule for these changes to be communicated to the client or clients that initially send a validation request for each specific update. To facilitate the communication between the router and SRx-Server, NIST's BGP-SRx provides a proxy module called SRx-Proxy, that implements the client side of the SRx-Server-Protocol. The Proxy-API hides all complexities in dealing with the SRx-Server by functioning as a communication wrapper in between the router and the SRx-Server. This SRx-Proxy is implemented as a C library that the router uses to request validations as well as register local functions that are called by the SRx-Proxy module, once notifications from the SRx-Server are received.

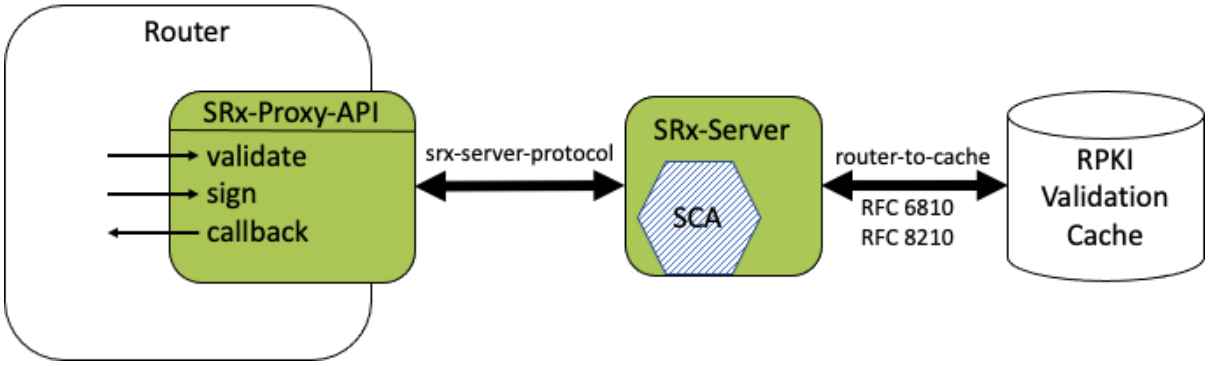


Figure 5 – SRx-Server and SRx-Proxy

For RPKI data retrieval, the SRx-Server does connect to the RVC. Once connected, it does perform continuous data synchronization using the RPKI to Router Protocol as specified in RFC 6810 and configurable using either implementations, RFC 6810 or its updated version RFC 8210. Here the protocol choice depends on the capabilities of RVC. RFC 6810 only provides ROA information whereas RFC 8210 includes BGPsec key information as well.

In both directions, RVC and router, the connections to SRx-Server are stateful.

For BGPsec UPDATE validation, The SRx-Server utilizes the RVC for retrieving BGPsec public keys as long as RFC 8210 is the selected communication protocol. For the BGPsec path validation, SRx-Server uses the SRxCryptoAPI (see Section 5). The SRxCryptoAPI also provides the mechanism for storing the public keys that are required for UPDATE validation.

As shown in Figure 6, the SRx-Server can entertain multiple routers. Routers can benefit from each other by sharing an SRx-Server instance. This assures that each attached router within the organization contains the same validation state.

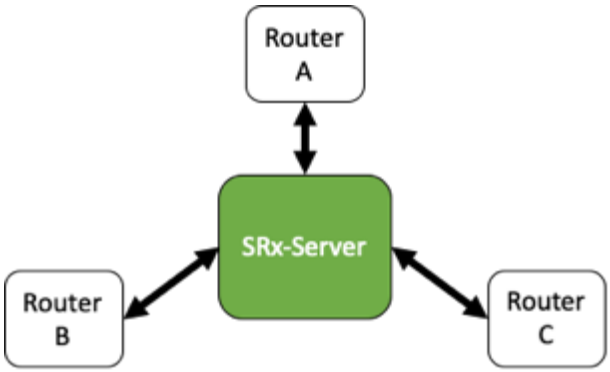


Figure 6 – SRx-Server and Routers

Sharing an SRx-Server instance mitigates the problem of validation conflicts due to inconsistent RPKI states across the domain [31]. Each validation request is accompanied by a validation result the router is using until it receives the final result from SRx-Server[4]. To be able to identify an UPDATE within the scope of the SRx-Server, each update will be assigned a unique SRx UPDATE identifier (SUID) that is unique within the SRx-Servers' domain. This assures that a given *UPDATE U1* received by *router A* will get the same SUID as the *UPDATE U1* when received by *router B*. This can be the case if both routers A & B are connected to routers of the same peering AS. Though this will be highly unlikely for BGPsec UPDATEs due to the nature of non-deterministic signatures which results in different SUIDs. Independent of the UPDATE itself, SRx-Server generates an internal data structure that allows connecting the prefix/origin (PO) pair to the UPDATE validation result (UVR). In case the pair is already known, the SRx-Server will return the validation result stored in the internal data structure. This assures an SRx-Server-wide synchronization of prefix origin validation (BGP-OV) and mitigates the problem that can arise when routers within the same AS synchronize independently at different times or even using different validation caches. This can result in different views on the global RPKI and therefore result in contradicting validation results as explained in detail on slide 10 in [31]. Additionally, using SRx-Server as an UPDATE validation engine allows sharing and synchronizing BGP-OV results across UPDATEs regardless of whether the prefix origin pair was received via BGP-4 or BGPsec.

Changes within the PRKI that do not affect the routing will be dampened. Only these validation changes are communicated to the routers, for which the router requested validation. For instance, SRx-Server received the updates U1 and U2 from router R1 and U1 from router R2. ROA changes are communicated to the SRx-Server that changes the validation result of U2 but not U1. In this case, SRx-Server only contacts R1 about the validation state change in U2. R2 did not request validation for U2 so R2 will not receive any notification.

[4] Only if it differs from the original communicated validation result.

The SRx-Server's data structure is designed in such a way that BGP-OV is a byproduct of adding UPDATE and ROA data to the Prefix Cache.

BGP-PV is slightly different. Validation is not a side effect of data management. Here validation must be triggered separately. This will happen when an UPDATE validation is requested or when key information changes. This happens during key revocations, or if a new key is published. The SRx-Server contains a data structure that allows identifying UPDATEs that contain the SKIs of keys that changed and start explicitly validation whenever needed. BGP-PV can be triggered either by an UPDATE validation request or by an update of the Key Cache after RVC synchronization is performed.

In case the router does not choose to use the SRx-Server for BGP-PV, the router is required to have a more complex data structure than the normal data structure used for the RIB. The data structure must be able to quickly identify all UPDATEs that are affected by a key change. Otherwise, that router would need to check each update in its RIB-IN. In the worst case, that can be a lot, with over 864 000 updates in the global routing table and growing. Also, in case a peer changes its BGPsec key, the complete table for that peer must be revalidated. For changes other than peers, it might be affecting each peer. Again, in the worst case, the number of peers can grow up to 864 000 times the number of peers. Once the UPDATE is identified, BGP-PV has to be restarted because the validation state could change. In addition, the router must perform key management which is memory intensive and not very cost-effective.

Considering these scenarios, deploying SRx-Server seems to be the most logical solution and with only notifying the router of the validation state changes of accepted UPDATEs, the maintenance of ongoing validation becomes completely transparent to the router.

4.1. The SRx-Server Component Design

This section will explain the overall internals of the SRx-Server and their overall relation to each other. Additional sections are used to explain the details of the different mechanisms within the SRx-Server. As outlined in Figure 7 the SRx-Server communicates via TCP. The router communicates with the SRx-Server using the SRx-Server-Protocol, an experimental protocol design developed particularly for this implementation. The details can be found in Appendix A.4. On the RPKI side, the SRx-Server communicates with the RPKI Validation Cache (RVC) via the router to cache protocol as specified in RFC 6810 and if router keys for BGPsec are expected, usage of RFC 8210 must be configured.

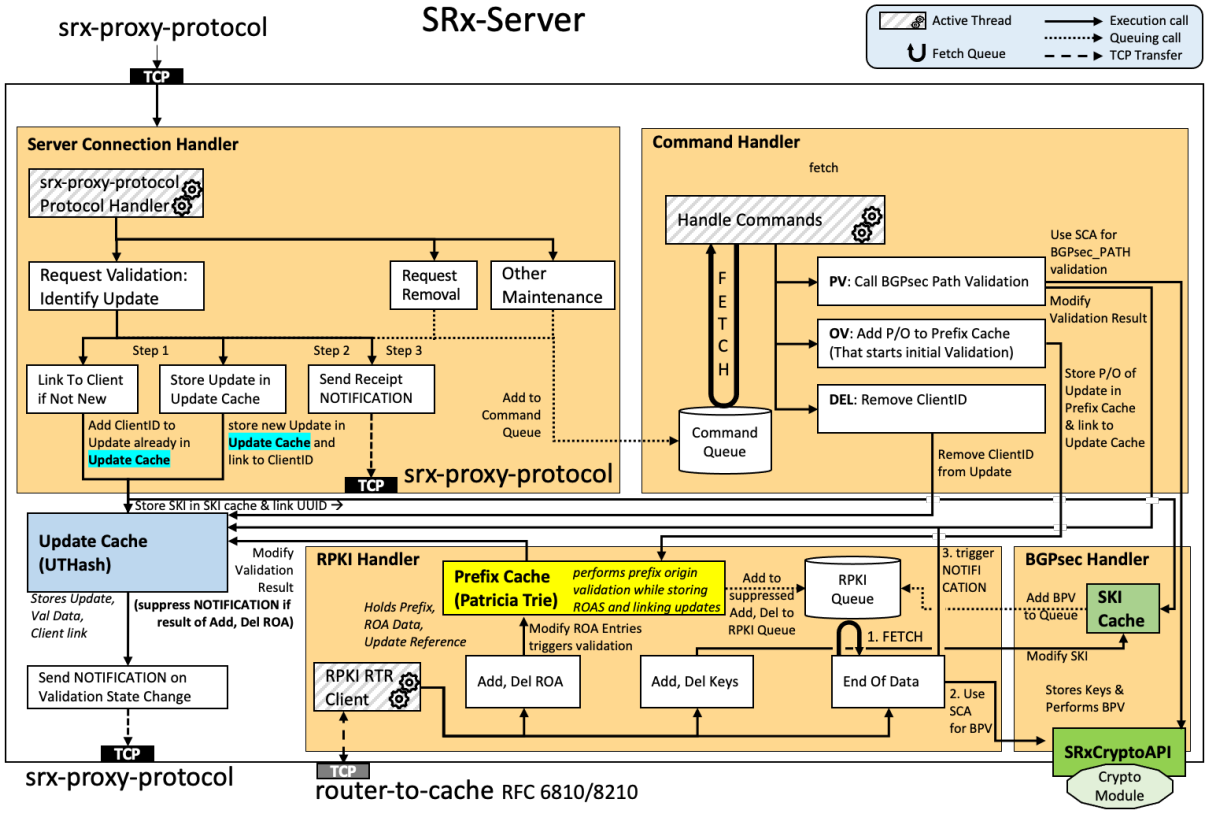


Figure 7 – SRx-Server internals

The internals of the SRx-Server consist of four handlers (Server Connection Handler, Command Handler, RPKI Handler, and BGPsec Handler) as well as three data caches (Update Cache, Prefix Cache, and SKI Cache). The handlers are responsible for keeping the connections stateful, the validations up to date, and the stored RPKI data maintained. The data and validation information are tightly coupled. Once an UPDATE is received for validation, the UPDATE itself is stored in the Update Cache and the prefix is stored in the Prefix Cache. It is to note that from the SRx-Server's perspective, each validation request does contain only a single prefix per UPDATE. Furthermore, the UPDATEs received are not full BGP UPDATEs. They only contain the prefix, AS path list as an array, and all BGPsec relevant data required for validation and signing. In case a BGP-4 UPDATE received by the router did contain multiple prefixes, a separate request is required for each Prefix/UPDATE combination.

Once a validation request is received by the Server Connection Handler, which operates in its own thread, the provided update data will be stored in the Update Cache and a Receipt Notification will be returned to the caller. This is necessary to establish a system-wide unique UPDATE identifier which is used in all follow-up communications. Once the update is stored, the Server Connection Handler does schedule the validation request and adds it into the Command Queue. The request type (BGP-OV and/or BGP-PV) is specified in the Flags field of the validation request (see Appendix A.4).

Each validation request can contain the following type settings as shown in Listing 2. The types are bit coded and can be combined.

```
The result type is bit coded.
            7 6 5 4 3 2 1 0
           +---------------+
           |x|0|0|0|0|0|x|x|
           +---------------+
            |           | |
            |           | |
            |           | +------RESULT_TYPE_ORIGIN =   1
            |           +--------RESULT_TYPE_PATH   =   2
            +--------------------RECEIPT_REQUEST    = 128
```

Listing 2 – Validation request type

The Connection Handler performs a preliminary examination of the PDU received and identifies an UPDATE validation request – ORIGIN or PATH. Furthermore, the Connection Handler will determine if the UPDATE is already known to the system. This is done by generating the SUID as described in detail in Section 4.3 and querying if the UPDATE is already stored in the Update Cache. If not, it initiates the initial storing of the UPDATE in the Update Cache and a receipt notification to the router with the default validation values provided within the request. As long as the SRx-Server does not have its own validation done, it will return the default validation state it received during the initial storing in the Update Cache. Only during this time, it is possible for SRx-Server to ever return the validation state "*Undefined*". In case the SRx-Server received a second validation request for the identical UPDATE but this time with different default validation values, it will add the previously received default values to the Receipt Notification as long as own validated results are not yet available. SRx-Server does this to maintain a system-wide synchronized validation state.

When the SRx-Server stores the update and after it sends the Receipt Notification, it adds the validation request into the *Command Queue*. It shall be noted that during the process of storing the UPDATE data in the Update Cache, the Update Cache does scan the UPDATE for BGPsec path information and stores all SKIs found in the UPDATE in the SKI Cache, and links the Update Cache's update entry to the SKI Cache's SKI entry.

Another thread that runs concurrently is the Command Handler Thread, which fetches commands from the Command Queue for new commands to be read and executed. The Command Handler itself handles all communication requests of the Server Connection Handler but the most important ones are the Verify and Sign commands. If the command is a BGPsec path validation (BGP-PV) request, the Command Handler will call the SRxCryptoAPI for validation. Once the validation is performed and differs from the validation state stored in the Update Cache, the Command Handler will modify the validation result in the Update Cache

which will trigger a validation state change notification to be sent to all clients registered with this update.

In case the Validation Request is a prefix origin validation (BGP-OV), the Update Cache will store the Prefix/Origin combination in the Prefix Cache. The process of adding the update also performs the BGP-OV which will trigger a modification of the update's validation state in the Update Cache in case the value changed. The same does happen during BGP-PV where a validation state change triggers a notification to be sent.

Both remaining handlers, the RPKI Handler that uses the Prefix Cache as well as the BGPsec Handler that uses the SKI Handler, rely on the reference back to the Update Cache. The BGPsec Handler though does not use a dedicated thread. It is only used to instantiate and remove the SKI cache at program start and termination.

This leaves the RPKI Handler, which uses the RPKI_RTR_Client as an actively running process. This process deals with the session handling with the RVC. It acts upon data received by the remote RVC. The difference between validations triggered by the RPKI Handler and validations triggered by the Command Handler is that RPKI Handler-generated validation state changes are not communicated to the registered clients until the last RPKI modification is received and the last validation is performed. Until then, all triggered validation state notifications are queued in the RPKI Queue. The main events are adding and removing of ROAs and BGPsec public keys. Adding and removing ROAs are changes that are made in the Prefix Cache and as mentioned earlier, modifications of the Prefix Cache also immediately modify the validation state of all affected updates. These changes are signaled immediately to the Update Cache, though during this time notification are suppressed and are stored in the RPKI_Queue instead. Key modifications, adding or removing of keys, are performed as they are received. The key management is communicated instantly to the SCA and SKI Cache. Then the SKI Cache adds to the RPKI Queue the list of attached Updated for BGP-PV for each SKI.

Once the Synchronization with the RVC is done, indicated by the EndOfData PDU [4] received, the RPKI_RTR_Client starts fetching all elements queued in the RPKI_Queue. These can be validation state changes to be sent to the registered client or validation requests for the SCA. The latter will modify the Update Cache in case of a state change which results in sending out a validation state change notification as well.

It will be possible that a single update was scheduled multiple times for validation during the processing. For BGP-OV these do increase briefly the communication between the router and SRx-Server. It is important to note though is that for BGP-OV all notifications will contain the last known validation state, because during the generation of the validation state notification, the state will be read from the Update Cache at that specific time. This means that if a router receives a notification 10 times, each time with the same BGP-OV validation state and the SRx-Proxy will drop all but the first notifications because the validation state did not change in between. It is different for BGP-PV though. If "*n*" SKIs of a single UPDATE change, the validation on the SRx-Server side will be scheduled "*n*" times, although only a single notification will be sent because all "*n-1*" validations will result in the same validation state, hence the validation state does not change, and no notification gets prepared. The reason for that is that BGP-PV only starts after all key modifications are processed.

4.2. Communication with SRx-Server

The communication between the SRx-Server and the router is done using the SRx-Server-Protocol, a TCP based protocol that allows keeping a stateful connection between the SRx-Server and the client/router. A detailed description of the SRx-Server-Protocol can be found in Appendix A.4. The client can choose to implement the TCP based communication or decide to use the C-based SRx-Proxy implementation, a library that implements the SRx-API. This implementation is meant for clients such as QuaggaSRx (QSRx) and hides the complexities of the TCP based communication by offering functions for validation and signing requests.

4.2.1. Connecting to SRx-Server

The SRx-Server and the router maintain a stateful connection. For maintenance reasons, the SRx-Server allows allocating a specific ID to a specific connection, as shown in Listing 3. The connection is identified by its IP address. Once a router connects to the SRx-Server, it is not expected that the router performs validation requests of all updates it contains. For that reason, the SRx-Server allows configuring a synchronization request. In this case, the SRx-Server requests a validation request for all UPDATEs the router holds.

```
Display proxy mappings:
=======================
* Client[0x01](  1): Proxy ID 000.000.000.002 [0x00000002](0000000002) (-------/pre-conf) - #updates=0
* Client[0x02](  2): Proxy ID 010.000.006.050 [0x0A000632](0167773746) ( active/dynamic ) - #updates=16
* Client[0x0A]( 10): Proxy ID 010.000.000.001 [0x0A000001](0167772161) (-------/pre-conf) - #updates=0
* Client[0x19]( 25): Proxy ID 010.001.001.002 [0x0A010102](0167837954) (-------/pre-conf) - #updates=0
```

Listing 3 – SRX-Proxy mappings

The router does not only send validation requests. During the connection handshake, the router provides the SRx-Server with all peers it intends to send Updates to. The reason is to allow the SRx-Server to pre-compute signatures during idle times and then when requested, the SRx-Server can either provide just the signature or the complete BGPsec_PATH attribute with the peer and signature included. For this, a default pCount value is provided to the SRx-Server. This functionality is especially interesting for algorithms such as the current BGPsec selected ECDSA algorithm as specified in RFC 8608.

During testing on different systems, we identified issues with TCP traffic flow between SRx-Proxy and the SRx-Server. We observed the SRx-Proxy freeze during operations, which in turn affected the router and caused issues. To prevent this from happening we implemented traffic control on the sender and receiver side by allowing to buffer traffic. As shown in Listing 4, the SRx-Server has the receiver queue enabled by default to allow receiving more data while reading data with regular speed from the TCP input buffer. The buffer queue is variable in size and therefore only restricted by system memory. This allows the SRx-Server to keep receiving validation requests, even if the SRx-Server is under load.

```
Configuration:
==============
port.....................: 17900
loglevel.................: 5
sync.....................: true
rpki.host................: localhost
rpki.port................: 50000
bgpsec.srxcryptoapi_cfg..: (null)
console.port.............: 17901
mode.no-sendque..........: true  (send queue turned off)
mode.no-receivequeue.....: false (receive queue turned on)
```

Listing 4 – SRx-Server configuration settings

4.2.2. SRx-API and the SRx-Proxy

The SRx-API provides an interface to the SRx-Server that allows binding the SRx-Proxy library, a lightweight C-based implementation that functions as a wrapper to the SRx-Server. The SRx-Proxy allows easy SRx-Server integration into the client application. The SRx-Server connection can be freely configured from within the client by providing the proper configuration functions.

As shown in Figure 8, the SRx-Proxy shields all communication with the SRx-Server and provides three main methods to the client/router:

- validate
- sign
- callback

as well as some maintenance functions.

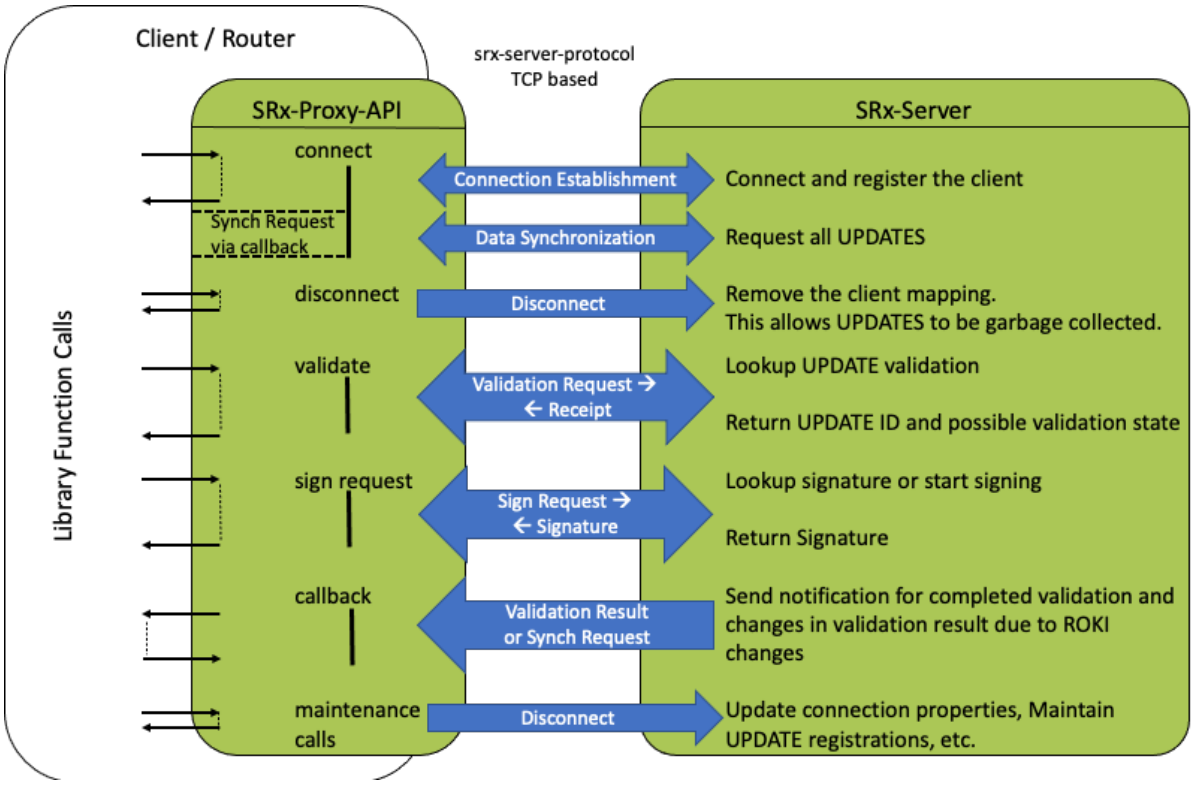


Figure 8 – SRx-Proxy

The validate function is used for both, BGPsec path validation as well as prefix origin validation. The callback function is used by the proxy thread to inform the router thread of the received data. The validation call is a blocking call until the receipt is returned containing the SUID that is needed for future communication between the SRx-Server and the router to identify the updates. The return is rather quick because the SRx-Server treats initial turnaround notifications with the highest priority.

At the current stage of the implementation, QuaggaSRx does not use SRx-Server for signing. Therefore, the implementation is not complete. Future plans for the signing include the SRx-Server being able to pre-sign updates. For this to function, the proxy provides the SRx-Server with a list of peers and a default pCount value used for each peer. At this point we decided to defer on using SRx-Server as a signer because signature generations are not as computationally expensive as validation requests.

The callback method is used when the SRx-Server sends notification messages to the proxy. In this case, the proxy calls the registered callback message and provides the necessary message information. This can happen, for instance, when the SRx-Server finishes a validation, or a validation result changes due to changes in the RPKI or key expirations. In this case the SRx-Server sends a notification to the client, which is received by the proxy. The proxy then calls the registered callback method to notify the client/router of the change. The notification call will be performed within the proxy thread.

The proxy also provides functions such as connect, disconnect, and maintenance calls. The connect and disconnect functions deal with the connectivity to the SRx-Server. Depending on the configuration of the SRs-Server the SRx-Server sends a synchronization request to the router after the connection is established. Even though Figure 8 shows this process attached to the connect function, the process is performed using a notification call which in return will be processed by the callback function. For simplicity, the initial synchronization call in Figure 8 is a logical placement. During the connection time, the SRx-Server might perform a synchronization call. This will happen once the SRx-Server identifies discrepancies within the statefulness of the connection.

4.2.3. The Telnet Client to SRx-Server

The SRx-Server provides a telnet socket that allows query of updates as well as triggering of cache dumps while operating. The dumps are in XML form and will be printed on the command line of the SRx-Server itself. They are not communicated to the telnet console. Also, these dumps can be rather large depending on the amount of data stored in the caches.

The telnet session is not only used to query UPDATE information, it also is used to shut down the server or initiate synchronization requests. The latter one helps the operator to re-synchronize the client/router and the SRx-Server in case the operator believes that the statefulness of the connection is compromised.

Once connected to SRx-Server via telnet, the operator can use the telnet session to verify the validation result by selecting the UPDATE's SUID. First, one finds the UPDATE in question from QuaggaSRx.

The SUID is the 4-byte hexadecimal value to the left of the UPDATE. As shown in Listing 5 the SUID for the valid UPDATE is *"70BFC549"*.

```
*> 70BFC549 v(v,v) +  25,       10.70.0.0/16     10.70.0.1              0     25s      0 70 ?
*  A903E850 i(i,v)          I   10.80.0.0/16     10.80.0.1              0              0 80 70 ?
```

Listing 5 – QuaggaSRx RIB-IN

Listing 6 shows how the Update Cache data of this UPDATE can be retrieved using the command: "*show-update id 70BFC549*".

```
[SRx]> show-update id 70BFC549
---------------------------------
UpdateID.........: 0x70BFC549 (1891616073)
 -Clients........: 0x02
 -AS.............: 70
 -Prefix.........: 10.70.0.0/16
 -ROA Count......: 953094771
 -Prefix Origin..: VALID
  * Default......: --
  * Source.......: ROUTER
 -Path...........: VALID
  * Default......: --
  * Source.......: ROUTER
 -ROA Coverage...: ROAs that render the update VALID...
                   AS(70), Prefix (10.70.0.0/16-20), ROACount 1
```

Listing 6 – SRx-Server Update Cache – Valid Update

In the current configuration, the router does not provide a default validation value. This is indicated by "---". The SRx-Server did both origin validation and BGPsec path validation and both validations returned 'Valid'. For prefix origin validation the SRx-Server displays the ROA entry that resulted in the VALID result.

To see why the second UPDATE for the same prefix was invalid, we query the given SUID, in this case "*A903E850"*. Using the same command "*show-update id A903E850*", the UPDATE cache shown in Listing 7 reveals the reason for the INVALID outcome. It becomes clear that the prefix 10.80.0.0/16, originated by AS70, was rendered invalid due to the ROA (10.80.0.0/16 with max length 20 for AS80).

Here, the origin AS does not match the ROA hence the UPDATE is invalid.

```
[SRx]> show-update id A903E850
---------------------------------
UpdateID.........: 0xA903E850 (2835605584)
 -Clients........: 0x02
 -AS.............: 70
 -Prefix.........: 10.80.0.0/16
 -ROA Count......: 4052041
 -Prefix Origin..: INVALID
  * Default......: --
  * Source.......: ROUTER
 -Path...........: VALID
  * Default......: --
  * Source.......: ROUTER
 -ROA Coverage...: ROAs that render the update INVALID...
                   AS(80), Prefix (10.80.0.0/16-20), ROACount 1
```

Listing 7 – SRx-Server Update Cache – Invalid Update

Appendix A.2.2 provides a complete list of all SRx-Server telnet commands and their function.

4.3. SRx Update ID Generation

The SRx Update ID (SUID) is essential within the BGP-SRx environment. Each update is identified by this uniquely generated ID and all routers/clients connected to the SRx-Server will have the exact same SUID for the exact same update. The SRx-Server only uses the SUID to communicate validation state changes to the attached routers to reduce traffic. The SUID itself is generated using a simple CRC32 algorithm, which is not collision free. This is the reason why each SUID must be calculated by the SRx-Server and not by the clients.

Let us assume for simplicity the SUID algorithm would just simply take the sum of the AS numbers in the path and each Router generates the SUID itself. As shown in Figure 9, the three updates "1-2-3", "2-1-3", and "3-2-1" all would result in the SUID "6". Therefore, using the clients to calculate the SUID would render the SUID useless.

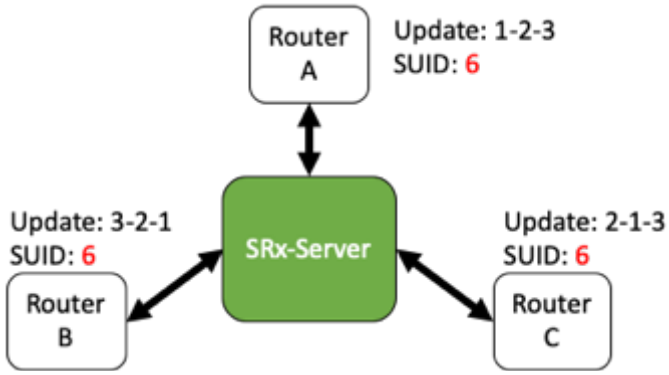


Figure 9 – Routers generate SUID

To reduce the data communication between the router and SRx-Server, each router does compute a local update ID (LOCID) which must be unique only within the router itself. As shown in Figure 10, the router performs the request using the LOCID. The SRx-Server then generates an ID unique to the SRx-Server and responds with a receipt containing both the LOCID and SUID.

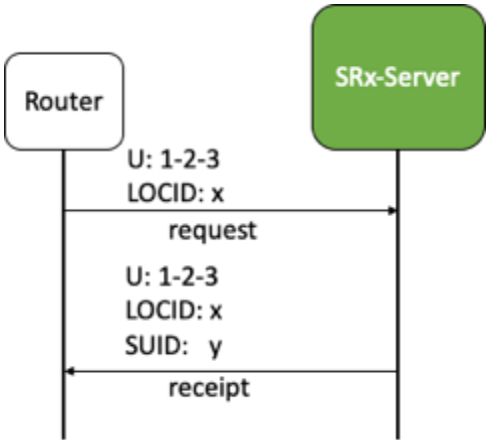


Figure 10 – Router and SRx-Server LOC/SUID exchange

From this moment going forward, the SRx-Server only communicates with the router using the SUID. As shown in Figure 11, the LOCID can be discarded by the router once the router received the SUID. The SRx-Server does not store the LOCID of the router.

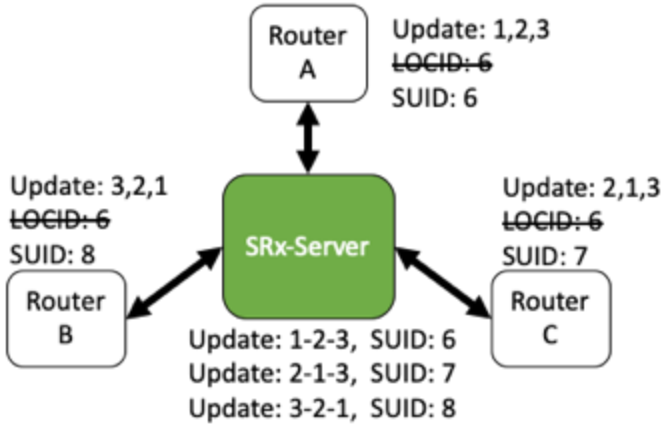


Figure 11 – SRx-Server generates SUID

Regardless of whether BGP-OV or BGP-PV are being used, the SRx-Server needs to generate the SUID for each request. Once the SUID is generated the SRx-Server verifies that the ID is not previously used by scanning the Update Cache for an update stored with the exact SUID. In case an update is found, the SRx-Server must compare the updates if it is the same. In case the Update is the same, that means is already stored, a receipt can be generated immediately. As explained in Section 4.5.1, the receipt contains not only the LOCID and SUID, it also contains the current validation view of the update itself.

In case the update found with the calculated SUID differs from the currently processed update, a SUID collision is detected. To mitigate the collision the SRx-Server simply increases the SUID by one. Now with the new SUID, SRx-Server again must verify that no other update is using this SUID. This process will be repeated as long as needed until either an unused SUID is retrieved, or an update is found that fully matches the update which is currently processed.

4.4. The Data Caches

As already mentioned in Section 4.1, the SRx-Server holds three internal caches, the Update Cache, the Prefix Cache, and the SKI Cache. Each update can be located through either of the caches. The reason for that is that modifications can be received in form of new UPDATEs, addition or removal of ROAs, and addition or removal of SKIs. The SKI Cache only holds the SKI of each key whereas the key itself is stored in the SRxCryptoAPI. Changes in the RPKI can have an impact on the validation state of previously validated updates. The following sections explain in more detail each of the caches.

4.4.1. The Update Cache

The Update Cache stores each BGP/BGPsec UPDATE in a hash table using the SUID as a key. This allows a quick location of the update within the update cache. Each UPDATE contains the AS path and if available the complete BGPsec path information as shown in Table 3. Furthermore, each update cache element contains information such as the route origin validation and BGPsec path validation values as specified in section 2.1.1.

Type	Name	Description
uint8_t*	clients	Clients with value 0 are unused
uint8_t	noPossibleClients	Maximum number of clients in the list without extending
SRxUpdateID	updateID	Systemwide unique ID
UT_hash_hanlde	Hh	The hash table where this entry is stored in
utin32_t	Asn	The Origin AS of this update
IPPrefix	Prefix	The IP Prefix
SRxResult	srxResult	The result generated by SRx
SRxDefaultResult	defaultResult	The result provided by verification request
uint32_t	roaRefCount	The number of ROAs that cover this update
uint16_t	gcFlag	Indicates when this entry can be deleted by the garbage collector.
UC_UpdateData	pathData	This element replaces the blob.

Table 3 – Update Cache – Cache Element

The UPDATE data itself is stored in the attribute pathData. It contains the BGPsec_PATH attribute, NLRI attribute, and the AS_PATH as a 4-byte array for easy path access. The cache element contains the data linked to the UPDATE. We will not explain every single attribute, but we will elaborate on some that deserve special attention.

The first attribute *clients* is a pointer to a dynamic array that stores the client IDs. SRx-Server holds a list of clients/routers. This list facilitates a reference on which routers/clients did send a validation request for the referenced UPDATE. This is required to keep the router/client informed about validation state changes. Once a router/client decides to be removed from the 'clients' list, the router must send an update removal (Section 4.6). Until then the SRx-Server will continue to send updates when they occur.

The attribute *updateID* is the unique ID of the update itself. The importance and how it is generated are explained in detail in section 4.3.

The next is a set of two attributes. The first one is *srxResult*, the second one is *defaultResult*. Each of these structures stores the origin validation as well as BGPsec path validation. Both attributes serve a different purpose and the range of values both can represent are different.

The attribute *defaultResult* stores the value that was provided by the initial validation request. Here the values can hold their validation results as specified in the respective RFCs for their validation and in addition, the value can be the SRx Specific value “undefined”.

The attribute *srxResult* can hold the value undefined, but in this case, the value stored in *defaultResult* will be returned to the requester instead regardless of its value. Once SRx-Server completed a validation, it stored the result in the variable structure *srxResult*. This validation result always contains values other than “undefined”. From this point going forward, the SRx-Server will always return the value stored in *srxResult*.

Modifying the BGP-OV and/or BGP-PV validation result of an update normally does cause the SRx-Server to send a notification to all routers that are affiliated with the update. To prevent unnecessary churn of notifications during an RPKI cache synchronization, these notifications can be temporarily paused and will resume once the synchronization is done. During such a synchronization, multiple ROA changes can cause the validation state of an update to change multiple times. During normal operation, this would trigger a notification to the router with each modification of the validation state. To send notifications to the router during RPKI updates, the SRx-Server waits until the end of the RPKI updates before validation state changes are communicated to the router.

To allow housekeeping and memory consumption, a router should unregister BGP UPDATEs once the router removes them from its RIB-IN. If the router does not unregister previously validated updates, the router will receive a validation state change notification each time the BGP-OV or BGP-PV state changes.

Once an update is not affiliated anymore with any router client, the garbage collection flag will be set. This allows the SRx-Server’s own garbage collector to remove the update. Not removing an update right away from the update cache allows the SRx-Server to cache the validation result for a little longer in case the update is received again. In this case update’s validation state is still maintained and does not need to be re-started when the router receives the update again.

4.4.2. The SKI Cache

Originally, the design called for a Key Cache to hold the keys within the SRx-Server. We moved away from that model once we started developing the SRxCryptoAPI (SCA) which includes a Key Storage (see Section 5.4). SCA allows using third-party key storages that come with the algorithm implementation. For the current implementation, the SRx-Server itself does not store the keys, it uses the default key storage of SCA. This allows removing the burden of secure key storage from the SRx-Server. With this out of the way, SRx-Server still needed to have a mechanism to identify changes in keys and the relationship between the keys and the UPDATEs. To achieve this, we implemented the SKI Cache. The SKI Cache is a data structure that stores SKIs known to the system. SKIs can become known through an UPDATE verification request or by receiving a key through the RPKI Validation Cache (RVC). In both events, the SKI will be stored.

The internal data structure is based on the AS numbers. The SKI itself is bound to an AS number and an algorithm ID. Looking at the AS numbers, which are 4-byte unsigned integers, we split them as shown in Figure 12 into two numbers, where each is two bytes in size. The upper 2 bytes field is called “upper” and the lower 2 bytes field is called “AS2”. At this point the majority of AS numbers are located in the same upper bucket and differ only in the AS2 bucket:

0x0000[0000] - 0x0000[FFFF].

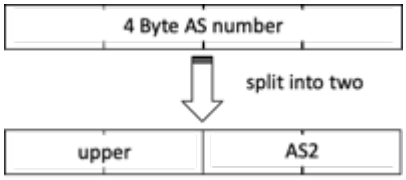


Figure 12 – SKI-Cache ASN splitting

The upper (left) bucket is relatively unused. For each value in the upper bucket, the cache reserves an array of 128 KB (64 KB * 2)[5], one element for each of the existing AS2 values within the bucket. This allows a direct access with O(1) access time. To keep the memory usage as minimal as possible but still have fast access as shown in Figure 13, the upper bucket uses a single linked list.

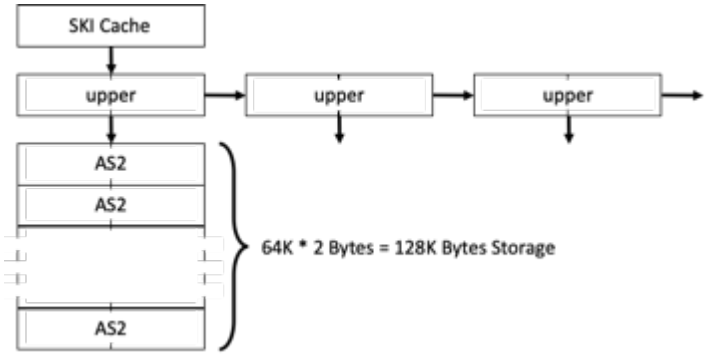


Figure 13 – SKI-Cache ASN storage access

Using this initial storage mechanism seems memory intensive but it provides access speed, and the SRx-Server can run on a regular system such as a desktop PC with the required memory, so benefits are worth the drawbacks. The initial question though is how to quickly find an SKI number. This does not help if one only has the SKI number, in this case, the search is very expensive. Considering the data that is available, each SKI is bound to an AS number. Therefore, looking up the ASN first is relatively fast. Once the proper AS2 number is located, the next step is to find the SKI number. Again, the SKI is bound to an algorithm ID therefore the next logical the is to attach the algorithm ID in form of a linked list. Currently, there is only a single algorithm available but as specified in RFC 8608 some values are reserved for experimentation so one can envision at this point the usage of experimental IDs. Underneath the algorithm ID then we added the data packet {SKI; ASN; AlgoID} (SAA). Of course, the ASN and AlgoID are redundant data but it eases the programming by having all data at hand,

[5] KB or K Bytes represent a unit in 1,024 bytes

especially if this data structure is handed over between functions. The last important part needed is the list of UPDATEs that use this SKI. For that, the SAA packet contains a pointer to a linked list of SUIDs.

Now when an UPDATE is stored in the Update Cache (see Section 4.4.1), the UPDATE is scanned for all existing SKIs within the BGPsec_PATH attribute. For each SKI the SKI Cache's data as shown in Figure 14 is scanned for the proper SAA packet and the SUID of the update will be added. When an UPDATE is removed (see Section 4.6) this link will be removed as well. If a key is received via the RPKI-Client, the Key is stored in the SCA using the SKI and the SKI will be stored in the SKI Cache. The SAA element contains some more attributes such as a counter to identify how many keys are stored using this SKI (a collision is highly unlikely but possible). Once the SAA packet is located and it already exists, each SUID will be sent to the RPKI Queue (Figure 7, Section 4.1) for BGPsec path re-evaluation. The same happens for any modification within the SKI cache triggered by an RPKI event.

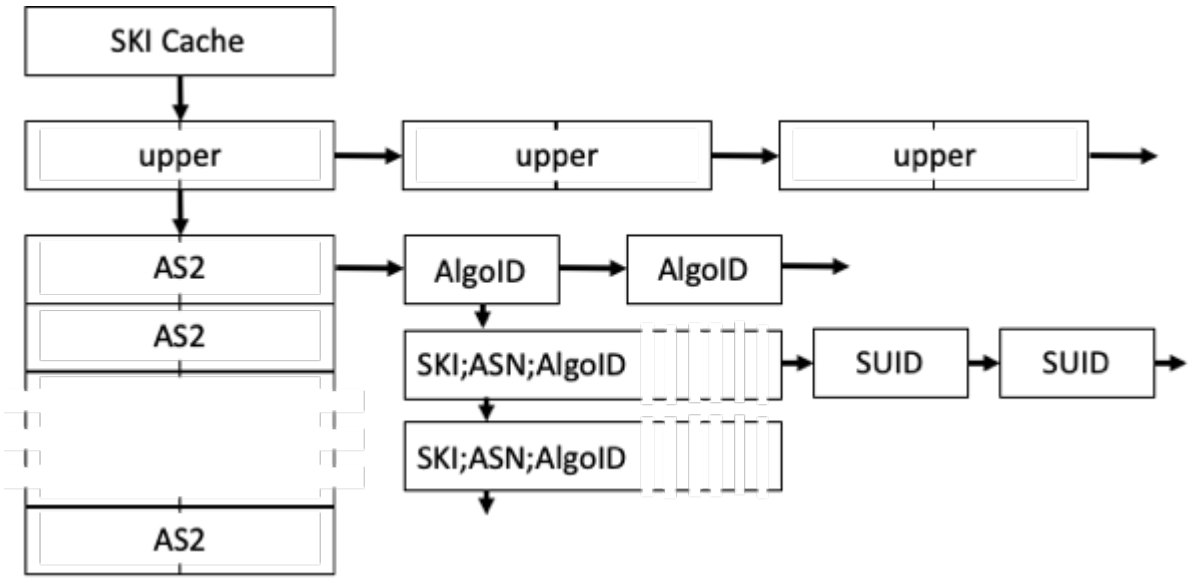


Figure 14 – SKI-Cache complete data structure

4.4.3. The Prefix Cache

The prefix cache is used for RPKI origin validation. The prefix cache uses internally a "Practical Algorithm to Retrieve Information Coded as Alphanumeric" (Patricia) Trie [25] to perform fast IP address prefix matching which is based on a radix tree using a radix of two.

The node content of the Trie is the prefix with the data structure as shown in Figure 15. The boxes with yellow background, *pc_Prefix*, *pc_Update*, *pc_ASx*, and *pc_ROA* specify the data components attached to the prefix.

The structure "pc_Prefix" contains two attributes and three different lists where two of them contain Updates and the third one contains all ASes. The AS list contains all ASes that originated the initial UPDATE as well as all ASes provided by ROAs that use that particular prefix. The two Update lists are called "*Valid*" and "*Other*". The idea of the two lists is very simple. Looking at all validated UPDATEs for a given prefix – here the origin AS might be different – only two possible combination cab exists at the same time for UPDATEs that share

the same prefix. The combination is either (*valid* and *not-found*) or (valid, invalid). Using the same RPKI data, no UPDATE can be "*invalid*" and "*not-found*" at the same time.

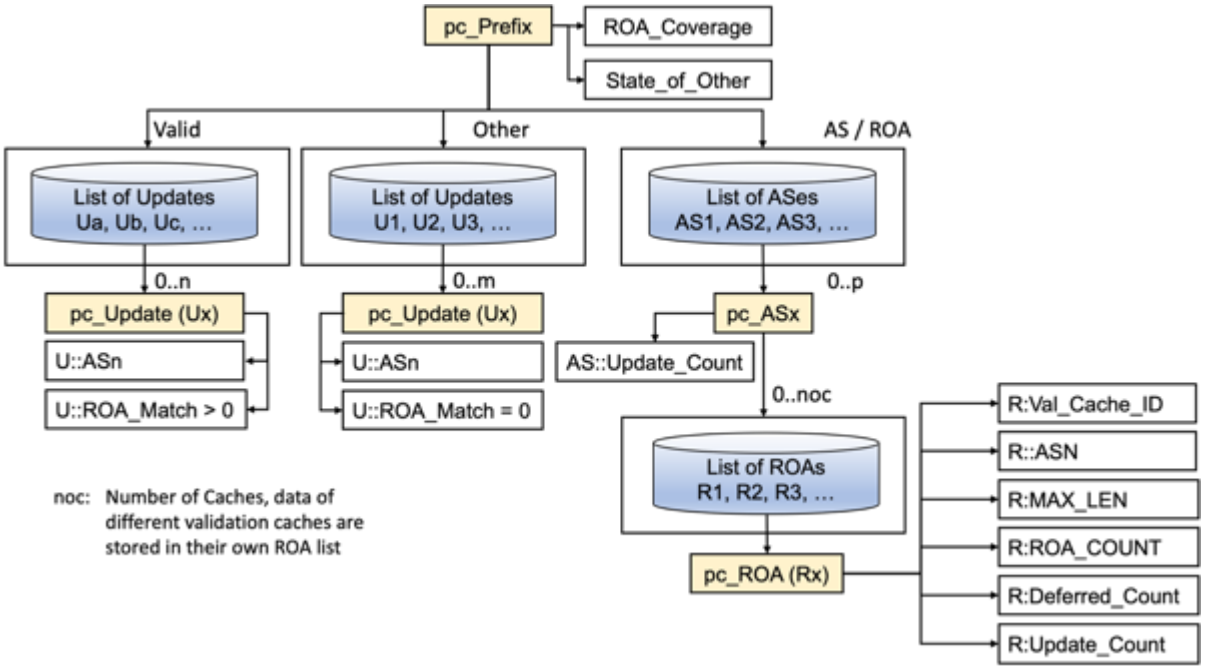


Figure 15 – The Prefix Cache

Besides the three lists, the *pc_Prefix* structure contains the attributes *ROA_Coverage* and *State_of_Other*. The attribute *ROA_Coverage* is a counter which gets increased with each ROA covering this prefix (including matches). This value helps to determine the validation state of "*Other*". In case the counter equals 0, the validation value of "*Other*" must be "*not-found*" otherwise the value of "*Other*" must be "*invalid*"

In case no ROAs are stored in the system, all updates will end up in the "other" list and the validation state of this list is labeled "*not-found*".

Figure 15 shows the complete structure of the prefix cache. The Prefix-Cache itself fulfills two major functions. It allows to determine the validation state of an update by simply adding the update into the prefix cache using the two update lists and it allows to manage ROAs and immediately identifies updates that are affected by modifications in the ROA storage. The ROA storage is part of the third list of the prefix, the AS list. This list contains all ASes that are either originators of the given prefix or listed in ROAs containing the given prefix. The structure pc_AS contains two attributes, the update count and an array of lists of ROA entries that list this particular AS and prefix combination. The update count must be equal to the sum of both update lists, "*Valid*" and "*Other*".

Within the SRx-Server, each RVC that SRx-Server is connected to has its own identifier *Val_Cache_ID*. Each ROA stored in the Prefix Cache contains the *Val_Cache_ID*. This allows identifying all ROAs attached to the particular RVC in cases such as loss of connectivity and a connection to a given RVC the tables must be cleared or an RVC requests a RESET. This can happen in case of a validation cache reset; in this case, the *Val_Cache_ID* is used to identify

the affected ROA entries. Each ROA has three counters, the *ROA_Count* which is the number of ROAs that are represented by this entry, *Deferred_Count* which is used during session resets between the SRx-Server and a validation cache, and the *Update_Count* which indicates how many updates are covered by this ROA.

Flow charts that show the in-depth cache management (storing & lookup) are listed in Appendix A.2.4. All flow charts are based on the data structure shown in Figure 15.

4.5. Processing UPDATE Validation

Validation requests occur in two different forms. One such request can be triggered by a router that sends a validation request of a particular UPDATE. This initial request is a "Router Triggered Validation Request".

The other validation request is triggered as a result of one or more updates within the RPKI. These updates may be due to the addition and removal of ROAs or the addition and removal of public BGPsec router keys and are sent to the SRx-Server via the router-to-cache protocol. Changes within the RPKI's ROA registration do affect prefix origin validation whereas changes in the RPKI's BGPsec Key registration do affect the BGPsec path validation.

These changes within the RPKI do trigger the UPDATE validation by modifying the Prefix Cache and SKI Cache respectively. The revalidation will be scheduled in the RPKI Handler's RPKI Queue.

1) **Router Triggered Validation Request**
   If an update validation request is received from the router, the Connection Handler within the SRx-Server that received the request will immediately generate a SUID (Section 4.3) and check if a validation must be performed. This is done by identifying if the UPDATE already exists in the Update Cache and if the proper validation flag is set. In case the UPDATE did not already exist, or the proper flag is not set, an initial validation is required. To do that, the SRx-Server adds a validation request command into the Command Queue. The Command Queue is processed by the Command Handler. Even though the implementation allows having multiple Command Handlers running in parallel to speed up the processing of the Command Queue, the currently used number of Concurrent running Command Handlers is restricted to one. Once the Connection Handler added the verification command to the Command Handlers Command Queue, the Connection Handler immediately starts the receipt generation and sends a receipt to the router. The Receipt generation is explained in detail in Section 4.5.1. The Command Handler then will fetch the command and trigger the requested validation.

2) **RPKI Triggered Validation Request**
   Another trigger relates to changes in the RPKI. Each time ROA information is added or removed in either the Prefix Cache or the SKI Cache, a revalidation of updates is triggered. This trigger is a function of the respective cache management that identifies updates possibly affected by the modification. In this case, the RPKI-Client that manages the SRx-Server and RVC connection triggers the respective cache management which can result in a re-validation request being added into the RPKI Queue.

4.5.1. Generating Receipts for Validation Requests.

Upon receiving a validation request from the client/router, the SRx-Server must not waste time and immediately return the receipt, as the router's UPDATE processing is blocked until it receives the validation request. Hence receipts are high priority messages that cannot be queued behind other validation result messages.

With that in mind, the SRx-Server receives the update information and generates an SUID. In case the SRx-Server already has an update stored with this SUID it will verify if the stored update is a binary exact match to determine if an SUID collision occurred or not. In case the updates match on a binary level, the SRx-Server retrieves the validation result of the already stored data and adds them to the validation receipt. In case the updates do not match, the SRx-Server increases the SUID by one until either no collision occurred, or the next collision is a match. This assures that in case two router clients perform the same request, both will be served with the same validation result.

As illustrated in Figure 16, AS 20 sends an update to its two peering points within AS 30. Both peering points, router A (at time t1) and router B (at time t2) receive the update with the same AS path and prefix origin information. Each router generates its internal data structure:

[Prefix, Origin, {Path}, Local-ID, { BGP-OV, BGP-PV }]

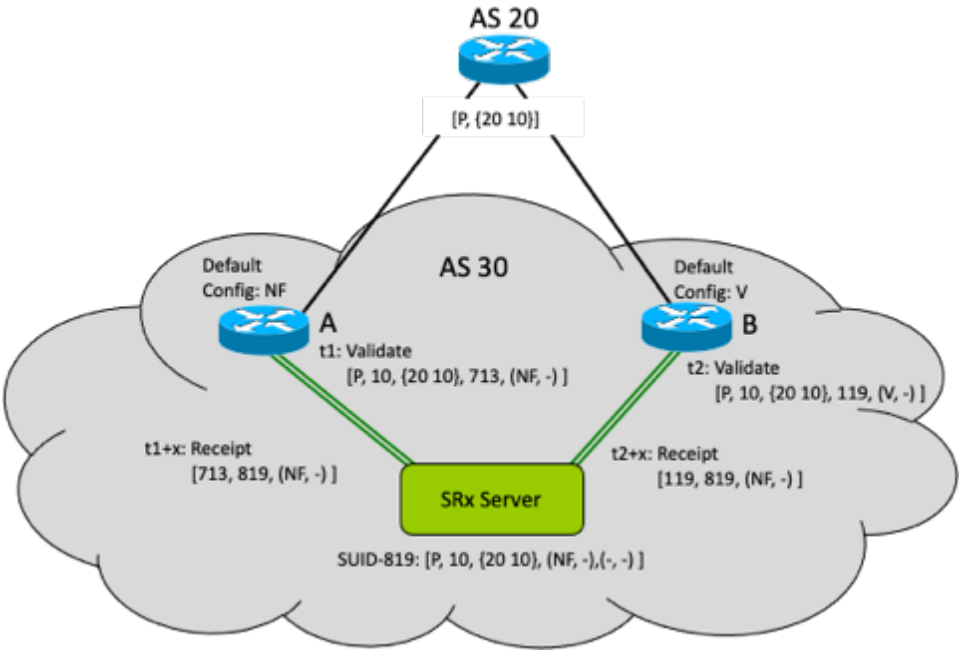


Figure 16 – Process validation request

Router A is configured to assign "not-found" as the default BGP-OV validation whereas router B is erroneously configured to assign "valid" as the default BGP-OV state to each newly received update. Router A is slightly ahead of router B and sends a validation request for RPKI origin validation to the SRx Server. It uses the internal update counter (local update id), which is assumed to be increased to position 713, adds the prefix P, origin 10, and as-path {20 10}

together with the configured default validation state of not-found (NF) for origin validation[6], and omits the default validation state for BGPsec validation. The SRx-Server used the triplet [P, O, path] and generated the SUID 819. In the next step, it attempts to find a previously stored update with this SUID, cannot find any, and stores the received update data using the SUID as the key. After that, the SRx-Server generates a receipt for router A and adds the ID (713) used by router A, the newly generated SUID (819), and the provided validation result values from router A. Router A now must associate the UPDATE with the provided SUID because from now on the SRx-Server will only refer to this UPDATE using the provided SUID. At this point router A can delete the locally generated ID. After the SRx-Server initiated the receipt it added the update to the internal validation queue.

In the meantime, at time t2, router B received the same update and prepared its validation request by packing prefix P, origin 10, path {20 10}, its own local generated ID 119, and the pre-configured route origin validation result valid (V) to the SRx-Server. Again, the SRx-Server used the necessary update information [P, O, path] and generated the SUID 819. It looks up the internal validation table and finds a previously stored update with the same ID. It performs a bit-level comparison and determines the update is the exact same one. It first verifies the validation result and figures out no validation is performed yet (-,-). Now to assure all clients are synchronized regarding validation states, it uses the previously stored, pre-defined validation value NF and generates the receipt for router B. The receipt contains the local ID of router B (119), the SUID (819), and the validation result (NF,-). Router B must use this validation result and apply it to the route because the SRx-Server only sends further notifications to all its clients once the validation state changes. If the SRx-Server returns from the validation process with the correct[7] validation result NF, it will store NF in the stored update but will not send any notification to the routers. SRx-Server knows it provided NF to the routers within the validation request receipt, so no further action is required from the router.

4.5.2. The Router Triggered Validation Process

The moment SRx-Server receives a validation request, the SRx-Server takes the update and generates an SUID and looks up if the route already exists. In case the route exists and is already validated, the validated result is taken and returned to the router piggybacked in the return receipt. In case the update is known but not validated yet, the router will return the default values within the stored UPDATE and returns this value back to the router. The reason for this is that a validation request for this particular update was already received by the SRx-Server. In this case, the SRx-Server assures that all assigned routers do have the same validation view on all UPDATEs in the system. In case the UPDATE is not already stored, the default validation result provided by the router will be selected and returned within the validation receipt to the validation requester. In this case, a validation request will be scheduled in the Command Queue.

Once the Command Queue fetches the validation request it will start the requested validation. SRx-Server has two distinct validation request methods. The Prefix Origin Validation request and the BGPsec Path Validation Request. Both are triggered by adding the given information into the corresponding Cache. For Prefix Origin Validation the Prefix/Origin information of

[6] BGPsec is omitted in this example for simplification.
[7] Following RFC 6811

the UPDATE will be stored in the Prefix Cache (see Section 4.4.1) and for BGPsec path validation the SKI information will be stored in the SKI Cache (see Section 4.4.2)

4.5.3. RPKI Triggered Validation Request

These validations are triggered by changes in the RVC. This is a third-party software that fetches ROA End Entity (EE) Certificates including Certificate Authority (CA) Certificates and validates the RPKI data. This includes the validity of ROAS and all the certificates up to the root. Periodically the RVC can send notifications to its clients notifying them to poll for updates. The clients, the SRx-Server in this case, can then decide to poll or to wait. In case the router waits for too long, the RVC requests a reset to assure the client is up to date with the RPKI. During these updates, receiving new ROAs, Keys, or deletion such, the SKI Cache and Prefix Cache undergo some churn. In case ROAs get removed and new ones added, each of these operations does modify the validation state of updates. Once this happens, the modification of the validation state will if trigger a notification to the client. To prevent unnecessary churn for routers, validation state change notifications are withheld as long as the synchronization with the RVC is in progress. No notifications other than validation receipts are sent to the clients/routers. Once the RVC synchronization is done, all notifications will be sent. The addition of a ROA can change previous validated UPDATEs from *Not-Found* to *Valid* or *Invalid*. Removal of such can change prefix origin validations from *Valid* to *Invalid* or *Not-Found* and *Invalid* to *Not-Found*.

In case a single UPDATE changed its validation state n times, n notifications will be scheduled but all notifications will use the final validation result. When SRx-Server schedules a notification, it does not store the validation state in the schedule. It stores the SUID instead that then is used once the notification packet is generated to retrieve the latest known validation state from the update stored in the Update Cache. This assures intermediate results during a cache update never make it to the router with this, the router might receive the same notification multiple times, the decision process itself will be trigger only once at the most.

4.6. Processing Update Removals

Removing an UPDATE by a router services two purposes:

i) reduce unnecessary notification traffic between client and SRx-Server and

ii) reduce memory and validation processing associated with the UPDATE

A router that receives an UPDATE and sends it to the SRx-Server for validation, will receive notifications for validation state changes of the UPDATE for as long as the router is registered with the UPDATE. Once a router withdraws the UPDATE it can send a delete request to the SRx-Server. This will remove the tight relation between the router and SRx-Cache. Only UPDATEs that have no client attached can be discarded, once the SRx-Server decides to start the internal garbage collection to revive previously used memory.

4.7. Data Synchronization

Once a server connects to the SRx-Server, the synchronization request triggers the router to send requests for all UPDATEs in its RIB-IN to the SRx-Server for validation. This assures that not only validation requests are sent once the connection is established and previously received UPDATEs are left out. It also helps in case the SRx-Server needed to reboot to re-gain the validation state. During the synch request, the router should fill the default validation

state with the validation state previously received by the SRx-Server. This will most likely assure to keep the churn related to validation state changes low because most if not all validation results will be the same. Discrepancies will happen only for cases where the state of the RPKI is different.

Data synchronization between RVC and the router will happen more regularly, especially in older RVC cache implementations, but is definitely to be expected if the timeframe between two update polls is too big.

4.7.1. Router Synchronization and Reset

When a router starts operating but does not have connectivity to the SRx-Server, it most likely will not have any UPDATE validation performed. Once the connection to the SRx-Server is established, the SRx-Server will immediately send a synchronization request if it is configured to operate in that way. When the SRx-Server receives such a request, it will go through the RIB-IN and resend the UPDATEs it learned up to that moment in time. In case the synch reset happens as a result of a temporary connectivity loss with the SRx-Server, the router will have some UPDATEs that were previously validated and some that are newer but not validated, the router will send all UPDATEs and should add the previous validation result as default values into the request. This assures that no validation state changes will be received except the state of RPKI changed during the connection loss. The current implementation allows for a router to keep the initial SUID and most likely the SRx-Server will generate the same SUID. Having said that, it is highly recommended to replace all SUIDs with local IDs in case the synch request occurred during normal operation, to account for the case in which the SRx-Server performed SUID collision mitigation on the server side. In such a case the collision mitigation will depend on the order in which UPDATE validations are received to guarantee a match. In case the local ID within the verification request is zero, no Notification receipt will be sent by the SRx-Server.

4.7.2. RPKI Validation Cache Synchronization and Reset

The current implementation of the SRx-Server does not provide the RVC synchronization reset functionality. This is a section that still needs to be added, though the foundations do exist: This functionality would need to freeze the current roa_count and refresh the cache information. Now with each ROA added the new counter “referred_count” will be increased instead of the roa_count. Once the cache update is completed all ROAs for the given cache_id with a deferred_count = 0 will be removed. Using this mechanism allows for a simple and quick cache refresh, with a minimum of churn assuming the state of the RPKI did not dramatically change.

4.8. Future Considerations

Currently, SRx-Server provides an implementation for RPKI Prefix Origin and BGPsec Path validation, though it still has some shortcomings. Going forward, three sections would be useful to finish:

- Allowing SRx-Server to pre-sign BGPsec UPDATEs. The information required for this operation is available. Each client has its peers pre-registered including a preset pCount value. Resending the same UPDATE can easily reuse the same signature. That would remove the issue of non-deterministic signatures without risking security.
- Finishing the RPKI reset functionality and allowing the usage of multiple caches.
- Adding a reference implementation for ASPA path validation [32] which is currently discussed at the IETF SIDROPS working group.

5. The SRxCryptoAPI

The SRxCryptoAPI (SCA) is an interface to BGPsec cryptography. It is a library that functions as a wrapper for supported BGPsec cryptographic implementations. At the time of this document, BGPsec specifies only one cryptographic algorithm: the ECDSA algorithm [6].

SCA allows outsourcing all cryptographic operations as well as key management, so the router or any other implementing entity such as the SRx-Server does not need to be burdened with the extra cryptographic operations. An algorithm change can be deployed faster if it is kept independent from the router. This also removes the requirement of any cryptographic capability by the router itself or the BGPsec implementing validation server such as the SRx-Server (Section 4). SCA is implemented as a wrapper API that can be configured to load dynamically algorithm implementations as well as key management implementations. For router implementations that do not provide any key management, SCA provides a basic non-secure key storage, though mainly for testing purposes. This allows the router or validation engine to perform calls to SCA to store public and private keys without the need of implementing its own key storage. Also, it allows developing platform-independent key storage for the SCA.

5.1. Basic Design

The basic design of SCA is very simple. The API provides a set of functions that will be linked to an API provided by a third party. The linkage of the API to the provided library can be done using the configuration script. The API must provide two basic functions, *sign* and *validate*. Figure 17 elaborates the broader flow chart of a regular BGPsec validation as it is specified in RFC 8205 and implemented in the NIST BGP-SRx framework.

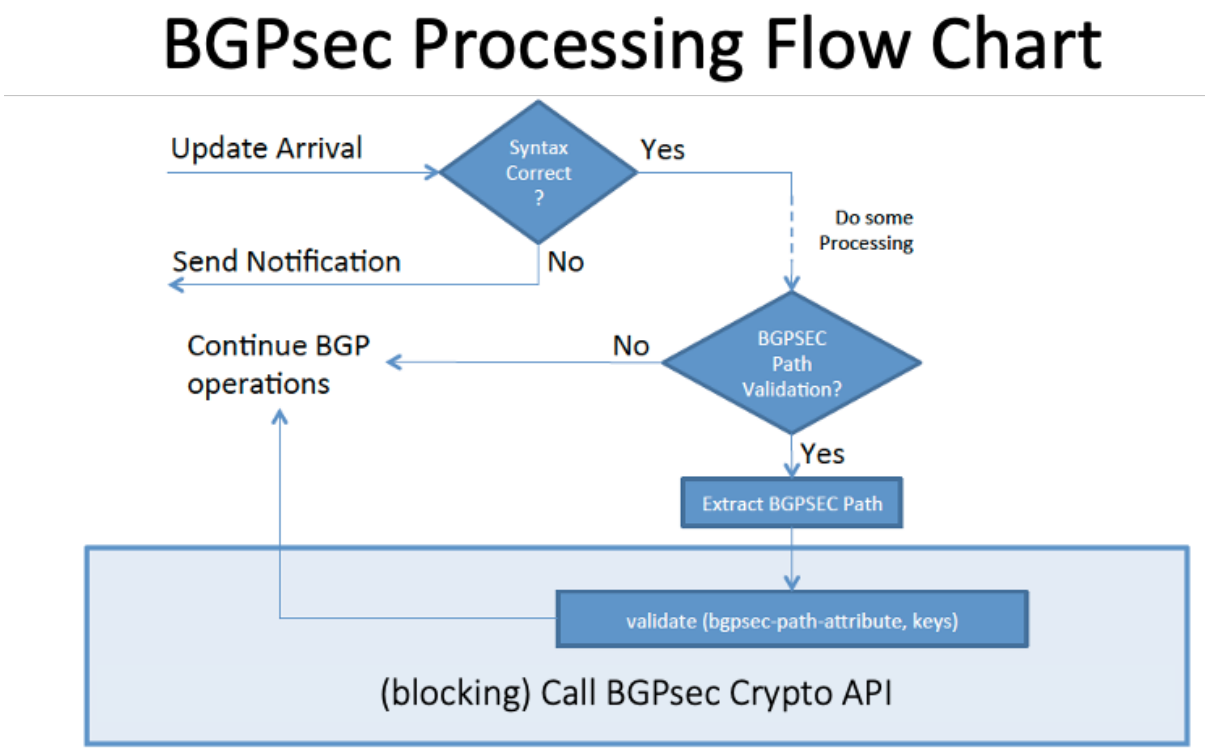


Figure 17 – BGPsec processing

Once the router receives a BGPsec UPDATE it first needs to verify the syntactical correctness of the UPDATE. After it is deemed syntactically correct, the router must decide whether to start validation or defer validation. In case the router decides to start validation, it extracts the BGPsec Path Attribute and the MP_NLRI Attribute which contains the prefix from the UPDATE and performs a blocking validation call to the SRxCryptoAPI. The signing of an update is not described here specifically but it functions the same way. The call is blocked until SCA finishes the cryptographic process of validation and returns with the validation result.

During the implementation and testing, we extended SCA by adding key loading mechanisms which resulted eventually in the SCA providing file-based key storage. Also, the implementation for parsing the BGPsec Path Attribute was added as an API function call to be used by cryptographic implementations.

5.2. Crypto Module Configuration

The SRxCryptoAPI configuration consists of two main sections. As shown in Listing 8, the first section specifies general configuration settings for the wrapper API. The second section is an embedded configuration for the crypto module.

```
# Name of the crypto module configuration to be used.
library_conf="bgpsec_openssl";

# Root folder for the keys
key_volt = "/var/lib/bgpsec-keys/";

# Specify the key file extensions for private keys DER encoded and
# public keys embedded in X509
key_ext_private="der";
key_ext_public="cert";

# Specify the debugging type which indicates only information that
# matches the debugging type or are less in its numerical value are
# displayed.
# The following debugging types are available:
#     0: LOG_EMERG   - system is unusable
#     1: LOG_ALERT   - action must be taken immediately
#     2: LOG_CRIT    - critical conditions
#     3: LOG_ERR     - error conditions
#     4: LOG_WARNING - warning conditions
#     5: LOG_NOTICE  - normal but significant condition
#     6: LOG_INFO    - informational (DEFAULT)
#     7: LOG_DEBUG   - debug-level messages
#debug-type = 6;

# Default NIST BGP-SRx crypto module configuration
bgpsec_openssl: {
  …
}

# High speed crypto module configuration
tara_crypt: {
  …
}
```

Listing 8 – SCA main configuration

As illustrated by the configuration above, SCA allows multiple module configurations to be specified within the same configuration script. The reason for that is to prevent requiring multiple configuration files. The main section of the configuration then allows selecting which crypto module configuration will be used. By default, and if not programmatically changed by the implementing client of SCA, the SCA API will look for the file *srxcryptoapi.conf* located in the folder */etc*. This behavior will change if the code is compiled using a different ***--prefix*** setting. The default setting for prefix is "***--prefix=/***".

There might be situations where one might be able to specify different configurations. This can be achieved by programmatically setting the location and name of the configuration file that will be used by SCA.

The configuration of the crypto module as shown in Listing 9 specifies the filename of the library to be used including an initialization string for the library. Furthermore the configuration allows mapping the implemented function names to the function names as specified in the SRxCryptoAPI's header file in case the differ. Even though the API proposes the function names to be used, implementers might choose a different naming convention. The SCA configuration allows mapping all functions that do not follow the API specifications to the chosen implementation. The only hard requirement for mapping is the function signature as specified in the API's header file '*srcxryptoapi.h'*. That means the types and order of parameters within the function. This must be followed, otherwise the crypto module cannot be used.

```
bgpsec_openssl: {
  # The name of the library
  library_name = "libSRxBGPSecOpenSSL.so";

  # A String e.g. "PUB:<filename>;PRIV:<filename>" or "NULL"
  # as initialization parameter.
  init_value                  = "NULL";
  method_init                 = "init";
  method_release              = "release";

  method_freeHashMessage      = "freeHashMessage";
  method_freeSignature        = "freeSignature";

  method_getDebugLevel        = "getDebugLevel";
  method_setDebugLevel        = "setDebugLevel";

  method_isAlgorithmSupported = "isAlgorithmSupported";

  method_sign                 = "sign";
  method_validate             = "validate";

  method_registerPublicKey    = "registerPublicKey";
  method_unregisterPublicKey  = "unregisterPublicKey";

  method_registerPrivateKey   = "registerPrivateKey";
  method_unregisterPrivateKey = "unregisterPrivateKey";

  method_cleanKeys            = "cleanKeys";
  method_cleanPrivateKeys     = "cleanPrivateKeys";
};
```

Listing 9 – SCA cryptographic module configuration

Furthermore, an implementation does not need to provide all functions that are specified by the API. In this case, the SCA will provide a skeleton implementation to prevent segmentation faults in case a function is not implemented. To a certain extent, the skeleton implementations do provide no real functionality other than hard-coded result values. This one exception is the implemented key storage which provides a file system storage for keys for test and demonstration purpose only.

The configuration setting "*init_value*" will be handed over to the library the moment the library is loaded into memory. This can be any string or NULL. The string value depends on the crypto module implementation. It could be the path to a different configuration file or a list of

parameters the library will parse. In case no parameter is provided SCA required this parameter to be set to “NULL”.

To allow proper memory management the API provides functions such as “freeHashMessage” or “freeSignature”. This is important if the crypto module was using allocation mechanisms that are not aligned with the SRxCryptoAPI. Therefore, it is recommended to use these messages for all memory allocations performed by SCA to prevent possible segmentation faults.

5.3. BGPsec Path Processing

The core API does provide more functionality than just functioning as a wrapper for crypto implementations. The SCA API provides a helper function for API implementations. One of these helper functions is the capability to process the BGPsec path. SCA allows to parse through the BGPsec_PATH attribute as specified in RFC 8205 and create a list of pointers into the provided data stream. These pointers allow direct access to secure path segments and more important the digest message used within the BGPsec Path Attribute. Therefore, the validating cryptographic algorithm does not need to parse the BGPsec Path Attribute but can take advantage of the provided functionality.

5.4. Key Storage

It is not trivial to provide a generalized key storage. Keys need to be stored in a physically safe manner so they cannot leak to the outside, especially when it comes to private keys. For this implementation, we decided to ease this requirement. We do not provide a secure file vault, as for our purpose for the reference implementation it is enough to base the storage on simple file system-based key storage. The keys, both private as well as public, are stored in the same directory. The default storage system used a distribution system where the SKI becomes a portion of the file structure. The root of the key filesystem-folder is specified using the ‘*key-vault’* setting in the SCA configuration. Within this root, two files are present: one is used for the private key-SKI-AS mapping and the other one for the public key-SKI-AS mapping.

The file structure of the keys as depicted in Listing 10 reflects the SKI for each key. The first two left most hexadecimal (hex) numbers of the 20 byte or 40-character wide SKI represent the first folder.

```
.
├── 18
│   └── 494D
│       ├── AA1B2DFD80636AE943D9DC9FF42C1AF9D9.cert
│       ├── AA1B2DFD80636AE943D9DC9FF42C1AF9D9.der
│       └── AA1B2DFD80636AE943D9DC9FF42C1AF9D9.pem
├── 3A
│   └── 7C10
│       ├── 4909B37C7177DF8F29C800C7C8E2B8101E.cert
│       ├── 4909B37C7177DF8F29C800C7C8E2B8101E.der
│       └── 4909B37C7177DF8F29C800C7C8E2B8101E.pem
├── priv-ski-list.txt
└── ski-list.txt
```

Listing 10 – SCA cryptographic module Key Vault

This assures that max 256 directories are generated within the key vault’s root structure. Then within each of these directories, the subdirectories consist of the next 4 hex numbers. For test

purpose, we do not anticipate a full number of 2^{16} directories but if this would be the case one could also reduce the second folder to only use the next 2 hex values instead, followed by another 'n' hex values. In this implementation, a 4-hex digit wide naming structure was chosen. Beyond that, the next layer consists of all keys, named with the remaining 34 hex numbers.

The AS – SKI mapping for private and public keys is done using the files priv-ski-list.txt and ski-list.txt.

Both files, `priv-ski-list.txt` and `ski-list.txt` as shown in Listing 11 are structured with the same format. The first entry is the AS number for the key, followed by "-SKI:". After that follows the SKI hex value which represents a 20-byte value or 40 characters.

```
2-SKI: 47F23BF1AB2F8A9D26864EBBD8DF2711C74406EC
5-SKI: 3A7C104909B37C7177DF8F29C800C7C8E2B8101E
7-SKI: 8BE8CA6579F8274AF28B7C8CF91AB8943AA8A260
50-SKI: FB5AA52E519D8F49A3FB9D85D495226A3014F627
60-SKI: FDFEE7854889F25BF6ECB88AFAF39CE0EBC41E08
70-SKI: C38D869FF91E6307F1E0ABA99F3DA7D35A106E7F
80-SKI: 18494DAA1B2DFD80636AE943D9DC9FF42C1AF9D9
90-SKI: 63729E346F7D10E3D037BCF365F9D19E074884E6
```

Listing 11 – Content of priv-ski-list.txt / ski-list.txt

The SCA API provides functionality to read and load the keys but a more secure alternative would be that the implementing crypto library provided the functionality for key management. In a future implementation, one can envision to also provide a second API interface just for key management.

Our analysis of BGPsec validation performance confirmed that cryptographic processing is not a bottleneck, but key management is. The main problem is located in key management: Each BGPsec UPDATE contains an average of 4 hops and therefore 4 signatures. This means that the validating engine must contain a large enough key storage in memory to not permanently load keys from disk, which will slow down the processing due to the read and write operations being orders of magnitude slower than in memory. Additionally, the operation of locating keys within the storage adds to the bottleneck.

5.5. BGPsec Cryptography Implementation BGPsecOpenSSL

The SRxCryptoAPI provides an OpenSSL-based crypto implementation for BGPsec. The internal cryptographic implementation ECDSA within OpenSSL is not as optimized as it could be and validates an average of 4.7 K path segments per second which results in approximately 1 200 UPDATEs with a length of 4 hops. This will approximately amount to 12 minutes of validation time for 850 K UPDATEs with an average length of 4 hops. This does lack high-speed production grade, though this module is functioning and good enough to run smaller tests and serve as an example build.

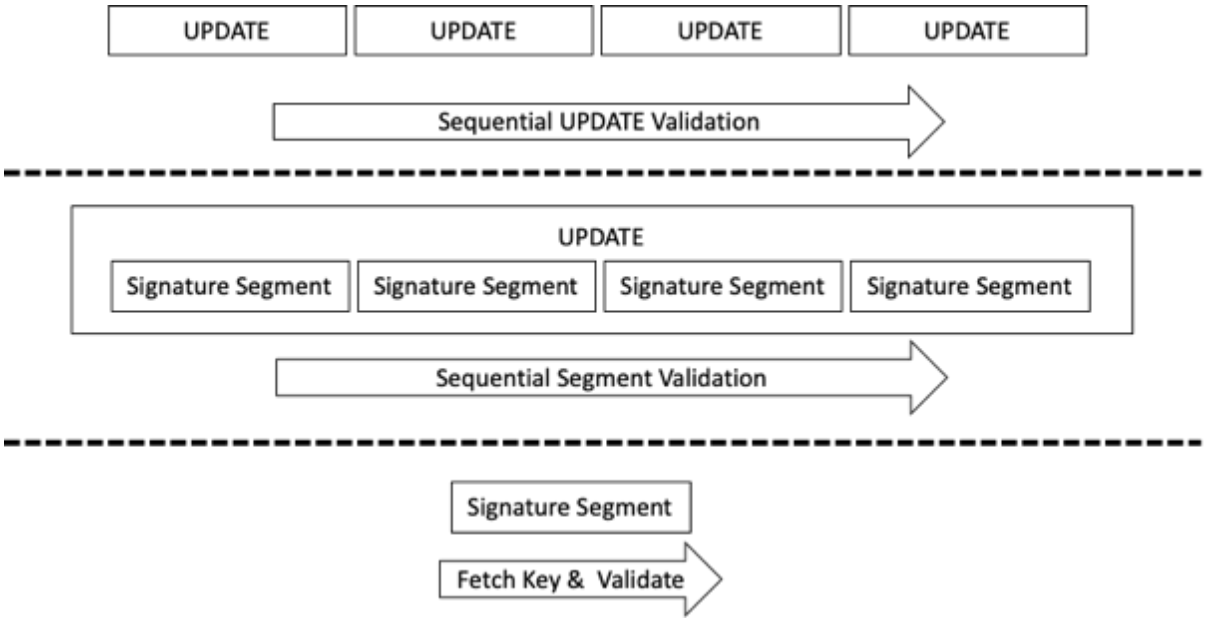


Figure 18 – BGPsecOpenSSL sequential cryptographic module workflow

The current BGPsecOpenSSL implementation (Figure 18) validates each UPDATE and its segments sequentially. Here the segments are an abstracted representation of the data and signature needed for each validation. This can be further optimized by parallelizing both, the processing of UPDATEs as well parallelizing the validation of each UPDATE's signature segments. Besides the cryptographic operation, loading the keys into memory is a very expensive operation. Caching certain keys such as the keys of immediate peers and possibly their peers would already amount to an estimated 50 % of the keys needed in an average 4 hop path.

Further improvements can be made by stopping the update validation process as soon as an invalid segment is detected – though this optimization is not expected to show a measurable impact since otherwise it would mean most updates are invalid which in itself would not be a good outcome.

The current tests were done using the BGPsec-IO implementation which allows testing the cryptographic validation of the implementation without the overhead of BGP processing.

5.6. High-Speed Implementation *tara*BGPsec™

To achieve a production-grade performance, NIST awarded an SBIR cooperative agreement 70NANB14H289 to Antara Teknik LLC to implement a high-speed implementation module [24] for the SRxCryptoAPI.

*tara*BGPsec™ is Antara's high-performance software library that implements security functionality for BGPsec as defined in RFC 8205. *tara*BGPsec is designed as a dynamical plugin library for the SRxCryptoAPI, the open-source reference implementation and research platform developed by NIST. *tara*CRYPT™ is Antara's efficient cross-platform crypto library that provides one of the fastest Elliptical Curve Digital Signature Algorithm implementations. It ensures that *tara*BGPsec performance meets strict Internet routing table convergence requirements.

As shown in Figure 19 a typical BGPsec sign operation includes: 1) Gathering and assembly of the BGPsec update data to be signed; 2) ECSDA Signature generation over the hash of the path data; and 3) asn1 encoding of the generated signature.

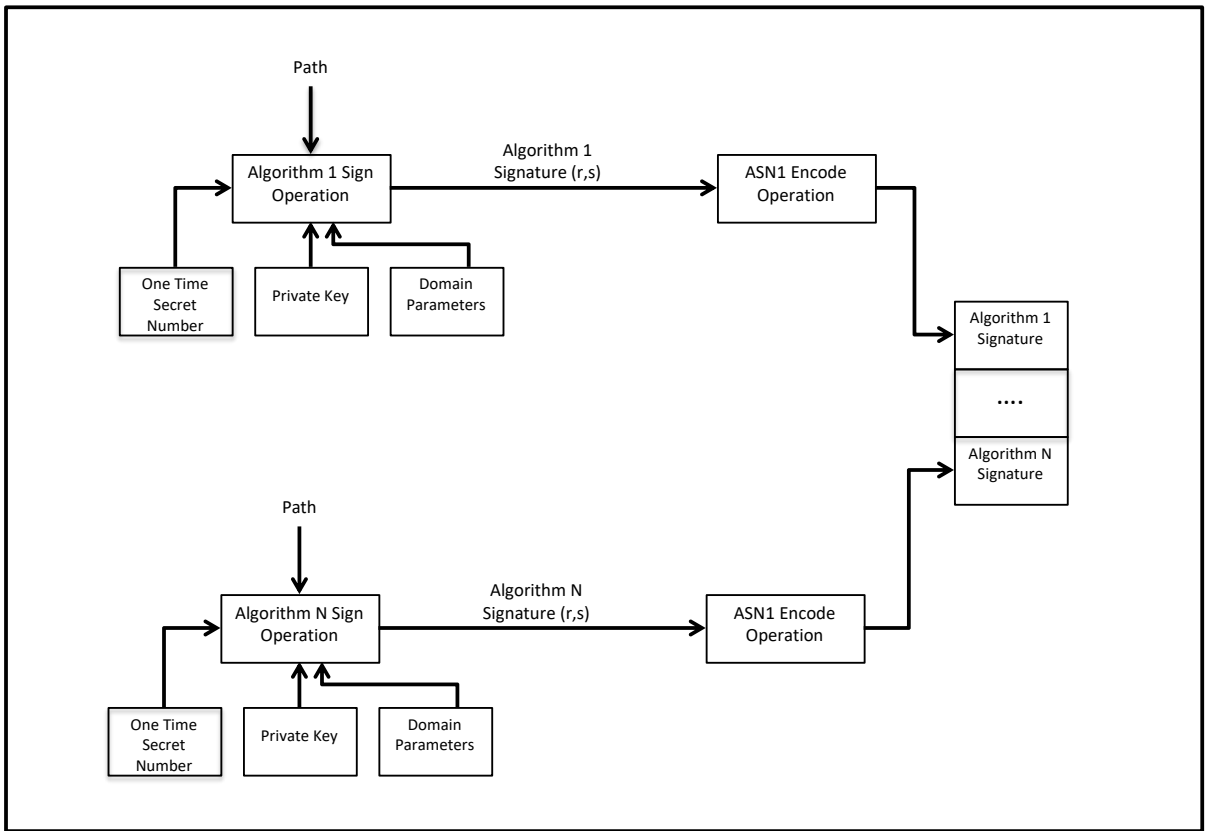


Figure 19 – BGPsec sign operation steps

As shown in Figure 20, a typical BGPsec Path Verification operation includes: 1) Parsing of update packets and the assembly of the path segment data; 2) Parsing and decoding of signatures for each segment (hop); 2) Fetching required public keys with assured integrity and decoding for each hop; 3) ECDSA verification of the signature over the hash of each segment data; 4) Assembling the results of running verification on each hop.

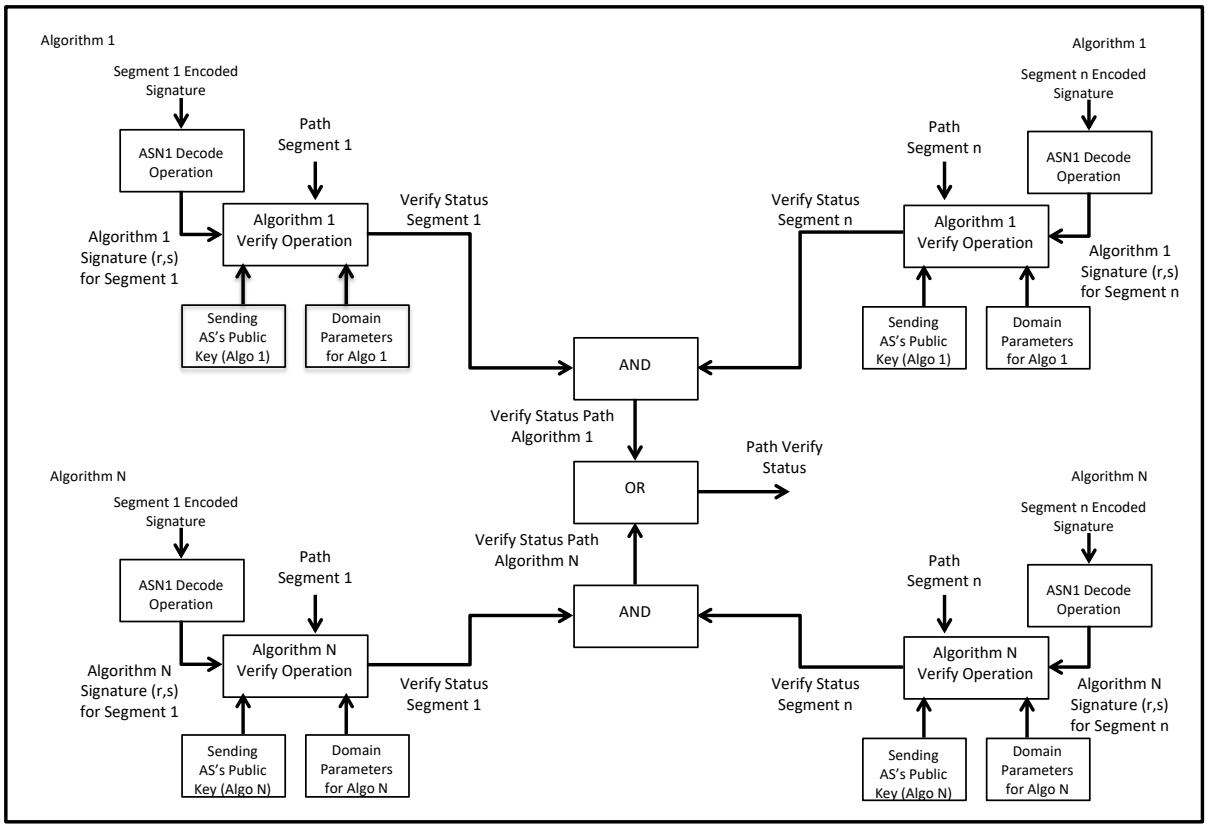


Figure 20 – BGPsec verification operation steps

The computational complexity of the signature generation and verification operations necessitate a high-performance solution. *tara*BGPsec is designed with advanced thread management mechanisms and seamlessly scales across CPU cores with an average core utilization of 80 %. While the average number of hops in a typical segment is around four, there are a substantial number of segments with one or two hops. Therefore, *tara*BGPsec uses a cognitive framework to dynamically decide on the number of available cores to utilize based on the load in order to minimize power consumption while sustaining high performance. Figure 21 shows a high-level view of the *tara*BGPsec Operations Parallelization Manager.

At a macro-level, the Secure Signature and Verification Operations Parallelizing Manager (SSVOPM) is responsible for managing and triggering the execution of the network protocol requests in a secure and optimized fashion. Therefore, it manages a collection of rule-based procedures to ascertain the optimal methods to serve the network protocol requests while maintaining data integrity, confidentiality, and availability. Furthermore, it creates and dispatches adjustable "Execution Agents," which contain asynchronous and duty-specific "worker" routines, specifically optimized to available compute resources, and creates "Task Pools" as staging areas for tasks to be fetched by the Execution Agents. In the current implementation of *tara*BGPsec, the secure signature and verification operations parallelizing manager binds itself to a specific computing unit core and then creates and dispatches Execution Agents as threads or processes and binds each to the remaining compute cores.

Upon receiving a sign or verify path request, the SSVOPM validates the request parameters, performs Intrusion Detection Mechanisms as applicable, and places task requests into a "Task

Pool" with the appropriate priority. The SSVOPM does not have to wait for the execution to complete and can issue back-to-back task requests. When an Execution Agent completes a task, it places the status and any output data (e.g. signature of a path) into a "Completed Task Pool," and continues with the next task. The secure signature and verification operations parallelizing manager is responsible for managing the results in the "Completed Task Pool" and fulfilling the original network protocol request.

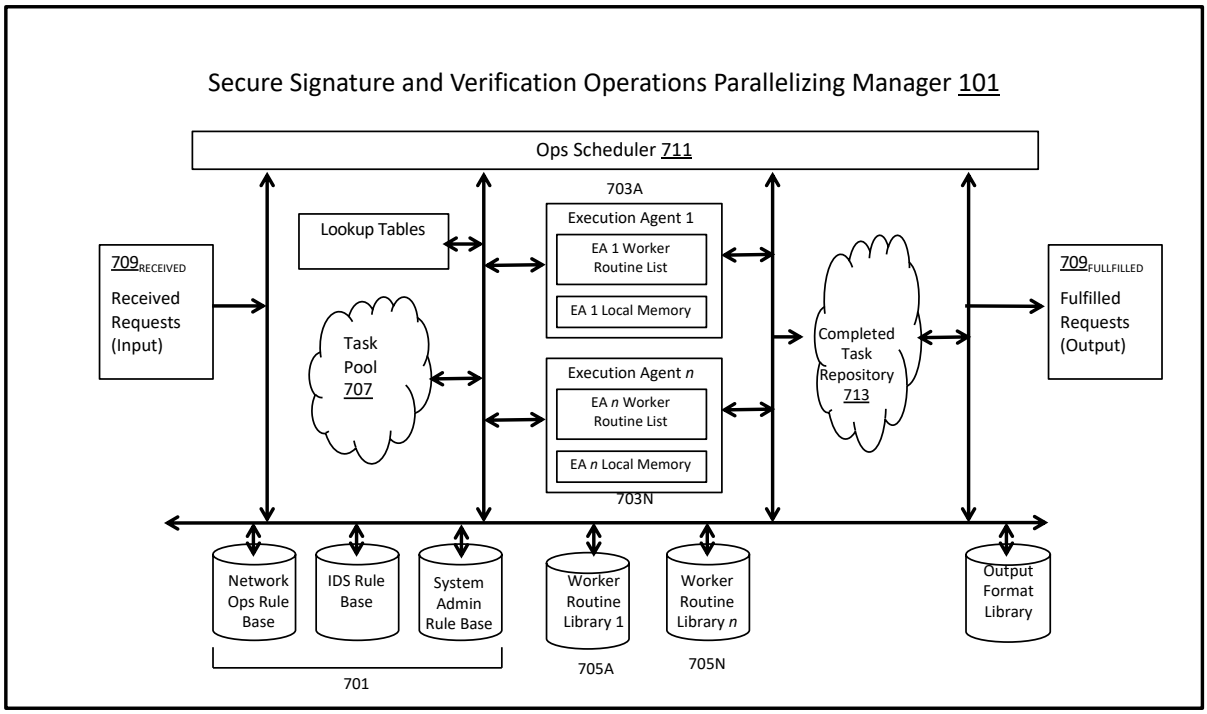


Figure 21 – *tara*BGPsec Operations Parallelizing Manager

The SSVOPM performs its operations based on a customizable rule-based configuration. This configuration mechanism provides flexibility, scalability, and adaptability to future protocol changes. Certain rules establish priorities and define the type and performance characteristics of network crypto operations. For example, when a path verification request for multiple segments is received, the SSVOPM could place that request in the Task Pool as individual segment verifications using a certain curve in order to exercise multiple Execution Agents to maximize the system throughput. Other rules could define the processes for periodic key rollover and emergency key rollover operations. Additionally, configurable Intrusion Detection System (IDS) rules provide a flexible mechanism to harden the system against known and future attacks. In one embodiment, the secure signature and verification operations parallelizing manager uses a reasoning system with forward and backward chaining inference engines.

Execution Agents are configured to embody a finite list of duty-specific worker routines and associated priorities based on the network protocol features. In the case of BGPsec, these worker routines could be defined as signing and verifying a path with ECDSA P-256, or with another signature algorithm. Furthermore, Execution Agents may also contain asynchronous worker routines for lower-level functions such as random number generation, public key

integrity check, and pre-calculations. Since the worker routines are specialized to execute specific functionality, they can be optimized to a much greater extent than generic functions. Depending on the overall protocol requirements and system capabilities, the Execution Agents can be populated at initialization or system run-time with an identical list of worker routines or contain different sets of optimized worker routines.

Any available Execution Agent can take and execute the next priority task in the Task Pool, ensuring system resource maximization as long as network tasks are available. However, due to the sporadic nature of network operations, there could be periods, which could be only a few milliseconds to several seconds at a time, where no network tasks are available in the Task Pool. The novel asynchronous worker routines embodied in the Execution Agents, autonomously use these periods to perform certain pre-calculations, which generate pre-computed data to be used as inputs by higher-level functions in order to maximize the overall system throughput.

In order to be interoperable and compatible with SRxCryptoAPI, *tara*BGPsec conforms to a strict API structure which is as shown in Table 4.

Function	Visibility	Type	Description
taraBGPSec_SystemInit	External	System Utility	Provides library initialization based on configuration parameters. Performs hardware capabilities check, initializes the crypto subsystem, runs self-test and optionally initializes and runs Execution Agents and batch loads keys.
taraBGPSec_SystemRelease	External	System Utility	Releases the crypto subsystem, obliterates confidental data structures, and if present, stops and releases Execution Agents
taraBGPsec_RegisterPrivateKey	External	Key Management	Checks provided private key for validity and securely registers it.
taraBGPsec_unRegisterPrivateKey	External	Key Management	If the private key was registered, unregisters the key and obliterates any memory used to store it.
taraBGPsec_RegisterPublicKey	External	Key Management	Checks the provided public key for validity, pre-computes the public key data, and registers it.
taraBGPsec_unRegisterPublicKey	External	Key Management	If the public key was registered, unregisters the key and releases any memory used to store it.
taraBGPsec_LoadKeys	Internal	Key Management	Provides batch registering of private and/or public keys (file based).
taraBGPsec_genOriginMessage	Internal	Data Management	Constructs an Origin Update Message per BGPsec specification.
taraBGPsec_genMessage	Internal	Data Management	Constructs an Update Message per BGPsec specification
taraBGPsec_Verify	External	Protocol Feature	Verifies the signatures in a received valid path using approved algorithms (currently ECDSA P-256). Supports up to two algorithms. Supports Early Termination and Mitigation of Denial of Service functionality as described in the BGPsec specification.
taraBGPsec_Sign	External	Protocol Feature	Generates signatures using using approved algorithms (currently ECDSA P-256). Supports Robutsness of Secret Random Number in ECDSA as described in the BGPsec specification.
taraBGPsec_initExecAgents	Internal	Thread Management	Initializes and binds execution agents using configuration parameters, system capabilities and configuration rules. Agents can be set as SignAgents, VerifyAgents, or UtilityAgents.
taraBGPsec_runExecAgents	Internal	Thread Management	Dispatches initialized agents. Agents are active only when there is work available.
taraBGPsec_releaseExecAgents	Internal	Thread Management	Releases any running execition agents.

Table 4 – *tara*BGPsec SRx compatible API

Along with System-Level cognitive parallelization optimizations, *tara*BGPsec also utilizes Algorithmic Level Optimizations such as early termination during signature validation. As soon as an invalid segment is detected, signature verification of any remaining segments is aborted.

Since BGP update messages must be cryptographically signed and verified, BGPsec requires real-time line-speed cryptographic operations on BGPsec path announcements. Given the large

size of the global Internet BGP routing table and strict response time necessity for re-convergence following router reboots, this additional requirement causes a *major performance challenge* due to the fact that most currently deployed router control plane hardware does not have the processing power to handle these cryptographic-intensive computations.

As specified in RFC 8608 Section 2.2.1, ECDSA *P-256*, a curve that has been used extensively in critical infrastructure projects, is being used as the Digital Signature Algorithm curve for the BGPsec protocol. The performance efficiency of ECDSA *P-256* is imperative to meet strict Internet routing table convergence requirements. *tara*BGPsec utilizes the highly efficient *P-256* signature generation and verification functionality of the *tara*CRYPT library.

*tara*CRYPT is a self-contained module for the low-level crypto functions that are needed to efficiently and securely implement ECDSA and key management functions. It is designed with standard interfaces so that it can be validated using the industry-accepted NIST Algorithmic Validation process. *tara*CRYPT's ECDSA implementation embodies several novel optimizations at algorithmic, group, and field levels.

There has been considerable research conducted to increase the security and the performance of ECDSA algorithms. Side-channel-attack resilience needs to be inherently built into core functions where applicable in an optimized fashion, rather than included as an after-thought. Performance can be increased via algorithmic or mathematical methods as well as with the facilitation of target platform features with low-level implementation techniques.

At the algorithmic level, the ECDSA sign algorithm requires the generation of (k, k^{-1}), a per-message secret number, and its inverse modulo n. To properly generate (k, k^{-1}) could take around 20 000 cycles or higher in a typical implementation. However, this part of the process does not depend on the contents of the message to be signed. In use-cases where it is important to reduce the latency of signing a message (e.g. cycles or time taken to return a signature after a request to sign a message is issued), (k, k^{-1}) can be pre-computed, per FIPS-186-4 Section 6.3 [28], using one of many available secure methods.

At the group level, the scalar multiplication consumes the bulk of the evaluation time for signature generation and must be implemented carefully to ensure that it does not inadvertently leak information about the secret scalar. For the ECDSA sign operation of prime curves, the point used in the multiplication phase is always the base point G, which is a known value and can be pre-calculated. This method can be extended for use with multiple known points, such as in signature verification for substantial performance gains.

For ECDSA, performance depends directly on the implementation of the multiple-precision arithmetic functions required to support the group-level algorithms. All field operations are performed modulo an associated prime number. Therefore, support for signed integers is not necessary, which substantially simplifies the implementation of the field functions. At the minimum, functions are needed for comparison, addition, subtraction, squaring, multiplication, modular reduction, and modular inversion. Optimizing these functions inherently increases the performance levels.

Other BGPsec performance optimization techniques such as caching already computed AS path segments and their BGPsec signature validation results have been studied [36]. These techniques may be considered in future enhancements of NIST BGPsec implementation.

5.7. Performance Measurements Using the High-Speed *tara*BGPsec and *tara*CRYPT Module

The AS path validation provided by the BGPsec protocol requires each AS in the path to cryptographically attest that it intentionally sent the update in question to the subsequent AS in the path. The attestation is done by means of adding a digital signature to the update. As described in previous sections, Elliptic Curve Digital Signature Algorithm (ECDSA) P-256, a prime curve that has been used extensively in critical infrastructure projects, is being used for AS path signing and verification in the BGPsec protocol. The performance efficiency of ECDSA P-256 is imperative to meet strict Internet routing table convergence requirements. Thus, the viability of BGPsec adoption is dependent on the availability of high-performance implementations of ECDSA P-256.

*tara*CRYPT's ECDSA implementation embodies several novel optimizations at algorithmic, group, and field levels. *tara*CRYPT has been extensively tested on various Intel x86-64 based platforms as well as ARMv8 (AARCH64). To obtain qualitative results, performance analysis and evaluation of the proposed optimizations have been performed on a platform with the 3.5 GHz Intel Xeon E3 1285v4 (Code name Broadwell) CPU with 16 GiB of RAM and CentOS 7. Additional performance tests have also been completed with the 3.8 GHz Intel Xeon E3 1270v6 (code name Kabylake) CPU with 16 GiB memory and CentOS 7. Both systems utilized Enterprise quality 6Gbps Intel SSD hard drives (model DC S3500 Series 240GB). These processor architectures and system components are relevant to the latest routing engine processors offered by leading Edge Router providers.

A set of repeatable performance tests have been run on signature generation and verification operations. The performance tests are available to licensed users of the *tara*CRYPT software. Each operation is run 6 000 times and the median cycles are captured to calculate the effective rate of operations per second. The message size used is nominal 1024 bytes. All tests are run on a single core with both HyperThreading and Turbo turned off. Note that both curve P-256 and P-384 results are included to show the applicability of the optimizations to higher security strength curves. Median cycles, rather than average cycles, are captured since the underlying Operating System processes are not deterministic and may skew performance runs.

Table 5 shows the baseline (prev.) Sign and Verify operation performance (either with provided or generated message Hash), as well as the current performance after applying optimizations (current). The processor is a XEON E3-1285 v4 3.5 GHz (Single Core).

ECDSA Operations (Ops)	prev. P256 (ops / s)	current P256 (ops / s)	Ratio	prev. P384 (ops / s)	current P384 (ops / s)	Ratio
Sign Op with provided Hash	46 192	64 696	1.40	19 434	25 588	1.31
Sign Op including Hash generation	40 076	54 401	1.36	18 163	24 131	1.33
Verify Op with provided Hash	32 895	46 933	1.43	10 955	17 548	1.60
Verify Op including Hash generation	29 521	41 219	1.40	10 707	16 886	1.58

Table 5 – Signing and verification speeds for taraEcCRYPT (single core)

Table 6 compares the original Sign and Verify performance obtained on 3.5GHz Intel Xeon E3 1285v4 vs. the current generation 3.8GHz Intel Xeon E3 1270v6 (Single Core) using the current implementation.

ECDSA Ops	prev. P256 (ops / s)	current P256 (ops / s)	Ratio	prev. P384 (ops / s)	current P384 (ops / s)	Ratio
Sign Op with provided Hash	46 192	73 706	1.60	19 434	28 920	1.49
Sign Op including Hash generation	40 076	63 036	1.57	18 163	27 532	1.51
Verify Op with provided Hash	32 895	59 530	1.81	10 955	20 750	1.89
Verify Op including Hash generation	29 521	49 760	1.69	10 707	19 068	1.78

Table 6 – Signing and verification speeds for taraEcCRYPT (single core)

Key Generation performance on the gateways/servers may also be considered an important characteristic. Table 7 shows the Key Generation Performance for both *P-256* and *P-384* on an Intel XEON E3-1270v6 3.8GHz (Single Core).

ECDSA Ops	P256 (ops / s)	P384 (ops / s)
Key Generation	85 631	32 132

Table 7 – Key generation speed for taraEcCRYPT (single core)

Routing table convergence time for BGP is a commonly used performance metric for measuring BGP router performance. The BGPsec protocol is a security extension for the traditional BGP (BGP-4) protocol. Therefore, the overall BGPsec routing table convergence time will continue to include a traditional BGP processing component that consists of (1) best path selection, (2) peering policies, (3) route filtering, (4) RIB management, etc. Additionally, the overall convergence time will include a BGPsec processing component that consists of (1) parsing BGPsec updates, (2) data assembly for hashing, (3) fetching public keys, (4) verification of signatures, (5) signing updates to peers, etc. The performance of the traditional BGP component varies significantly in the real world BGP routers or route servers (e.g. IXPs or Internet numerous other operational details that are operator configured). In this study, the focus is only on the incremental CPU cost (e.g. processing time) due to the BGPsec component that was described above.

A BGPsec update typically has multiple signatures (one per hop) and an update is Valid only when all its signatures have been successfully verified. So, the BGPsec update processing speed for verification is inversely proportional to the AS path length. The maximum AS path length seen in updates forwarded by a large global ISP based on updates originated from ASes within its customer cone is typically 8 or 9. Based on BGP Update Reports published online weekly, the average AS path length (after compressing AS prepends) is about 4 (3.77) in the Internet [39].

Figure 22 shows the *tara*BGPsec Path Verification operation performance with P-256 running in parallel on a Xeon E3 1285v4 3.5 GHz as a function of #cores.

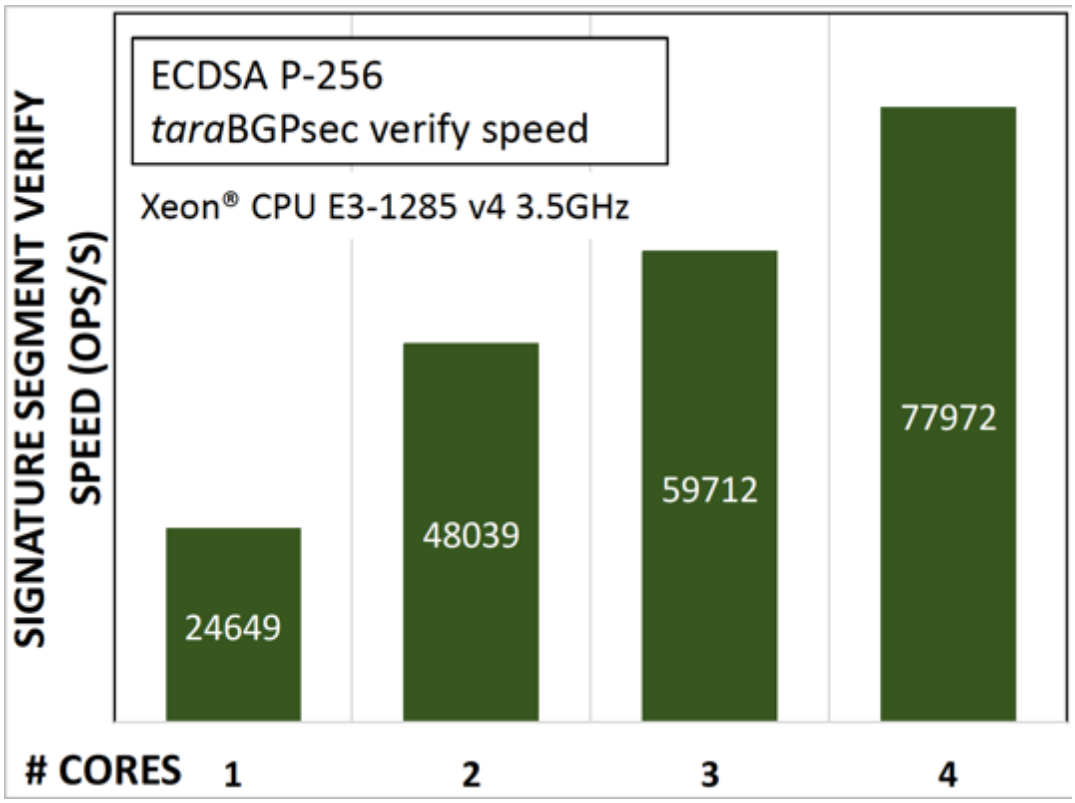


Figure 22 – taraBGPsec Segment signature verification performance over CPU cores

As the results indicate, the sustained multi-core performance of *tara*BGPsec will minimize the additional time required for BGPsec processing and thus maintain an acceptable BGP routing table convergence time.

6. Traffic Generation with BGPsec-IO

BGPsec-IO (BIO) is a BGPsec traffic generator that allows the generation of multi-hop fully signed BGPsec UPDATE messages as specified in RFC 8205. As illustrated in Figure 23, the BGPsec Path Attribute can either be embedded in a BGPsec UPDATE message to be sent to a peering BGPsec capable router, used directly to perform validation calls to test configured SRxCryptoAPI crypto module, or stored in a file for later retrieval. BIO uses its own internal crypto engine based on OpenSSL using the signing algorithm specified in RFC 8608.

Hooks are already in place to allow BIO to facilitate the integration of the SRxCryptoAPI for update signing. These hooks are currently not completely implemented, so BIO uses its own implementation for UPDATE signing.

To eliminate the computation of multi-hop fully signed BGPsec UPDATEs, BIO allows to pre-generate traffic and store it as a binary stream for later retrieval or to generate BGP and BGPsec UPDATE traffic "on the fly". This is important when it comes to performance testing of BGPsec router implementations. Using pre-generated BGPsec UPDATEs removed the crypto and marshaling operations. Therefore, the only slowdown is the I/O speed of loading the data from the drive to the memory which can be drastically reduced by using SSD drives or RAM drives.

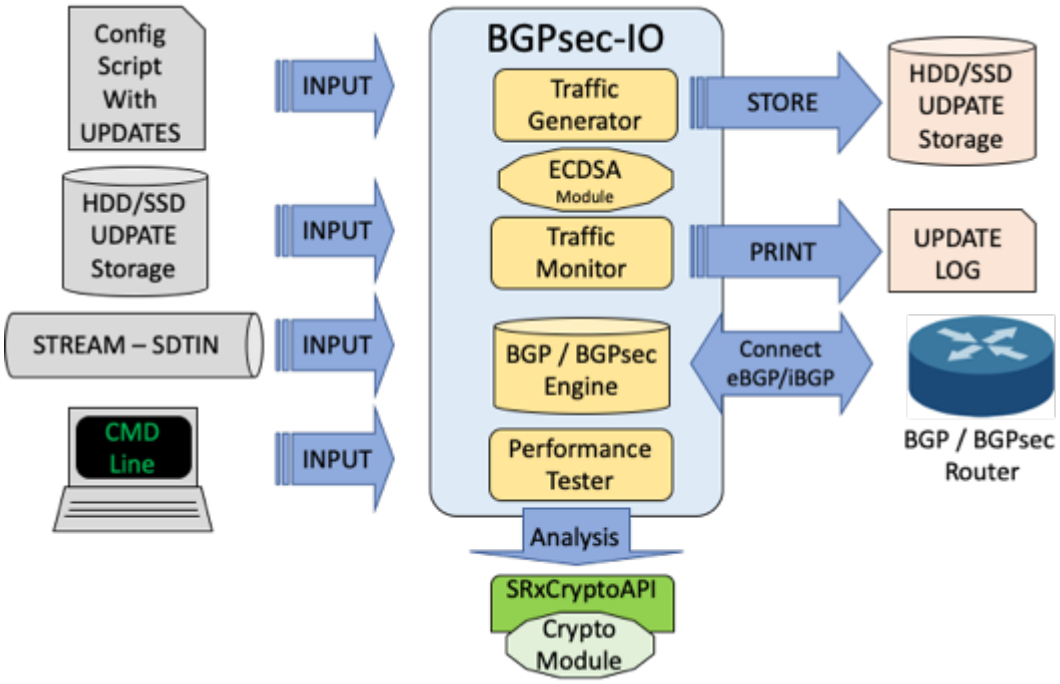


Figure 23 – BGPsec-IO functional design

Besides generating fully signed BGPsec UPDATEs, BIO is capable of using both iBGP and eBGP configurations. This is important because both types of sessions are handled differently within the BGPsec protocol. iBGP sessions are BGP sessions within an AS whereas eBGP sessions are BGP sessions between two different ASes.

In case of missing keys which will lead to unsuccessful signed BGPsec paths, BIO allows to specify different fallback solutions:

(1) Generate a regular BGP-4 UPDATE,
(2) Generate BGPsec UPDATE with "fake" signature using "fake" SKIs, or
(3) Ignore the UPDATE generation completely.

The first solution is the same as the operation a BGPsec speaker does when talking to a non BGPsec speaker RFC 8205.

The second solution allows testing the correctness of BGPsec implementations by allowing the operator to pre-script the signature and SKI.

To allow scripting interesting scenarios of BGP traffic where BGP UPDATEs and BGPsec UPDATEs are mixed together, BIO allows specifying updates to be sent as BGP-4 UPDATEs only as specified in RFC 4271.

Furthermore, BIO allows producing Wireshark-like outputs of sent and received BGP messages. This includes OPEN, NOTIFICATION, KEEP-ALIVE, and UPDATE messages.

6.1. BGPsec-IO and Session Handling

The original intent for BGPsec-IO was to develop a BGPsec capable traffic generator that could be used to test QuaggaSRx as well as the simultaneously developed BIRD [13] implementation. In addition, the plan was to have a single test engine that not only could generate multi-hop signed BGPsec UPDATEs, but also take the role of multiple BGPsec traffic generators, each assuming the role of a different AS. For this reason, the BIO configuration allows scripting multiple sessions. The implementation to read multiple session configurations, as well as the internals, is already put in place, though mostly not activated using define statements within the codebase. The reason for that is that these portions in the code are barely tested.

As shown in Figure 24, BIO can be operated in a different way. Instead of operating one single BIO instance per platform, multiple instances can be operated on one single platform as well. The only requirement to configure a distinct routable IP address for each BIO instance.

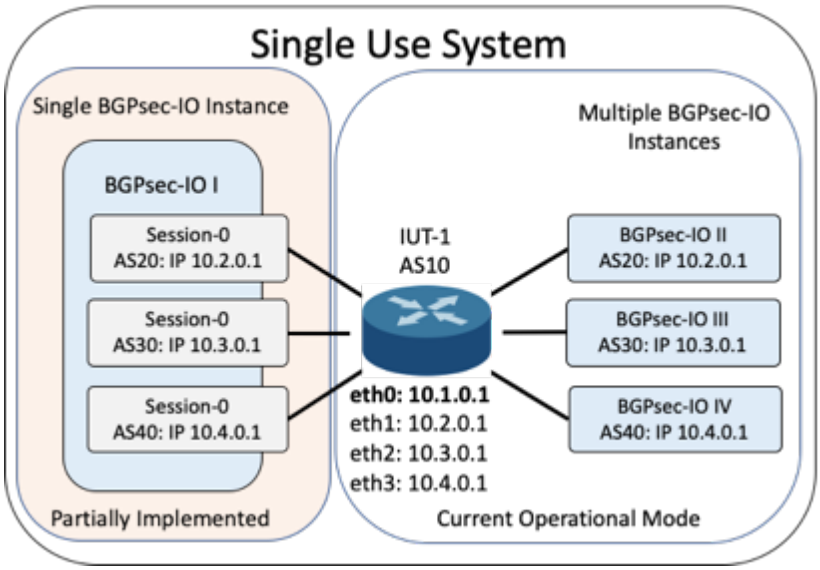


Figure 24 – Single use system

This can be achieved by either adding network interface cards or the simpler form of configuring interface aliases. Each physical interface can have multiple aliases bound to it with each alias providing its own IP address. BIO does bind itself to the interface that serves as the IP address configured in the session configuration (see Appendix A.3.1). Binding to the interface is possible because BIO does not listen on port 179, specified as the dedicated port for BGP. BGP sessions with BIO will be initiated only by BIO. To reduce unnecessary BGP

traffic, the router Implementation Under Test (IUT) should be configured to operate in "passive" mode. In this mode, the IUT will not attempt to send an OPEN message to BIO. Though this configuration for the UIT is not required.

6.2. Generation and Handling of BGP / BGPsec UPDATEs

BGPsec-IO is built to generate fully signed multi-hop BGPsec UPDATEs that can be sent to a BGPsec capable router as well as being used to test and evaluate the performance of BGPsec cryptographic implementations. One major requirement is to be able to generate updates simply from a string that contains the prefix and as-path. As shown in Figure 23 BIO accepts four different forms of update specification:

(1) Via command line passing updates during the program call,

(2) Via the configuration file – session section and global section,

(3) From the standard input, when "piped" into the program, and

(4) From a file containing pre-generated in protocol encoded binary format – as UPDATEs and/or PATH_ATTRIBUTEs

This section will further elaborate on how updates are scripted, generated, and what additional features are possible.

6.2.1. Scripting of UPDATEs

Independently of the source of scripted UPDATEs, each update must follow the syntax as shown in Listing 12.

```
"<prefix>[,[B4]?[ <asn>[p<repetition>]]*[ ]*[I|V|N]?]"
```

Listing 12 – BGPsec-IO Scripted UPDATE format

The most important information to remember is that BIO does insert its own AS into the UPDATE. The only exception here is if BIO originates the UPDATE and sends it to an iBGP peer. In this case, BIO does not generate the AS_PATH attribute as specified in RFC 8205 Section 4.1 or RFC 4271 Section 9.1.2.2 respectively.

The minimum information the scripted UPDATE must provide is the prefix itself. This is also done if the prefix is considered being originated by the AS that BGPsec-IO is configured as. The prefix and the path information must be separated by a comma.

By default, BIO attempts to generate a BGPsec encoded UPDATE. As shown in Listing 13, the parameter *B4* directs BIO to not generate a BGPsec update but generate a BGP-4 update instead. This allows testing a mixture of BGP-4 and BGPsec UPDATEs which most likely will be seen in the early stages of BGPsec deployment. It also allows generating BGP-4 only traffic for scenarios where BGPsec is not important but the focus is on testing of signaling BGP-OV validation results as specified in RFC 8097. This setting was extensively used for the National Cybersecurity Center of Excellence (NCCoE) project "Protecting the Integrity of Internet

Routing" published in NIST Special Publication 1800-14A [29]. The absence of the *B4* attribute directs BIO to generate BGPsec UPDATEs [8].

```
  "192.0.2.0/24, B4",
  "192.0.2.0/24, B4 65535 65536"
```

Listing 13 – Scripting BGP-4 UPDATEs

In BGPsec, UPDATEs with AS number concatenations are scripted differently than in BGP-4. As shown in Listing 14, to extend the path length, whereas BGPsec uses a prefix counter called pCount to mimic the number of repetitions of the UPDATE.

```
BGP4:   65536 65536 65536 65536 65536 65536 65537
BGPsec: 65536 pCount=6 65537
```

Listing 14 – Simplified UPDATE concatenation BGP-4 vs BGPsec

BIO does accept both notations (repetition of the AS number or using the pCount value). The notation in BIO is shown in Listing 15.

```
Regular Expression: [ <asn>[p<repetition>]]*
Without pCount....: 65536 65536 65536 65536 65536 65536 65537
With pCount.......: 65536p6 65537
Using Both........: 65536p4 65536 65536 65537
```

Listing 15 – UPDATE concatenation in BIO

Regardless of how it is scripted, BIO will use the AS-number repetition for BGP-4 UPDATEs, and will generate the proper pCount value for BGPsec UPDATEs.

As mentioned in the introduction of this section, BIO implements RFC 8097. This feature must be enabled on a "per UPDATE" basis. BIO does not perform BGP-OV, but it allows to specify the validation result within the scripted update.

As shown in Listing 16 the validation result is indicated at the end of the UPDATE's path using I=invalid, V=valid, or N=not found. Only a single result is permitted per update.

```
Regular Expression: [I|V|N]?
Usage: "192.0.2.0/24, 65536 I"
```

Listing 16 – Signaling route origin validation result.

Updates can be scripted in multiple areas:

- the configuration file,
- as a piped list using STDIN, or
- or as a command-line parameter.

[8] In case BIO is not able to generate a fully signed BGPsec UPDATE due to a missing or faulty key, BIO will resort to the fallback method as specified in the configuration.

If scripted within the configuration file, it can be scripted in two separate sections: the router configuration or the global configuration. At this time, only a single router can be configured, therefore it does not really make a difference where in the file UPDATEs are scripted. It is envisioned that BIO can create multiple BGP / BGPsec sessions to the same router but as different peers. Once this feature is activated then the session specified UPDATEs are only generated for that specific section, but the global section is applied to all instances.

In the case of scripting the UPDATEs within the configuration file, UPDATEs are scripted inside an array as shown in Listing 17.

```
Update = (
  "192.0.2.0/24",
  "192.0.3.0/24, 65536 65536 65537",
  "192.0.4.0/24, 65536p2 65537"
);
```

Listing 17 – UPDATEs Scripted within the configuration.

If UPDATEs are passed using the "piping" function (for instance processing an MRT file and passing the UPDATEs to the runtime environment) each UPDATE must be provided in its own line.

The final form of scripting UPDATEs in BIO is by using either of the command line options:

-u "<prefix, path>" or *"--update <prefix, path>"*

A comprehensive list of command line parameters for BIO is listed in Appendix A.3.2.

6.2.2. Update Generation and Processing Order

As mentioned earlier, BGPsec-IO has four different methods of retrieving UPDATE data:

(1) by command line,

(2) by scripting,

(3) by "piping" and

(4) by loading from file.

How to pre-generate UPDATEs is explained in detail in Section 6.4.2. It is important to fully understand the order in which BIO generates UPDATEs, especially if they are used to generate BGP and BGPsec traffic. For this, the guarantee of a deterministic behavior of UPDATE processing is essential. The ability to run diagnostics and being able to debug through a deterministic message flow is highly important. Experiments might be carefully scripted around the order of received UPDATEs. The configuration and operations between BIO and BGP/BGPsec speakers are explained in detail in Section 6.1 of this document.

The following list shows the order in which UPDATEs are generated and processed.

(1) Command line (first session only)

(2) Scripted in configuration file (session specific config)

(3) Scripted in configuration file (global config)

(4) Piped in via STDIO (first session only)

(5) Loaded from binary file (first session only)

In case a binary file is used to load BGPsec UPDATEs, it is to note that only stored BGPsec path attributes are used in the SCA C-based API (CAPI) mode. Stored BGPsec UPDATEs will not be parsed to retrieve the BGPsec Path Attribute.

The internal operations of the UPDATE "Cache" within BIO as shown in Figure 25 is built like a First-In-First-Out (FIFO) queue. In fact, to the caller of the UPDATE queue, it appears to be just that. The internal queue implementation is much more complicated though, especially as we did not want to require loading all UPDATEs into the memory. This could be very memory intensive if BIO is meant to send complete BGP tables to a router. In configurations where multiple BIO instances run on the same physical test system as the peering router, it is important to keep the memory consumption of BIO moderate to low. The less memory BIO consumes, the more memory is available for the router implementation under test (IUT). In cases where BIO operates in a "live" mode, that means where an external traffic generator generates UPDATEs occasionally without an end in sight, waiting until the UPDATE is generated prior to sending them to the IUT would be unreasonable.

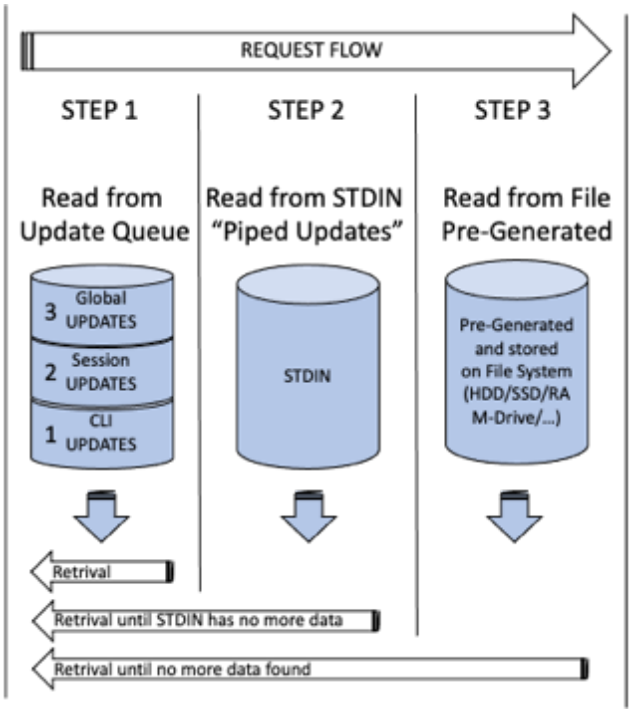


Figure 25 – BIO UPDATE stack inner workings.

As shown in Figure 25, BIO implements an internal data retrieval process. From the outside, this process acts as a FIFO stack API. The data structure is loaded in the following manner: First, BIO loads all scripted updates provided via the command line parameter. These are the first ones to be retrieved. This allows to easily inject updates in front of scripted scenarios. After that, all updates scripted in the session section are configured, followed by all updates in the global section. At the time of this document, support for multiple sessions is not fully implemented and therefore all existing code for multiple sections is disabled using define statements.

The last section contains global scripted updates. These are the updates that are loaded regardless of which session is configured. Only once all these updates are consumed, BIO starts polling the standard input. This is when so-called "piped" updates are written. This can be accomplished using the provided tool "***bio-updates.sh***" which is explained in more detail in Appendix A.3.3. This tool allows generating many updates, including AS path and writes them one by one to standard out (STDOUT), each in its own line. Using the pipe "|" symbol allows redirecting the STDOUT of this tool into the standard in (STDIN) of the BGPsec-IO program "***bgpsecio***". Then, during Step 2 of the flow shown in Figure 25, the data will be retrieved from the STDIN once no further scripted updates are available. As soon as all data is read from STDIN, pre-generated UPDATEs and BGPsec PATH attributes are loaded and processed. The last step will not be performed in GEN mode and only partially in CAPI mode. The modes will be explained in detail in Section 6.4

6.3. The Internal Cryptographic Engine

The internal Cryptographic Engine within BIO is only used for signature creation and can be configured with four different settings:

- **CAPI:** This setting uses the SRxCryptoAPI (CAPI) mode. This mode is not fully operational for signature generation but is used extensively for signature verification.
- **BIO:** This mode specifies the BGPsec-IO internal crypto implementation. This implementation was developed autonomously from the BGPsecOpenSSL implementation by a different developer, without any code sharing. This was important to verify that the crypto algorithm can be developed independently and produce the same outcome.
- **BIO-K1 / BIO-K2:** Same as BIO with the exception that BIO allows selecting between two constant values for 'k'.

Listing 18 displays how the signature generation is configured.

```
signature_generation = "BIO-K1";
```

Listing 18 – Configure BIO signature configuration

Even though the BIO's configuration engine allows to configure CAPI mode for signature generation, BIO currently can only use its own BGPsec-IO's embedded cryptographic implementation. Though the core functionality exists within BIO's implementation that allows using SCA for exact that operation using the CAPI setting for signature creation, this feature was put on hold to make room for parts of the implementation deemed more critical. Currently, using this functionality produces the message shown in Listing 19.

```
Function "CAPI_createSignature" is not prime time ready!ERROR:
Registering public key:
ERROR: 0x10000
       8 4 2 1 8 4 2 1
       0 0 0 0 0 0 0 1
                     |
                     +--API_STATUS_ERR_NO_DATA
```

Listing 19 – CAPI signature creation error BIO

One of the more pressing required features was finding a solution to generate deterministic signatures using ECDSA. In general, ECDSA is non-deterministic. This characteristic of ECDSA makes it very difficult for debugging or analyzing an IUT in case it behaves unexpectedly. RFC 6979 Section A2.5 [30] provides settings for ECDSA that address exactly this issue. It provides two specific values for "k" as shown in Table 8 that are used for the algorithm to be deterministic.

Mode	Description / Value
BIO-K1	Signatures With SHA-256, message = 'sample'
	0xA6, 0xE3, 0xC5, 0x7D, 0xD0, 0x1A, 0xBE, 0x90, 0x08, 0x65, 0x38, 0x39, 0x83, 0x55, 0xDD, 0x4C, 0x3B, 0x17, 0xAA, 0x87, 0x33, 0x82, 0xB0, 0xF2, 0x4D, 0x61, 0x29, 0x49, 0x3D, 0x8A, 0xAD, 0x60
BIO-K2	Signatures With SHA-256, message = test
	0xD1, 0x6B, 0x6A, 0xE8, 0x27, 0xF1, 0x71, 0x75, 0xE0, 0x40, 0x87, 0x1A, 0x1C, 0x7E, 0xC3, 0x50, 0x01, 0x92, 0xC4, 0xC9, 0x26, 0x77, 0x33, 0x6E, 0xC2, 0x53, 0x7A, 0xCA, 0xEE, 0x00, 0x08, 0xE0

Table 8 – "k" value from RFC 6979

This very helpful value allows the generation of reproducible and therefore deterministic signatures for testing and debugging cryptographic implementations on the client-side (BGPsec router) or cryptographic implementation for SCA. Even though this configuration is very helpful, it must not be used in production environments because it will allow to identify the private key and therefore render the security useless!

An additional important feature is the capability to specify the behavior in case BIO itself does not have a valid private key. This situation can be intentional but, as the nature of experimentation often shows, it is sometimes unintentional. Table 9 shows the different fallback methods BIO provides in the event of an unsuccessful BGPsec signature creation.

Fallback Mode	Description
BGP4	Skip BGPsec generation and generate a regular BGP-4 UPDATE
DROP	Skip all UPDATE generation and drop this update
FAKE	Generate an UPDATE with a predefined SKI as well as a predefined signature that can be traced in TCP dumps

Table 9 – BIO signature generation fallback mode "*null_signature_mode*"

The first solution "BGP4" was implemented at a time when update scripting did not include the specification "B4" to generate BGP-4 UPDATEs. Until the introduction of the parameter "B4" during update scripting, the only possible form of generating BGP-4 updates was to use an AS in the path that did not provide a private key. There the BGPsec UPDATE generation failed and the fallback method was to generate regular BGP UPDATEs. This is unsustainable because preparing the generation of BGPsec UPDATEs up to the moment where the fallback gets executed, the costs of generating the BGPsec Path Attribute is very high in comparison with the generation of a simple AS_PATH attribute. Experimentation of 5 %, 10 %, or 15 % BGPsec deployment rate still would mean at least 85 % of UPDATEs generated would generate large portions of the BGPsec Path Attribute. For experiments that facilitate "piping" and inline BGPsec UPDATE generation, this process adds an avoidable slowdown in processing. Still, this setting provides a meaningful solution for debugging. Problems with the dataset can easily be found by searching the TCP stream for the existence of BGP-4 UPDATEs (assuming no updates are scripted using the B4 flag).

The second option is to "DROP" the update completely.

The third option specifies generating a "FAKE" signature. This is an extremely useful feature, especially for analyzing the correctness of BGPsec implementations. Both, the SKI as well as the Signature are preconfigured within the configuration script as shown in Listing 20.

```
null_signature_mode = "FAKE";
fake_signature      = "1BADBEEFDEADFEED" "2BADBEEFDEADFEED"
                      "3BADBEEFDEADFEED" "4BADBEEFDEADFEED"
                      "5BADBEEFDEADFEED" "6BADBEEFDEADFEED"
                      "7BADBEEFDEADFEED" "8BADBEEFDEADFEED"
                      "ABADBEEFFACE";
fake_ski            = "0102030405060708" "090A0B0C0D0E0F10"
                      "11121314";
```

Listing 20 – FAKE signature creation

It is very important that the signature must be within the range of 69 to 71 bytes scripted in hexadecimal values with 2 values per byte and the value for the SKI must be exact 20 bytes scripted in 40 hexadecimal values.

This allows specifying signatures that must fail on the receiver's side – and if they do not that indicates that the IUT does not use a correct signature verification algorithm. Furthermore, being able to freely specify both the SKI as well as the signature itself prevents a hard-coded

detection of intentionally scripted invalid updates. It also allows to easily find the UPDATE in TCP traces.

6.4. BGPsec-IO Operational Modes

BGPsec-IO operates in three different modes. As shown in Figure 26, the first mode is CAPI, used to test cryptographic modules that perform all BGPsec operations and are embedded within the SRxCryptoAPI. The second mode is GEN, which generates BGP / BGPsec related UPDATE data as found on the wire. The third and last mode, BGP, is the mode in which BIO takes the role of a BGP router.

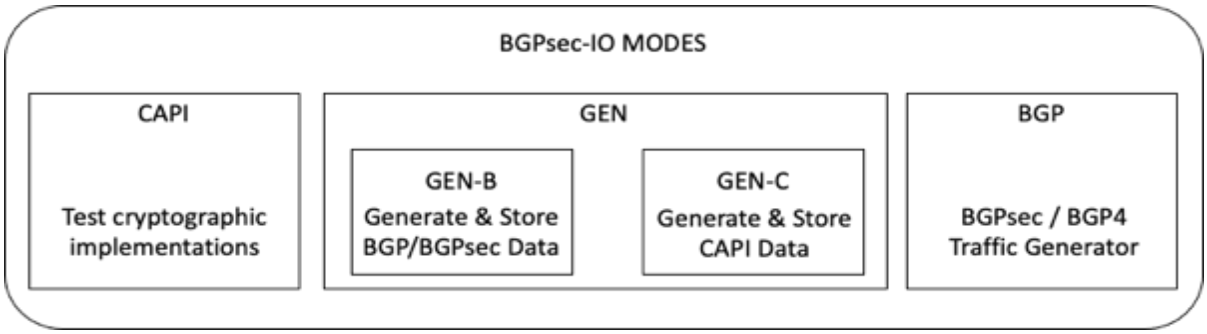


Figure 26 – BGPsec-IO modes

The mode is normally selected within the configuration but can be modified using the command line parameter -m followed by the mode itself. The following subsections explain each mode in more detail.

6.4.1. CAPI Mode

The CAPI mode is the SRxCryptoAPI performance and correctness tester mode. It can be used for functionality tests of the crypto module by providing valid and invalid BGPsec UPDATEs. Additionally, it allows measuring the performance of the API by measuring the time that elapses from function call to result.

As shown in Listing 21, BIO generates output for valid and invalid updates. This particular run was performed on a Proxmox [35] virtual machine with 12 MiB of RAM and 12 cores using the BGPsecOpenSSL implementation.

```
Statistics Invalid:
====================
  0 updates (0 segments) in 0 ns processed
  - average time per update:  0 ns
  - average time per segment: 0 ns
  - average number of segments per update: 0.00

Statistics Valid:
====================
  100001 updates (500001 segments) in 103893353503 ns processed
  - average time per update:  1038923 ns
  - average time per segment: 207786 ns
  - average number of segments per update: 4.00
  - segments per second: 4812
```

Listing 21 – CAPI statistics output

When measuring performance, BIO does not include the time for UPDATE generation, as BIO only measures the time from the start of the validation call to the time the validation call returns.

As shown in Figure 27, BGPsec-IO first loads all the scripted updates. Then it generates the BGPsec_PATH and prefix and hands the information over to SCA. The only time that will be measured is the time that elapses from making the function call (*t1*) to the return of the program pointer (*t2*). This call is a direct mapping into the cryptographic implementation. In the case of BGPsecOpenSSL, portions of the SCA are used, such as key loading and Path processing. Other implementations may or may not use any of these helper functions provided by SCA. The recorded time is $t = t2 - t1$.

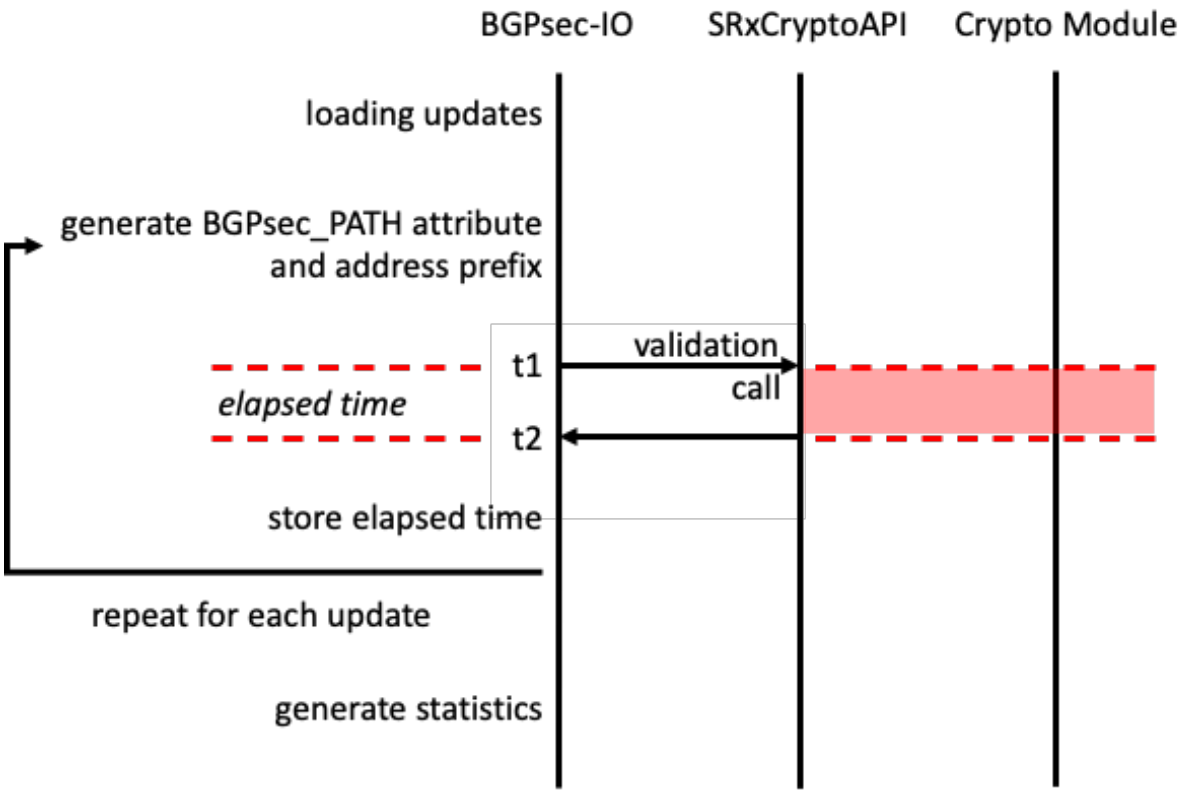


Figure 27 – CAPI performance measurement

In the case of "piped-in" updates or updates loaded from files, BIO does not pre-load these updates. They will be loaded one by one in between each validation call. The condition "repeat for each update" is in fact the call to the update stack which is explained in detail in Section 6.2.2.

At this point, BIO does not provide performance tests of SCA signing operations. The implementation for this mode is incomplete and not fully tested.

6.4.2. GEN-[B|C] Mode

The GEN mode was designed originally to be able to generate reference data that can be archived and reused. At that time, BIO did not provide the BIO-K1 or BIO-K2 modes for

generating signatures and therefore the generation of BGPsec data was non-deterministic. The simplest solution at that time was to generate the BGPsec data and store it in its binary format as specified in the respective RFCs. For this, BIO provided a single GEN mode that stored the BGPsec_PATH attribute as specified in RFC 8205. This way, BIO could immediately use this data and perform the validation call using the SRxCryptoAPI. More importantly, the experimentation immediately became deterministic, as it became possible to generate reference data that could be used to test crypto implementations. Experiments can be repeated with the exact same input data, which is very important, especially when unexpected results are seen which need to be reproducible. This would be almost impossible using a non-deterministic behavior.

Once the BGP mode (see Section 6.4.3) was implemented, it became clear that for testing a BGPsec peer, it would be useful to have already the complete BGPsec UPDATE available and therefore eliminate the processing time to generate the rest of the UPDATE itself. This becomes very important if BIO is used to test implementation performance improvements in the peer's implementation, especially if the bottleneck would be BIO's UPDATE signing (remember that BIO not only signs the same update once, a five-hop update needs to be signed five times; longer paths even more often).

At this point, the mode GEN was split into GEN-C that only generated BGPsec-PATH information for CAPI usage, and GEN-B which generates the complete BGPsec UPDATE. With GEN-B, BIO can immediately copy the data from the file input stream into the TCP output stream. This allows to preconfigure experimentations and store the updates in a file for replay. This allows running BIO on platforms that are not very powerful when it comes to cryptographic operations. Here the bottleneck is most likely in the IO speeds, which can be fixed using SSD or RAM drives.

In addition to each data entry, BIO stores the public keys needed to verify the signatures. Now for replaying data, BIO can use both data types *C* and *B* when operating the BGP mode. If the binary data only contains the BGPsec_PATH attribute including the Prefix information, BIO can generate the remaining missing part to complete the BGPsec UPDATE message. When operating in CAPI mode, BIO only uses GEN-C generated data packages. Theoretically, it would be possible to just extract the CAP relevant data from the BGPsec UPDATE, but it was not implemented.

6.4.3. BGP Mode

The BGP mode allows BGPsec-IO to function as a BGP / BGPsec speaker. In BGP mode, BIO provides not just BGPsec functionality: It can be used as a traffic generator as well as a traffic collector. It is possible to test the functionality of BGP-OV signaling as specified in RFC 8097 as well as testing the usage of extended message support for BGP as specified in RFC 8654. In fact, BIO is listed as one of the reference implementations during the draft development.

Performance and compliance testing are not trivial, and BIO attempts to serve as a tool that makes this operation easier. For this, one important feature of BIO is to ease the way in which experimentation can be scripted and configured. For instance, enabling RFC 8097 we added the validation state to the scripted path (see Section 6.2.1).

As mentioned above, BIO allows scripting updates that can mimic all different kinds of scenarios. Valid and invalid BGPsec updates, traffic can be a mixture of BGPsec and BGP-4,

updates can exceed the 4 KiB boundary of RFC 4271 by negotiating RFC 8654's extended message support. BIO also differentiates between iBGP and eBGP connections and generates the traffic accordingly. This functionality was heavily used during the NIST's NCCoE "Protecting the Integrity of Internet Routing: Border Gateway Protocol (BGP) Route Origin Validation" [29] project.

One aspect feature that was not yet mentioned is BGP-4 prefix packing: This is a feature that allows reducing the number of UPDATEs being generated in BGP-4 by bundling prefixes that share all attributes and send them all as a list of prefixes within the same UPDATE message. This not only reduces the message count and therefore the traffic load, but it also reduces the policy processing that is applied to community string values, etc. BIO also allows performing prefix packing for BGP-4 if configured. Even though this feature is not something that has an impact on BGPsec, for BGP-4 though it is very essential, and because BIO not only can be used as a BGPsec traffic generator, it also can be used as a BGP-4 traffic generator. The only limitation is that the BGP-4 updates that share the same prefix MUST be scripted in a row. The reason for that is that BIO does not contain a RIB. The updates are stored in a FIFO stack-like fashion and for BGP-4 only the next UPDATE in the list is examined for the possibility of packing.

6.5. Logging BGP / BGPsec Traffic

When developing protocol extensions such as BGPsec or community attributes for BGP, being capable of analyzing the "data on the wire" is an indispensable tool. One can use TCP dump or tools such as Wireshark™ to visualize the data and make it human readable but most of the time these tools are either not available or do not provide the proper plug-ins. This is especially true for BGPsec. Even if these tools are available, their usage often complicates the experiment setup. For this reason, BIO got gradually expanded to allow printing the data sent to the peers. The BIO implementation provides a "Printer" framework, where the printer scans the BGP UPDATE and displays its content in human-readable form. Portions that are not digested properly are displayed using their hexadecimal values instead. Because not the complete set of data is interesting to monitor, additional filters as shown in Table 10 were needed that allow to specify what data to be print and on which port, the incoming, the outgoing, or both.

Sending – Traffic Generation	Receiving – Traffic Monitoring
`printOnSend = true | false`	`printOnReceive = true | false`

`printOnSend = {`
`  open  = true | false`
`  update  = true | false`
`  keepalive  = true | false`
`  notification = true | false`
`  unknown  = true | false`
`}`
	`printOnReceive = {`
	`  open  = true | false`
	`  update  = true | false`
	`  keepalive  = true | false`
	`  notification = true | false`
	`  unknown  = true | false`
	`}`

Table 10 – BIO – Traffic printer

This additional filter allows specifying either all or a subset of the BGP message types. Forward-thinking this implementation also allows filtering for data types that are not yet known. These types will be printed most likely as hexadecimal values.

```
UPDATE Message
  +--marker: FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
  +--length: 256
  +--type: 2 (UPDATE)
  +--Unfeasible routes length: 0
  +--total path attr length: 233
     +--ORIGIN: INCOMPLETE (4 bytes)
     |  +--Flags: 0x40 (Well-Known, Transitive, Complete)
     |  +--Type Code: ORIGIN (1)
     |  +--Length: 1 byte
     |  +--Origin: INCOMPLETE (1)
     +--MULTI_EXIT_DISC (7 bytes)
     |  +--Flags: 0x80 (Optional, Non-transitive, Complete)
     |  +--Type Code: MULTI_EXIT_DISC (4)
     |  +--Length: 4 bytes
     |  +--data: 00 00 00 00
     +--MP_REACH_NLRI (16 bytes)
     |  +--Flags: 0x80 (Optional, Non-transitive, Complete)
     |  +--Type Code: MP_REACH_NLRI (14)
     |  +--Length: 13 bytes
     |  +--Address family: IPv4 (1)
     |  +--Subsequent address family identifier: Unicast (1)
     |  +--Next hop network address: (4 bytes)
     |  |  +--Next hop: 10.80.0.1
     |  +--Subnetwork points of attachment: 0
     |  +--Network layer reachability information: (4 bytes)
     |     +--172.16.7.0/24
     |     +--MP Reach NLRI prefix length: 24
     |     +--MP Reach NLRI IPv4 prefix: 172.16.7.0
     +--BGPSEC Path Attribute (206 bytes)
        +--Flags: 0x90 (Optional, Non-transitive, Complete, Extended Length)
        +--Type Code: BGPSEC Path Attribute (33)
        +--Length: 202 bytes
        +--Secure Path (14 bytes)
        |  +--Length: 14 bytes
        |  +--Secure Path Segment: (6 bytes)
        |  |  +--pCount: 1
        |  |  +--Flags: 0
        |  |  +--AS number: 80 (0.80)
        |  +--Secure Path Segment: (6 bytes)
        |     +--pCount: 1
        |     +--Flags: 0
        |     +--AS number: 7 (0.7)
        +--Signature Block (188 bytes)
           +--Length: 188 bytes
           +--Algo ID: 1
           +--Signature Segment: (92 bytes)
           |  +--SKI: 18494DAA1B2DFD80636AE943D9DC9FF42C1AF9D9
           |  +--Length: 70 bytes
           |  +--Signature: 30 44 02 20 6A 99 26 70   E6 56 6E 42 5A 12 5B BC
           |                C2 C6 E8 5B E2 AE D7 A0   0B 94 AF 42 57 F1 7B 38
           |                85 3B 48 FA 02 20 65 D8   A1 68 63 35 50 F3 88 51
           |                A9 23 76 CB 18 86 E5 9E   54 65 1B A6 93 87 FE 85
           |                D7 9C 33 BB 9E 9E
           +--Signature Segment: (93 bytes)
              +--SKI: 8BE8CA6579F8274AF28B7C8CF91AB8943AA8A260
              +--Length: 71 bytes
              +--Signature: 30 45 02 20 33 85 12 07   F9 B2 94 7D 42 13 6D F2
                            92 3D 56 8A D5 FD 03 78   0E E6 30 46 B8 69 94 D8
                            7B 6E 3C E3 02 21 00 D1   AD 14 B4 E7 2C 9E 22 E3
                            8E 5B 49 CB 97 E0 E6 A6   25 9F 26 B7 84 09 F7 04
                            A0 C4 3C 72 84 4F 96
```

Listing 22 – BGP Printer output in long format

As shown in Listing 22 the output of the traffic printer is oriented on the output provided by widely used tools such as Wireshark. The default output for BGP/BGPsec UPDATE messages is rather large and should only be used for debugging purposes. BIO not only provides the large, detailed output, but it also provides a denser version that can be activated using the configuration parameter *printSimple.* By default, this parameter is set to false. Once set to true, the output of the "simple" mode will be much denser.

As shown in Listing 23 each message appears in a single line. The prefixes ">" and "<" indicate if the message was received ("<") or sent (">"). The above output was generated with the setting *printOnSend=true;* and *printOnReceive=true;*.

```
> OPEN
< OPEN
> KEEPALIVE
< KEEPALIVE
< KEEPALIVE
> +PFX: 10.80.0.0/16, BSP: 80 70
> ASP: 80 60, +PFX: 10.80.4.0/24
> +PFX: 172.16.7.0/24, BSP: 80 7
> +PFX: 172.16.7.0/24, BSP: 80 2
< +PFX: 10.50.0.0/16, ECA: 43C8000000000000, BSP: 50
```

Listing 23 – BGP Printer output in short format

This allows to easily generate scripts that parse over the output for post-processing. Because a BGP UPDATE does not require a specific order in which its attributes are generated, BIO does provide indicators on what type of attribute is printed.

A post-processor can filter the data according to the given label. Table 11 focuses only on BGP / BGPsec UPDATE messages and displays the labels used for each path attribute. Using this functionality allows to easily attach some monitor to the output of BIO. As shown, BIO does not perform UPDATE validation of received UPDATEs but adding this functionality would not be very difficult.

Label	Description
+PFX	Announcement of the IP prefix that follows:
-PFX	Prefix Withdrawal
BGP	BGPsec_PATH attribute with the PATH as AS list (BGPsec UPDATE)
ASP	The AS_PATH attribute as AS list (BGP-4 UPDATE)
ECA	Extended Community String. The value is written as a hexadecimal String.

Table 11 – BIO – Traffic printer simple codes for BGP/BGPsec UPDATE messages

With this capability, BIO can be used as both a traffic generator as well as a traffic monitor.

6.6. Future Considerations

BGPsec-IO can easily be extended with many more features. The capability of inserting UPDATEs during runtime and the extraction of information using the traffic printer can be used to connect it with SDN-based software modules and more. Additionally, with the capability to store UPDATEs using the GEN-B functionality, one could envision storing received raw UPDATEs into a file to be played back later. Even though this capability is currently not activated, all necessary modules are implemented, they just need to be connected in the right form.

Going forward though it would be important to finish the integration of CAPI to allow proper testing and performance evaluation of SCA signature generation. With this feature finished, SCA can be used to test algorithm implementations other than ECDSA.

One more important part of BIO which is not implemented yet but can be a great tool is the integration of the RPKI Cache Test Harness developed with the SRx-Server. Integrating this tool into BIO will allow scripting traffic tests where BIO can provide the public keys to the IUT via the RPKI to Router Protocol as specified in RFC 8210.

7. Additional Implementations Using the SRxCryptoAPI and SRx-Proxy-API

This section deals with additional work performed in which we researched embedding the developed APIs in different technologies. Most of the processing of prefix origin validation and BGPsec path validation is done either in the SRxCryptoAPI or the SRx-Server. The QuaggaSRx implementation mainly uses both to perform the validation portion and uses the result values for policy processing. The reason for this design choice was to allow a fast transition into other implementations. This modular design allows another chosen router platform to also use these API-based modules by accessing their features. With this in mind, routers that want to perform BGPsec must still be able to perform basic handling of the BGPsec_PATH attribute (syntax parsing and constructing/extending the BGPsec_PATH attribute), although they do not need to worry about retrieving PRKI information or any prefix/origin and path validation. Cryptographic processing, key management, as well as RPKI management, and validation processing are provided by the NIST BGP–SRx Framework.

7.1. Design and Build ExaBGP-SRx

ExaBGP-SRx is based on ExaBGP, a highly flexible BGP speaker that allows controlling BGP announcements programmatically. It supports IPv4, IPv6, L2VPN, L3VPN, and FlowSpec to name a few. Similar to BGPsec-IO, the underlying ExaBGP is not a routing engine. It allows generating routing traffic received via STDIN or through Representative State Transfer (REST) API. In addition, ExaBGP-SRx also allows processing the routing information it receives from its peers by dissecting them into messages in plain text or JSON format. These can be further processed by other applications. As shown in Figure 28, ExaBGP-SRx uses ExaBGP as core and extends the protocol processing “Attribute Class” by adding the BGPsec_PATH attribute as well as the protocol handling mechanics.

ExaBGP-SRx got extended by additional processing for MP_NLRI processing regarding IPv4. BGPsec UPDATEs require the IPv4 prefix to be encoded using the MP_NLRI attribute. Furthermore, ExaBGP-SRx requires basic BGPsec_PATH attribute processing for the error detection as required by RFC 8205. Next to the required processing of the modified BGP UPDATE, BGPsec negotiation is required during the session establishment. This extension

was applied to the generation of the BGP OPEN message. ExaBGP-SRx uses SCA to enable BGPsec path validation and signing.

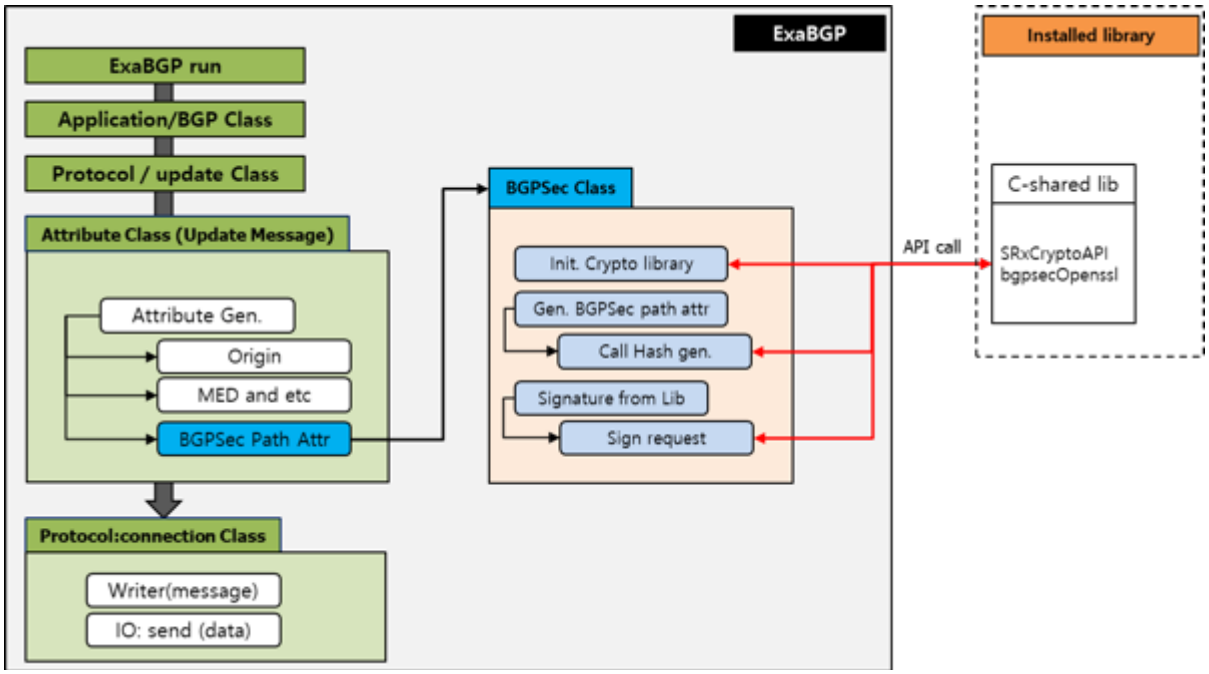


Figure 28 – ExaBGP-SRx BGPsec design

As previously mentioned, ExaBGP-SRx is a Python-based implementation and the NIST BGP-SRx implementation is C-based. For implementations such as ExaBGP-SRx that are written in Python, wrappers are available to be able to integrate these libraries into Python. To accomplish the integration of the C-based libraries, Python provides a module called 'ctypes'. This module allows the embedding of C-based APIs by creating a data and function wrapper between Python and C. Using this wrapper, ExaBGP-SRx can call functions within the linked C-based SRxCryptoAPI library. Using these mechanics, ExaBGP-SRx is able to access SRxCryptoAPI and its embedded BGPsecOpenSSL cryptographic validation/signing module.

7.1.1. Parsing BGP-4 / BGPsec UPDATE Messages Using ExaBGP-SRx

Even though ExaBGP-SRx does not have the feature of RIB processing, it will be able to parse the received BGP and BGPsec messages OPEN, UPDATE, and KEEP-ALIVE. It will process them and write the information gathered into designated log files or STDOUT. This information can then be used by a monitoring agent. As shown in Figure 29, router R3 receives a BGPsec UPDATE from router R2 and forwards it to the ExaBGP-SRx peer. The ExaBGP-SRx peer then will process the UPDATEs and store them into log files. In this case, ExaBGP-SRx does perform BGPsec path validation for all BGPsec UPDATEs by using SRxCryptoAPI. Now, the operator of router R3 can use the output of ExaBGP-SRx to compile policy filters which can be imported into R3. This also could be done automatically by any monitoring agent. At this point, there is no other option on sending the validated UPDATEs back from ExaBGP-SRx to R3.

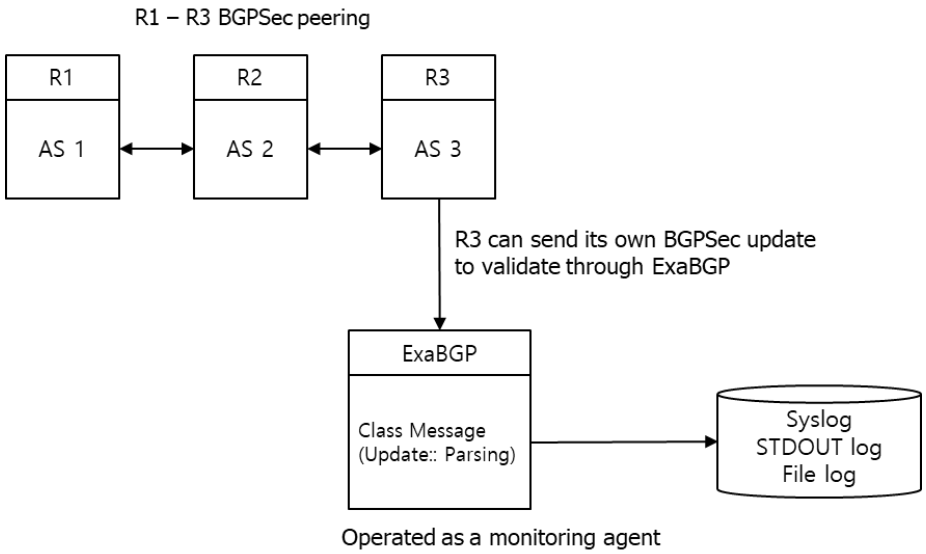


Figure 29 – Parsing BGP message on ExaBGP-SRx

7.1.2. ExaBGP-SRx as BGPsec Traffic Generator

The topology shown in Figure 30 uses three QuaggaSRx routers and one ExaBGP-SRx node. The ExaBGP-SRx node is peering with both, AS 60002 as well as AS 60004, and the peering sessions have negotiated BGPsec. Both routers will also peer with AS 60003.

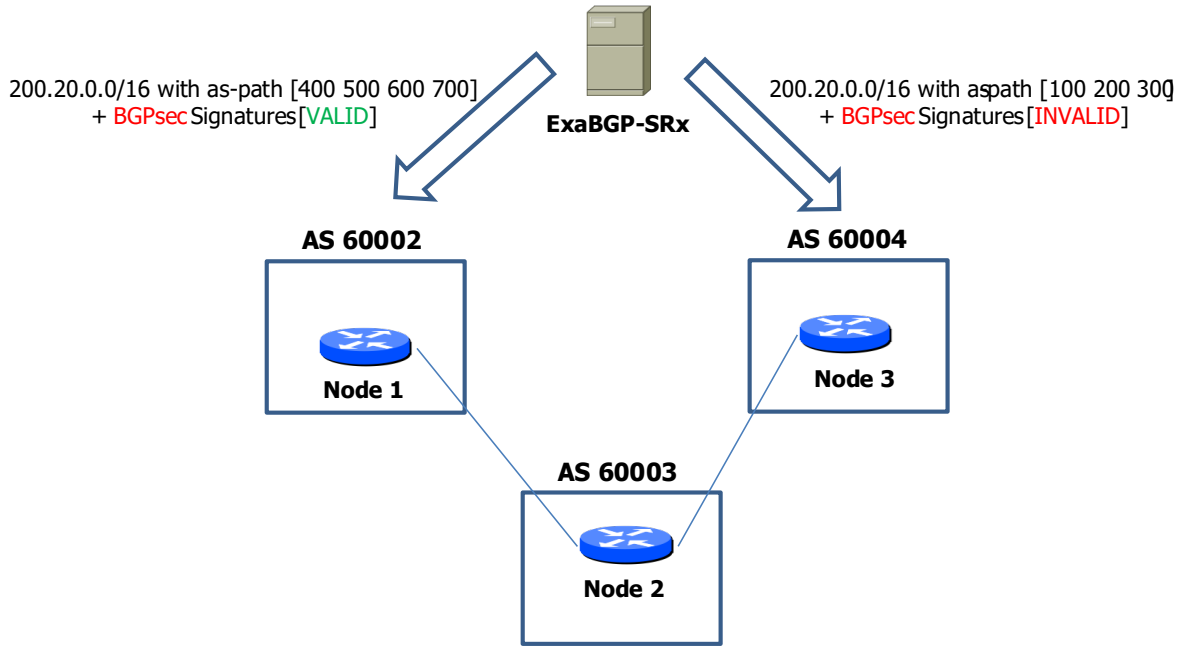


Figure 30 – ExaBGP-SRx simple BGPsec route selection test

ExaBGP-SRx will generate two separate UPDATEs, a longer one for AS 60002 and a shorter one for AS 60004. The shorter Update will fail validation whereas the longer path will pass validation.

Examining the RIB-IN of AS 60003 displayed in Listing 24, it is clear that AS 60003 performed both BGP-OV and BGP-PV and the longer path did pass the validation whereas the shorter path did not. This shows that ExaBGP-SRx is capable of generating proper multi-hop BGPsec UPDATEs.

```
srx_docker_R2_60003> sh ip bgp
BGP table version is 0, local router ID is 172.17.0.4
Status codes: s suppressed, d damped, h history, * valid, > best, i - internal,
              r RIB-failure, S Stale, R Removed
Validation:    v - valid, n - notfound, i - invalid, ? - undefined
SRx Status:    I - route ignored, D - SRx evaluation deactivated
SRxVal Format: validation result (origin validation, path validation)
Origin codes: i - IGP, e - EGP, ? - incomplete

   Ident    SRxVal SRxLP Status Network          Next Hop            Metric  LocPrf Weight Path
*> 8D17E35D v(v,v)              200.20.0.0/16    172.17.0.3                             0 60002 61702 400 500 600 700 i
*  00BB9EF4 i(i,i)                             172.17.0.5                             0 60004 61702 100 200 300 i
```

Listing 24 – RIB-IN table of AS 60003

7.2. Design and Build GoBGP-SRx

GoBGP-SRx is an additional router implementation that uses the NIST BGP-SRx software suite. It is based on the GoBGP BGP-4 router implementation [33] that already provides the functionality of prefix origin validation. This includes the capability to connect to RVCs using the router to cache protocol specified in RFC 6810. To create GoBGP-SRx we extended the core GoBGP implementation by adding BGPsec path processing as specified in RFC 8205. For BGPsec path validation we integrated the SRxCryptoAPI. This was made possible by using the "cgo" module that allows integration of native C libraries (such as the SRxCryptoAPI library for BGPsec path validation) into Go language constructs.

As mentioned above, GoBGP-SRx is based on the open source GoBGP implementation. The core GoBGP implementation is split into two parts: The client implementation which is used to configure and query the router, and the server implementation, which provides the routing engine. The client and server are communicating using the gRPC protocol [34]. This allows the client and server implementation to be implemented in different programming languages as long as they support gRPC.

7.2.1. GoBGP-SRx Using SRxCryptoAPI

The BGPsec integration of GoBGP-SRx is shown in Figure 31. Once the server receives a client request, the '*BgpServer*' module wakes up to process the request. In order to handle BGPsec processing, we added the '*bgpsecManager*' to the server implementation. This new module uses the '*bgpConfig*' module to access user-specific BGPsec configurations such as SKI, key info, etc. For BGPsec path signature generation and path validation, the server employs the "cgo" module to import the C-based SRxCryptoAPI.

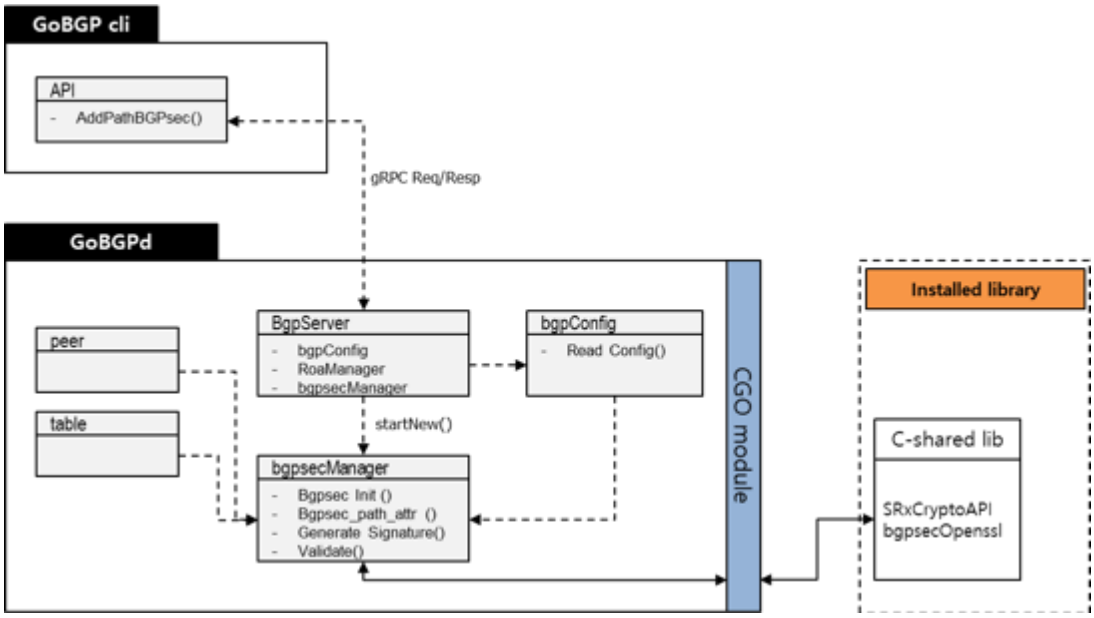


Figure 31 – GoBGP-SRx server design using SRxCryptoAPI

Figure 32 shows the process flow connecting the GoBGP-SRx client and SRxCryptoAPI. The server module *goBGPd* controls the BGPsec peering session with router R1 (AS1) and uses "cgo" calls to facilitate the native C library calls for the signature generation as well as BGPsec path validation. Once the validation is performed and the results are returned, the *goBGPd* daemon determines the proper path selection depending on the validation outcome. In addition, the client, *GoBGP* can be controlled using CLI as well as an optional web-based client.

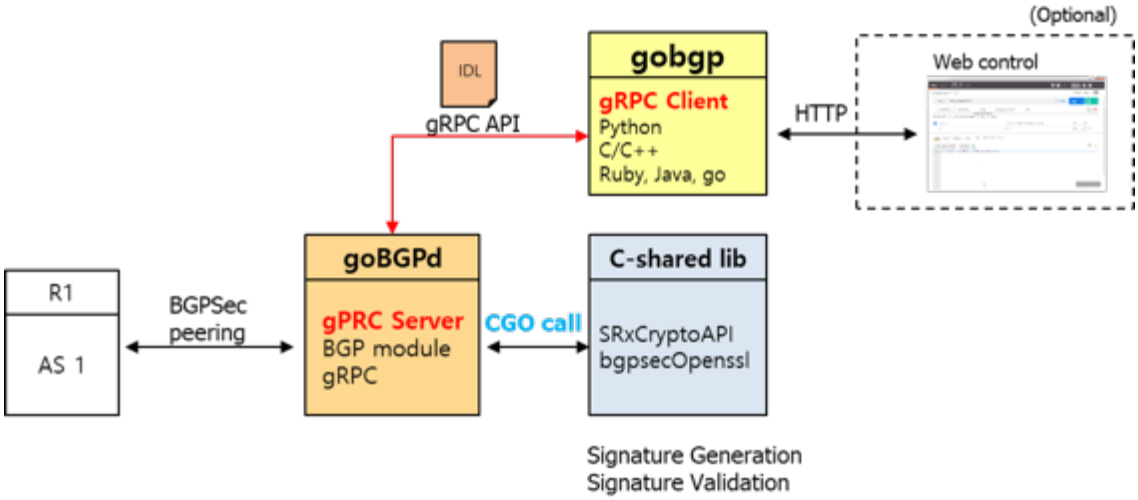


Figure 32 – GoBGP importing C shared library, SRxCryptoAPI, for BGPsec signing and validation

7.3. Interoperability Testing ExaBGP-SRx, GoBGP-SRx, and QuaggaSRx

The interoperability tests use three BGPsec router implementations. Each of the three implementations is installed each on its own virtual machine as shown in Figure 33. AS 65011 using ExaBGP-SRx is generating a three-hop fully signed BGPsec UPDATE and sends it to its BGPsec enabled peer AS 65005. AS 65005, also running BGPsec but with GoBGP-SRx installed is validating the BGPsec UPDATE. GoBGP-SRx is also signing the UPDATE and

sending it to AS 65023. AS 65023 runs QuaggaSRx with BGPsec enabled. QuaggaSRx will validate the BGPsec UPDATE upon receiving and apply the validation result to the UPDATE.

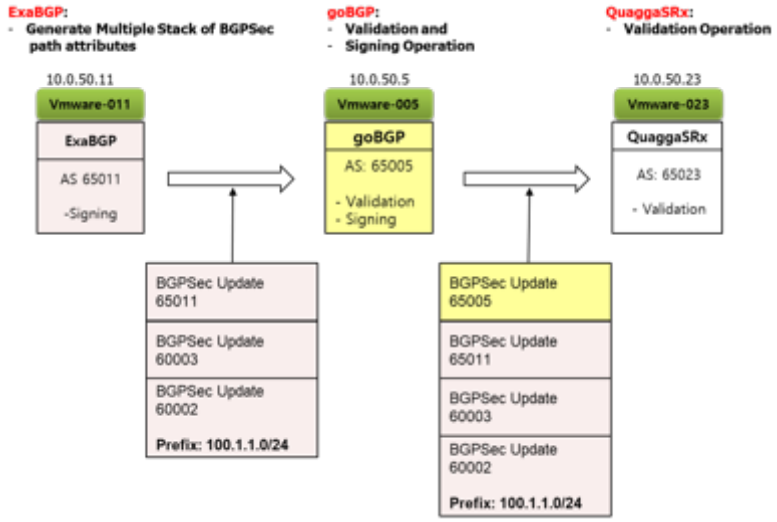


Figure 33 – Experiment setup of interoperability among ExaBGP-SRx, GoBGP-SRx, and QuaggaSRx

As shown in Listing 25, after validating the received BGPsec UPDATE, the validation result returns valid. This testing successfully demonstrates the interoperability between all three BGPsec capable routers.

```
vmware-023> sh ip bgp
BGP table version is 0, local router ID is 10.0.50.23
Status codes: s suppressed, d damped, h history, * valid, > best, i - internal,
              r RIB-failure, S Stale, R Removed
Validation:    v - valid, n - notfound, i - invalid, ? - undefined
SRx Status:    I - route ignored, D - SRx evaluation deactivated
SRxVal Format: validation result (origin validation, path validation)
Origin codes: i - IGP, e - EGP, ? - incomplete

   Ident    SRxVal SRxLP Status Network          Next Hop            Metric  LocPrf Weight Path
*> 309D9E76 v(v,v)             100.1.1.0/24     10.0.50.5                                0 65005 65011 60003 60002 i

Total number of prefixes 1
vmware-023>
```

Listing 25 – BGPsec validation result inside QuaggaSRx

7.4. Extending QuaggaSRx Using gRPC

As described earlier in Section 3.1, the communication between QuaggaSRx and SRx-Server is TCP-based. gRPC is a different technology with a focus on high performance: It is an open-source version of Google's internal framework Stubby [34]. A client-server model using gRPC is able to gain high performance, full-duplex streaming, and transparent communication across languages and platforms. In this section, we describe the design of a gRPC based communication between the SRx-Server and QuaggaSRx using a gRPC based Proxy communicating with the SRx-Server.

7.4.1. Designing the SRx-gRPC-Proxy for QuaggaSRx and SRx-Server

Being able to support the gRPC framework, the SRx-Proxy needs to be replaced with a gRPC capable Proxy, hence the SRx-GRPC Proxy as shown in Figure 34. On the SRx-Server side, a

gRPC Server Service module needs to be added. A gRPC service library is required for both additions.

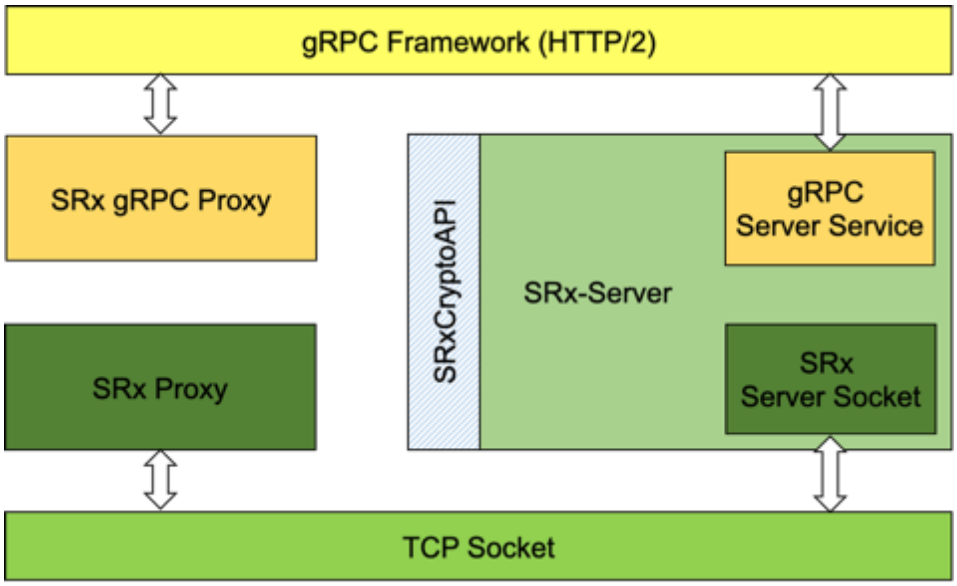


Figure 34 – SRx-Proxy and SRx-gRPC-Proxy implementation design

Using this design, the SRx-gRPC-Proxy required a C wrapper because gRPC does not support C integration yet. The same is required to the SRx-Server side. Once these two interfaces were generated, The QuaggaSRx implementation was able to use the SRx-gRPC-Proxy as if it was an SRx-Proxy, the gRPC technology complete hidden from QuaggaSRx as is the TCP communication using the SRx-Proxy.

For the implementation of the gRPC communication, we chose to use the Go language due to its feature-rich networking support and natural integration into the GoBGP-SRx router. The gRPC Communication module is shown in detail in Figure 35. A simple but effective design was to modify the current SRx-Proxy and exchange the underlying TCP communication with a gRPC driver that performs the C to Go translation. The Go implementation then performs the gRPC communication. On the SRx-Server side, the same happened just in reverse. The gRPC Server Driver receives the requests and forwards the appropriate function calls into the SRx-Server using the Go to C translation modules.

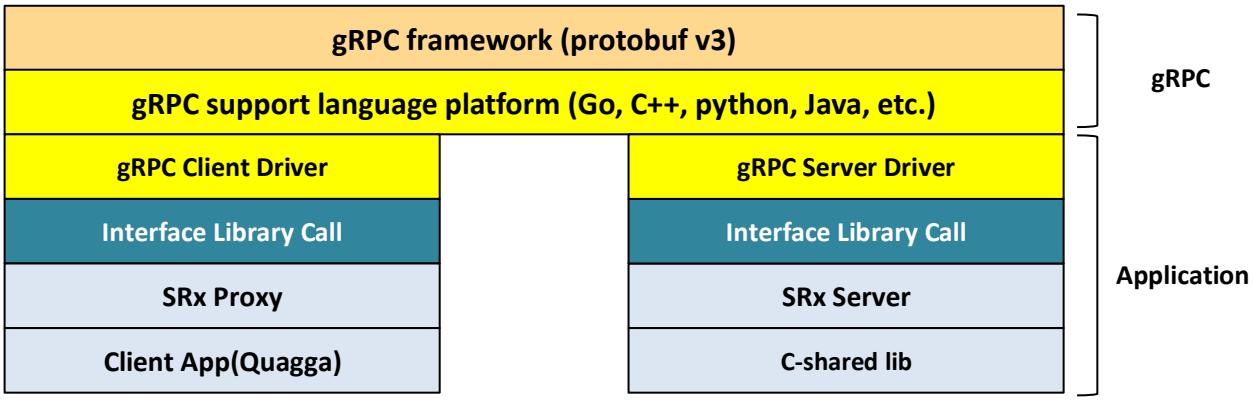


Figure 35 – Layers of gRPC-enabled SRx suite

The gRPC layer is the path in which a serialized traffic is exchanged using the gRPC protocol buffer which is explained in detail in [34].

Library calls for connection between C-based SRx-Server (or SRxProxy) and gRPC-based server (or client) driver also include gRPC service functions (mostly callback functions and concurrent threads).

7.4.2. Integration into QuaggaSRx

Integration into QuaggaSRx is fairly simple: QuaggaSRx by default requires the SRx-Proxy libraries. These can be replaced with the SRx-gRPC-Proxy libraries which then will seamlessly replace the TCP connection between QuaggaSRx and the gRPC Framework to communicate with the gRPC enabled SRx-Server implementation. Figure 36 shows the communication flow between QuaggaSRx and the SRx-Server.

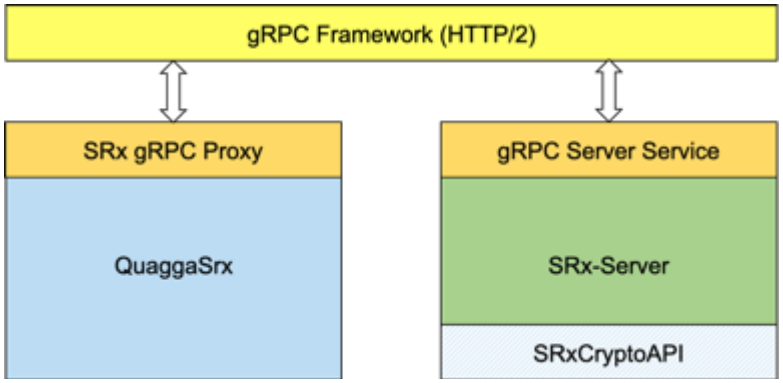


Figure 36 – QuaggaSRx using SRx-gRPC-Proxy

Once Quagga receives a BGPsec update packet from a peering neighbor, it initiates a validation request with the SRx-gRPC-Proxy module as it would using the SRx-Proxy module. The validation request then gets sent to the gRPC-enabled SRx-Server. The gRPC enabled SRx-Server then receives the validation request and proceeds as normal. The only difference is that with the gRPC enabled SRx-Server the chosen communication is gRPC.

7.5. ExaBGP-SRx and GoBGP-SRx Future Consideration

As described above, both implementations, ExaBGP-SRx and GoBGP-SRx are currently using direct access to SRxCryptoAPI through wrapper modules that allow embedding C-based libraries. In this section, we want to present three possible scenarios on how the use of the SRx-Server could be facilitated.

7.5.1. Access SRx-Server Using ExaBGP-SRx-Proxy Module

One possibility of using the NIST BGP-SRx suite is to use a Python-based SRx-Proxy as shown in Figure 37. This can be easily accomplished by embedding the current SRx-Proxy API library into a wrapper in the same way the current implementation uses the SRxCryptoAPI. The SRx-Proxy API too is a C-based library. This will be somewhat more complicated though because this API requires also the registration of a C-based callback function. Another option would be to implement a Python-based proxy that uses the SRx-Server-Protocol and therefore completely avoiding the SRx-Proxy API itself.

Additionally, ExaBGP-SRx will need to build a short RIB-IN cache to allow the SRx-server time to process the validation requests.

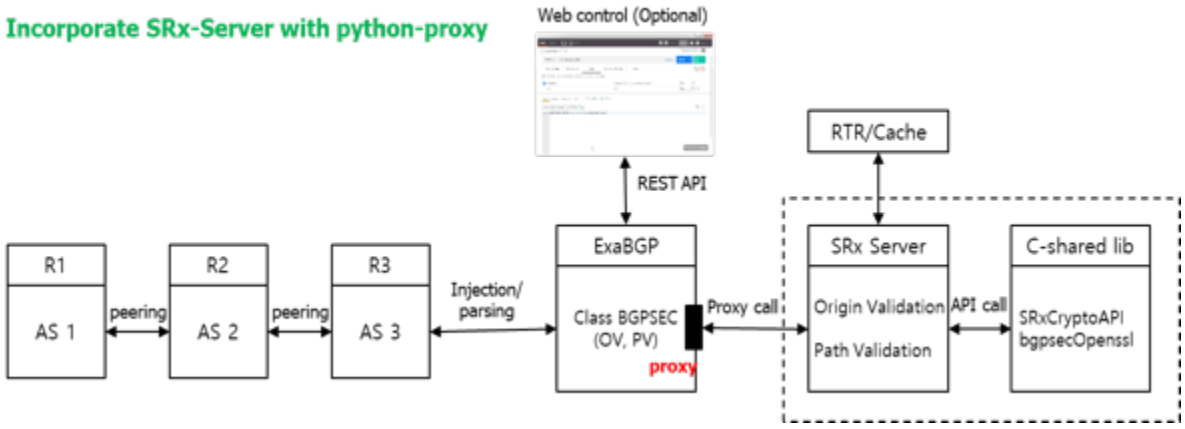


Figure 37 – Access SRx-Server with ExaBGP-SRx Python-Proxy

7.5.2. GoBGP-SRx Using Go-Proxy

In addition to the SRxCryptoAPI integration that allows the GoBGP-SRx router instance to perform BGPsec path validation, the next step is to also integrate the SRx-Server into GoBGP-SRx. Figure 38 shows an integration using a Go implementation of the SRx-Proxy implementation that utilizes TCP communications as specified in the SRx-Server-Protocol. This would not need any C-wrapper libraries, as it would use the Go-provided TCP implementation.

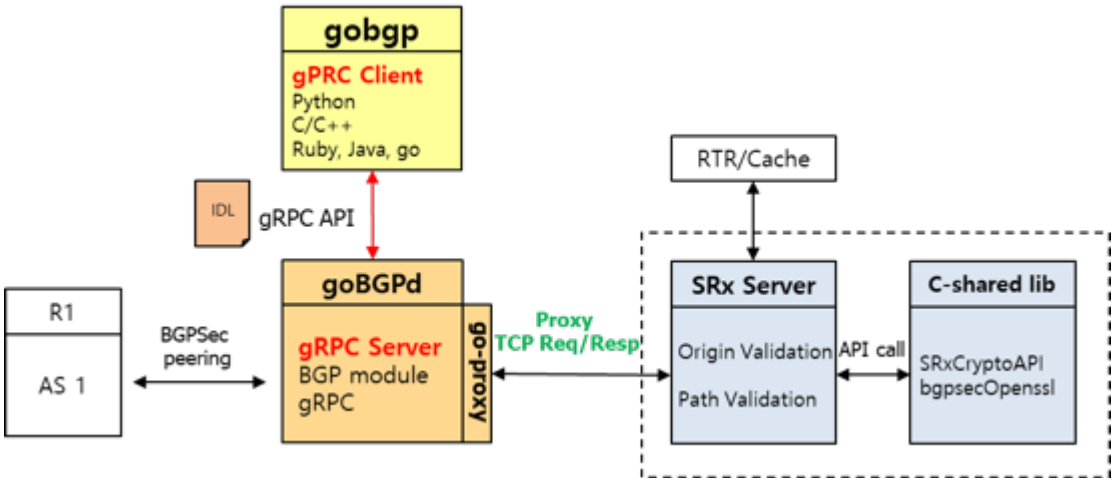


Figure 38 – GoBGP using Go-Proxy using TCP communication

7.5.3. GoBGP-SRx Using gRPC to SRx-Server

The final implementation showcased here is the usage of the gRPC Client as shown in Figure 39. This is already developed and embedded in the SRx-gRPC-Proxy implementation. The Interface Description Language (IDL) is defined, and the SRx-Server side is implemented. The last missing piece is the GoBGP-SRx integration which is not hard to accomplish.

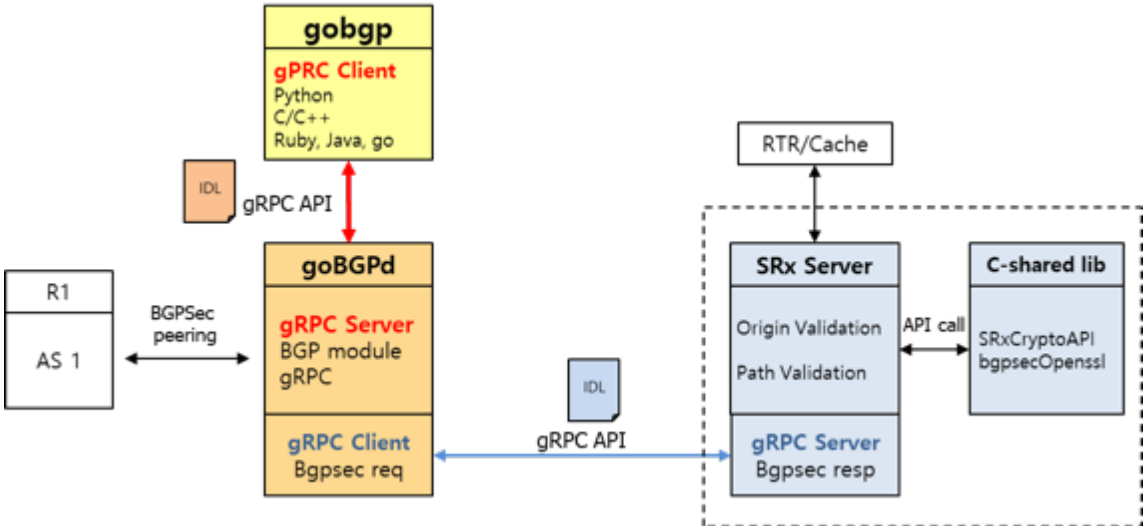


Figure 39 – GoBGP inter-operation using gRPC framework to gRPC enabled SRx Server for BGPsec signing and validation

8. Performance Measurements and Comparisons

Performance measurement in BGPsec has two aspects to it. The first thing that comes to mind is the processing time required by cryptographic operations. Signing and validating BGP UPDATEs is considered costly and worrisome to operators. The questions that arise are what the processing overhead is like, and how long will it take for the routing table to converge. Furthermore, many routing platforms that are still deployed do not provide the required cryptographic implementation. Using BGPsec-IO (see Section 6), the measurement of performance of cryptographic implementations is fairly simple: just measure the time needed by the cryptography module to perform BGPsec update validations. This is a crude form, but it provides an idea of the performance differences between different implementations. To facilitate this, the SRxCryptoAPI, as introduced in Section 5, provides an API that allows exchanging cryptographic implementations through configuration rather than compilation. This allows to easily compare multiple implementations using the same input parameters.

The other concern is the BGPsec UPDATE format, which allows only one prefix per UPDATE. In contrast, in BGP-4, many prefixes are announced in a single UPDATE message by employing prefix packing. Prefix packing means that multiple prefixes that share the same path attributes can be packed into one single UPDATE. The reason prefix packing is not possible in BGPsec is because signatures span over the AS path and the prefix. If the signature were to span over multiple prefixes and the AS path, then packing, unpacking, and repacking prefixes along the path would be infeasible.

This does have an impact when it comes to regular BGP-4 UPDATE processing. Many factors come into play and policy processing is regarded as a very costly operation. Without prefix packing, processing requirements will increase. Prefix packing also reduces the traffic between two routers, especially during a table transfer. According to the NIST RPKI Deployment Monitor [26] from June 2021, the global routed Internet consists of approximately 1.1 million unique prefix origin pairs. Depending on the average number of prefixes per update, the number of UPDATEs required to be sent will be reduced inversely proportionately. Considering that one of the demanding operations in routing is policy processing, packing prefixes makes a significant difference in BGP-4 workload. With multiple prefixes being packed into a single UPDATE, the policy engine only needs to run once for each UPDATE. Depending on the complexity of the policy itself, parsing through a large number of policy elements each time an UPDATE is received can be very costly. Now with BGPsec having the prefix packing feature removed, the impact on the convergence time will be noticed. Maybe future additions to RFC 8205 would allow similar mechanics for UPDATEs but this is out of scope for this publication.

In the following sections, we compare the performance of the cryptographic portion of BGPsec path validation. We will compare the performance across all SRx implementations (Quagga, GoBGP, etc.) presented in this technical note.

8.1. Performance Testing Using Quagga and QuaggaSRx

The experimentation setup uses two "off the shelf" desktop computers called BGPSEC-1 and BGPSEC-2. Both systems are equipped with an Intel XEON X5 3.5 GHz processor, and 16 GiB of RAM. BGPSEC-1 functions as a traffic generator using BGPsec-IO to generate multi-hop fully signed BGPsec UPDATE messages. BGPSEC-2 is configured with QuaggaSRx and functions as a measurement platform. Measurement hooks were installed into the router implementation to allow setting measurement pointers. For this experiment, the SCA is configured to use the BGPsecOpenSSL crypto module, which is explained in detail in Section 5.5. Additionally, QuaggaSRx is configured to perform BGP-PV directly on the router platform as soon as the UPDATE arrives. The SRx-Server implementation is only used for BGP-OV. All required public keys are pre-loaded into the SRxCryptoAPI and available directly for the BGPsecOpenSSL crypto module.

As mentioned earlier, hooks were temporarily embedded into the QuaggaSRx router implementation to allow setting measurement points as shown in Figure 40. The intent was to measure the time it takes from receiving the first UPDATE until the last received UPDATE is fully processed and fully validated by the SCA. These two measurements are carried out: (1) the period of receiving BGPsec updates and (2) the processing time of BGPsec path validation from the first update arrival at the QuaggaSRx to the completion of the validation process for the last update.

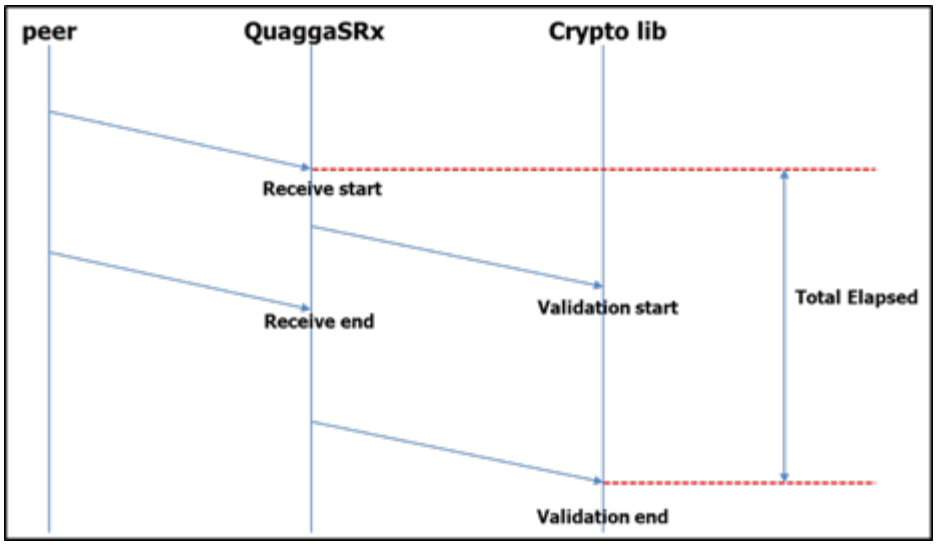


Figure 40 – Flow chart for time intervals on receive, validation and total elapsed time

As depicted in Figure 41, we also repeated the same experiment with the stock Quagga implementation using BGP-4 UPDATEs to see the difference in processing performance between BGP-4 UPDATE processing and BGPsec UPDATE processing. We increased the number of prefixes gradually from 1K to 10K and up to 100K UPDATEs.

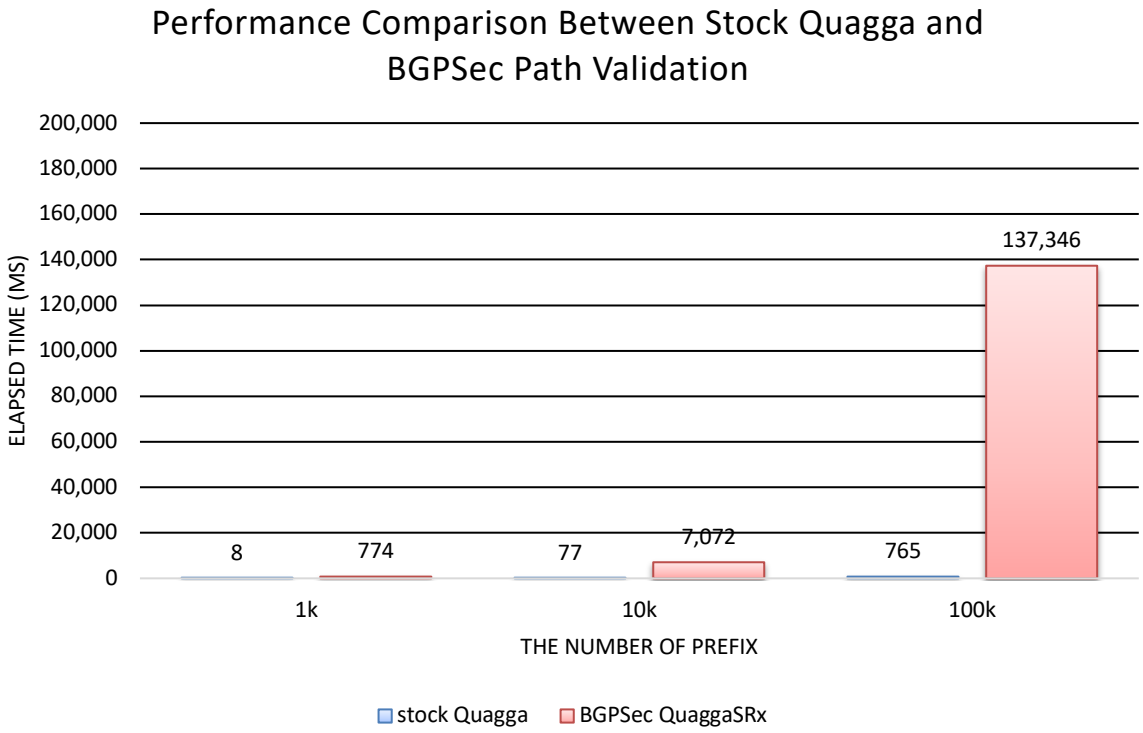


Figure 41 – Performance comparison between stock version of Quagga and QuaggaSRx for BGPsec PV

The processing time for the stock version of Quagga (the blue-colored bars in Figure 41) linearly increased with the number of updates received. That is not the case for BGPsec

QuaggaSRx. For the case of Quagga for 100 K UPDATEs, the elapsed time is still under one second, while the corresponding number for BGPsec Quagga SRx is slightly over 137 s.

Studying BGPsec Quagga SRx processing time data as shown in Figure 41 in further detail, the processing time from 1K and 10K UPDATEs linearly increased in the same ratio as the increase in the number of prefixes. But for the increase from 10K to 100K UPDATEs, the time increased approximately 20 times. BGPsec processing time in QuaggaSRx increases nonlinearly with the number of UPDATEs.

To verify if the BGPsec UPDATE signature validation itself has anything to do with the increase of the processing time, we stress-tested the BGPsecOpenSSL validation processing using BGPsec-IO in CAPI mode. This test, shown in Figure 42, was performed on a different system with a XEON X5660 2.8 GHz and 16 GiB of RAM. We wanted to see is if the BGPsecOpenSSL implementation changes the signature validation speed over time. The measurement only applies to the time from the signature validation call until SCA returns with the validation result.

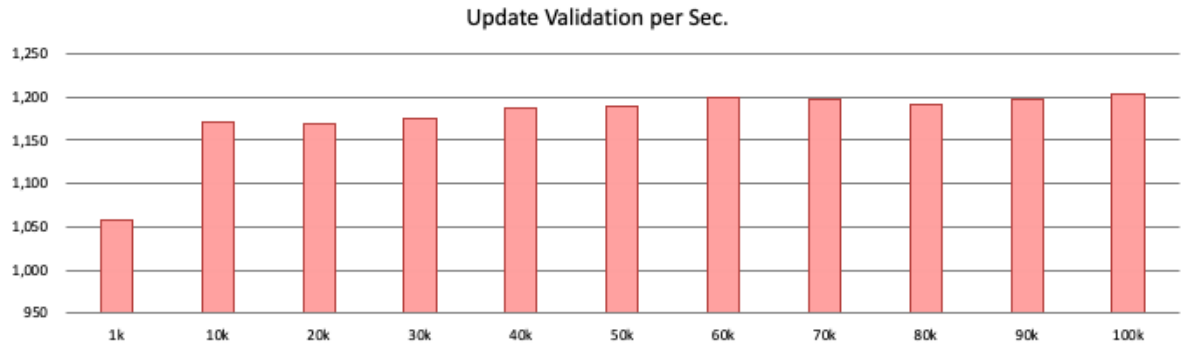


Figure 42 – Average time of 4 hop BGPsec UPDATE validation using BGPsecOpenSSL

As shown during this test, the average processing time stays roughly the same, somewhere between 1 180 to 1 200 UPDATEs per second, which is not a bad number in itself considering the unoptimized and non-parallelized implementation of the signature validation. Therefore, the cryptographic execution (signature validation) seems not to be the reason for the jump observed in Figure 41.

This leaves the only explanation that the increasing amount of memory must have an impact on the processing. One major difference between BGPsec and BGP is that in BGPsec, UPDATEs cannot be stored in an optimized form as they can in BGP-4. Even if in BGP-4 prefix packing is disabled, the router itself identifies the updates' attributes shared and therefore stores the attributes only once for multiple prefixes that share them. In BGPsec, this is not the case since the signatures, which cover the prefix and path, are unique for each path and are an integral part of the path itself. Therefore n prefixes received via BGPsec require n paths stored in the routing table which cannot be optimized. Other previous tests during the development showed performance loss within QuaggaSRx starting at about 70K unique UPDATEs.

8.2. Performance Testing Using Inline Validation vs. Lazy Evaluation

It is important to recognize the BGPsec performance might be enhanced by reducing the BGPsec UPDATE processing burden from the router implementation to a 3rd party server, such as the SRx-Server. Figure 43 shows the overall measurements between the path validation processing on the router side (inline) and the processing with lazy evaluation enabled using the SRx-Server. Lazy evaluation describes the process of assigning a pre-set validation state to the UPDATE and then sending the UPDATE for validation to the SRx-Server. Once the validation request is sent out, the router uses the pre-set validation state and performs the route selection. Once the SRx-Server calculates the validation, it reports back to the router and in case the reported validation state differs from the validation state used during route selection, the router will restart the route selection for the route in consideration. Using this method, the router does not need to extensively parse the UPDATE. More importantly, the validation can be done on a separate system, and therefore both the SRx-Server as well as the router do not interfere with each other by competing for computing resources or memory.

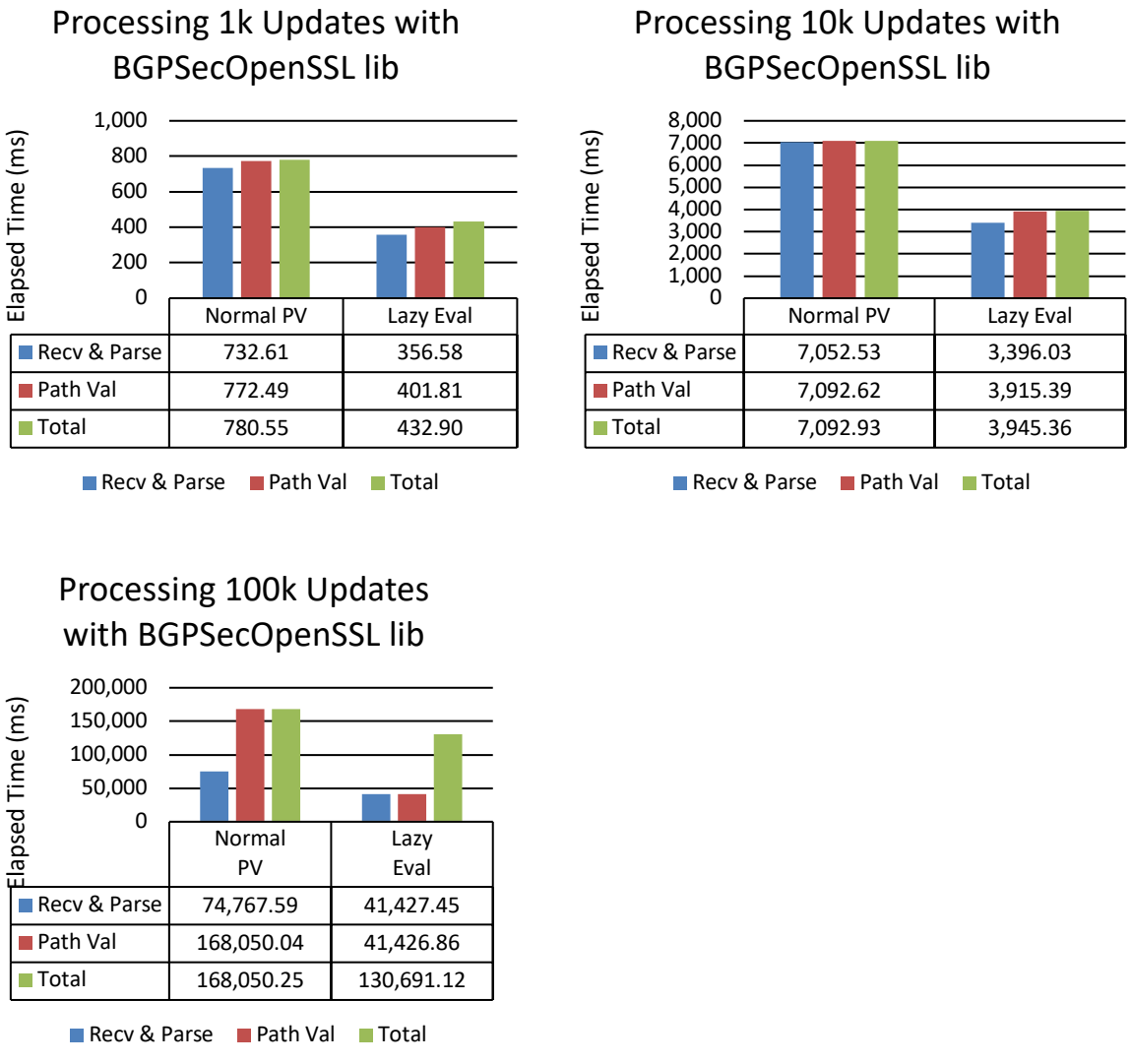


Figure 43 – Comparison of BGPsec path validation between a normal path validation and lazy evaluation

As shown in Figure 43, lazy evaluation using SRx-Server has a better performance in each case. All times are averages. Receive & parse (blue) is performed in parallel to path validation (red). The total value (green) contains both. Lazy evaluation does cut the processing roughly in half. The outlier here again is the processing of 100K UPDATEs. Though looking closer it becomes evident that the overall processing has a significant component other than receive & parse and path validation. Especially the Lazy Evaluation mode shows clearly that the path validation (signature verifications) is not the issue here. The main issue is related to the internals in QuaggaSRx, most likely the internal handling of the UPDATE tables. Clearly with BGPsec, the router cannot achieve the memory savings as it can in BGP-4 by the using AS path sharing mechanism. In BGPsec, even if the AS path is the same, the path attributes (BGPsec_PATH) in the UPDATEs differ due to the signatures (which cover the prefix and path). Therefore, it is important to examine the use of hash tables and optimize the code in this direction.

8.3. Performance Analysis between Open Source BGP-4 Implementations

The performance of a router is measured by the convergence of BGP UPDATES. The metric can be defined in two ways:

(1) Total convergence time: The time from the initiation of the BGP connection to the moment when the processing of all received UPDATEs is complete.

(2) Table convergence time: The time from receiving the first UPDATE to the time when the last UPDATE is fully processed.

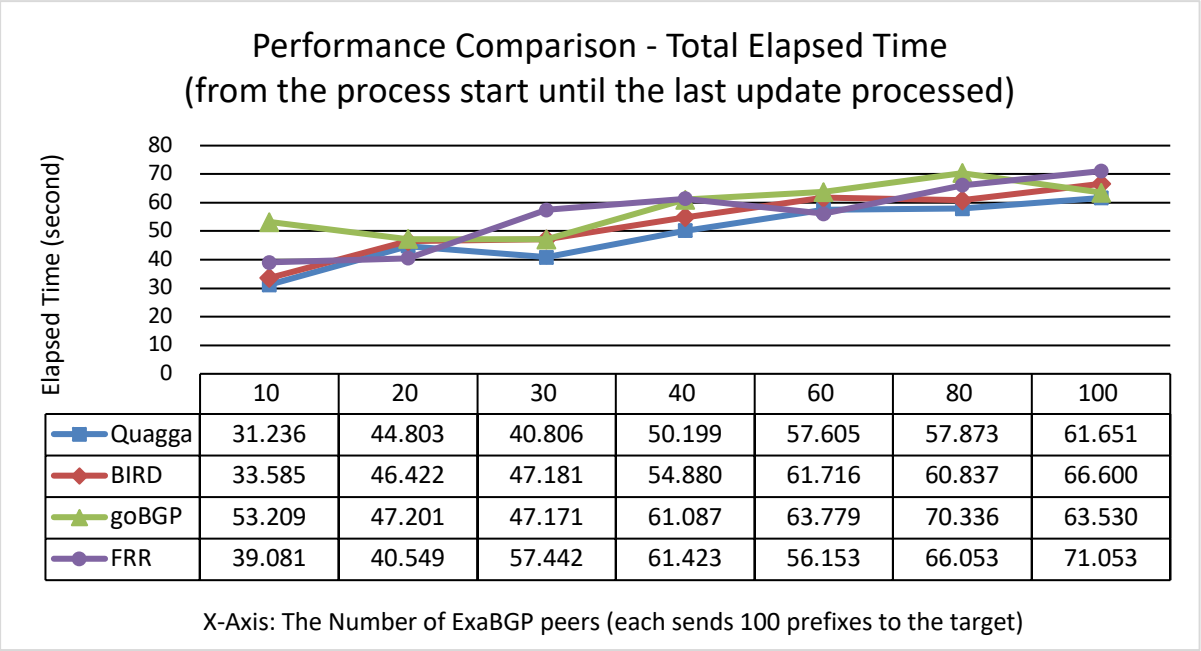
The first one is the initial convergence time; the second one can be considered the convergence time when processing a table refresh.

For this experimentation, we look at five different open source BGP routing platforms and compare them just using BGP-4 UPDATEs. As test subjects we chose the following implementations:

- Quagga v.1.0.2
- BIRD v 1.6
- ExaBGP v 3.4.19
- FRR v 2.0.0
- GoBGP v.1.13

Peers are started in its separate containers and each peer is sending BGP-4 traffic to the router implementation under test (IUT) which runs in a container as well. The container technology used is Docker. The peers are each running an instance of ExaBGP, sending 100 UPDATEs to the respective IUT. The IUT's used are Quagga, BIRD, goBGP, and FRR.

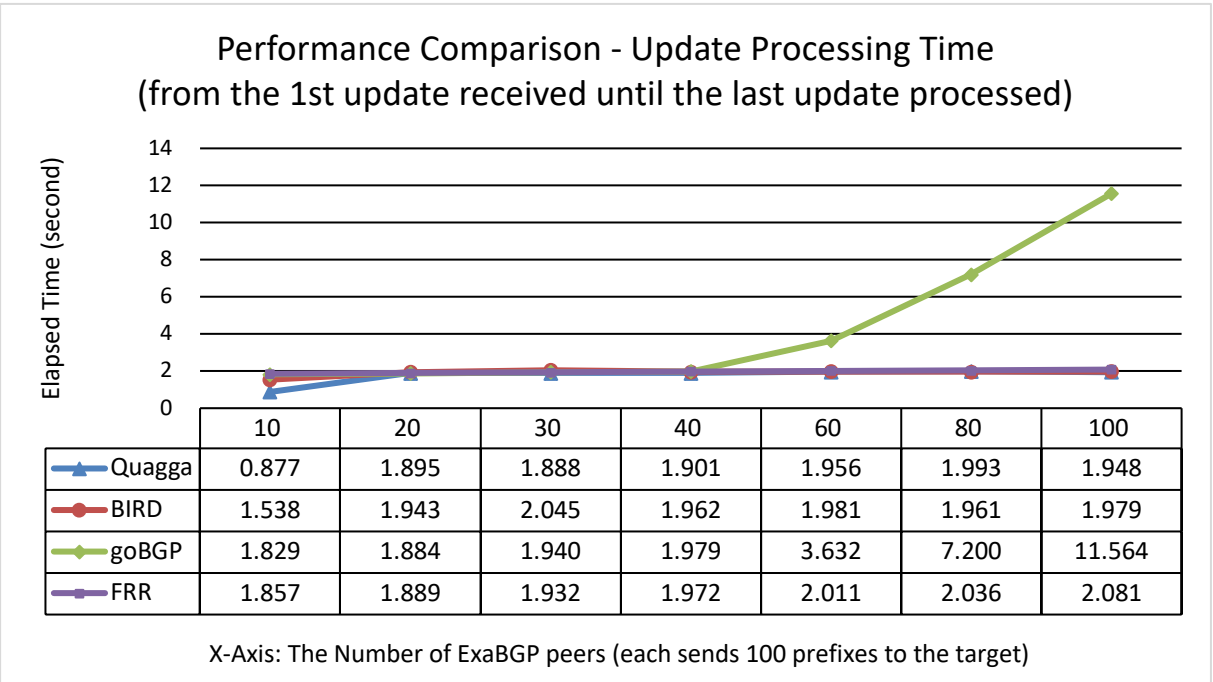
For the testing, we vary the number of peers for each run as follows: 10, 20, 30, 40, 60, 80, and 100 peers. As shown in Figure 44 the x-axis represents the number of parallel peering sessions and the Y-axis represents the time taken for initial convergence in seconds. Looking at the plots in Figure 44, it appears that about 30 seconds are used for the session negotiations and setup and from thereon the time increase is almost linear (especially for Quagga and BIRD). Also, all 4 implementations seem to perform roughly in the same ballpark in terms of the initial convergence.



	10	20	30	40	60	80	100
Quagga	31.236	44.803	40.806	50.199	57.605	57.873	61.651
BIRD	33.585	46.422	47.181	54.880	61.716	60.837	66.600
goBGP	53.209	47.201	47.171	61.087	63.779	70.336	63.530
FRR	39.081	40.549	57.442	61.423	56.153	66.053	71.053

Figure 44 – Total convergence time comparison among BGP open-source implementations

Looking only at convergence time without including the session establishment time (in other words, the time from receiving the first UPDATE to when the last UPDATE is processed), the data plots in Figure 45 look different compared to the results shown above for initial convergence. Looking at these plots, we see an unexpected behavior in the case of goBGP implementation starting with 40 concurrent sessions and up, namely, the convergence time shoots up although it still stays in the range of multiple seconds.



	10	20	30	40	60	80	100
Quagga	0.877	1.895	1.888	1.901	1.956	1.993	1.948
BIRD	1.538	1.943	2.045	1.962	1.981	1.961	1.979
goBGP	1.829	1.884	1.940	1.979	3.632	7.200	11.564
FRR	1.857	1.889	1.932	1.972	2.011	2.036	2.081

Figure 45 – Table convergence time among BGP open-source implementations

8.4. Comparison of GoBGP-SRx, QuaggaSRx, and gRPC Enabled QuaggaSRx

For the performance evaluation of BGPsec path validation, we employed two virtual machines using Intel XEON E5-2699 V3 2.5 GHz and 8 GiB of RAM. ExaBGP-SRx was used as a traffic generator and the receiver used QuaggaSRx, gRPC enabled QuaggaSRx, and the GoBGP-SRx implementations. The experimentation applied loads of 1K, 10K, and 100K BGPsec encoded UPDATEs for all three router implementations and an additional case of 500K BGPsec encoded UPDATEs for the GoBGP-SRx implementation.

The measurements start with the first BGPsec UPDATE received and stop after the router finished processing the last received BGPsec UPDATE. As shown in Figure 46, the GoBGP-SRx implementation performed the best. This experimentation suggests that the reason for the performance decay observed in the QuaggaSRx, as well as the gRPC-enabled QuaggaSRx implementations, is not the BGPsec path validation. If this would be the case, we also would observe the same degradation with GoBGP-SRx. This points to inefficient memory handling of the BGPsec UPDATE data within both flavors of the QuaggaSRx implementation.

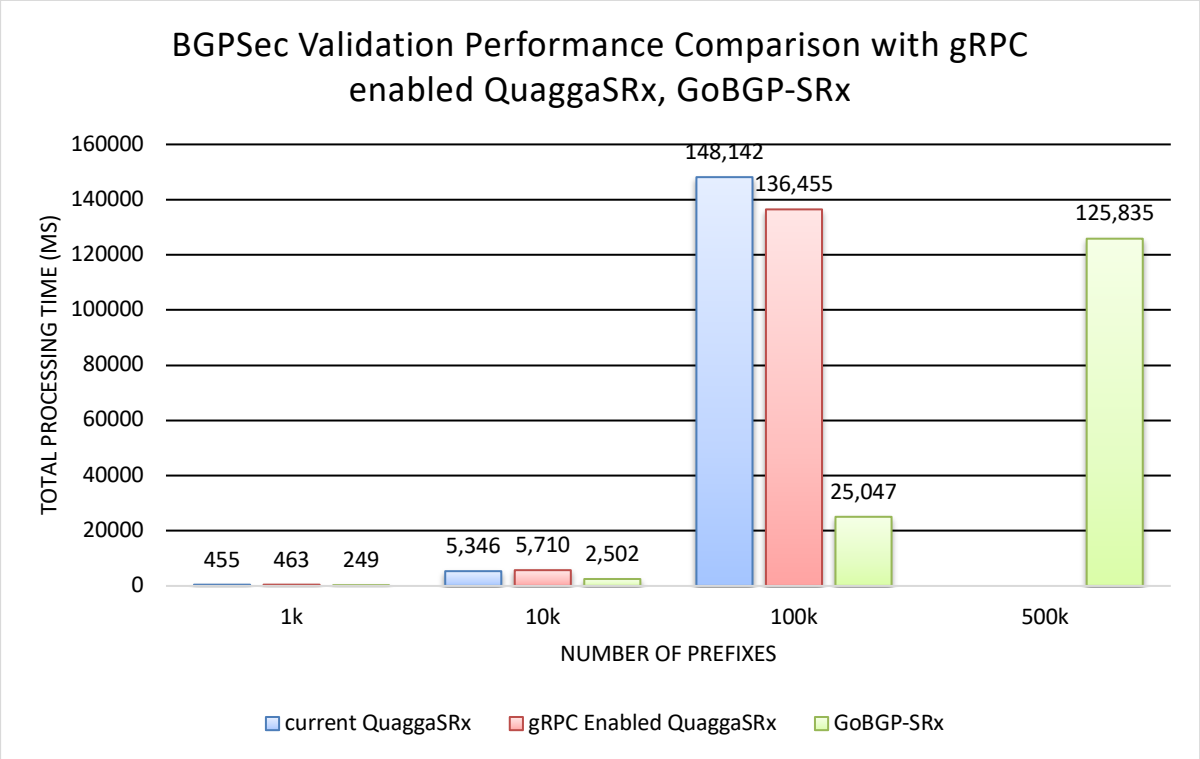


Figure 46 – BGPsec validation performance comparison

A second interesting observation is that during the first two tests the raw TCP socket implementation did show better performance than the gRPC enabled implementation. The 100 K results might be skewed by other issues within the QuaggaSRx's internal management. It would be interesting to repeat the experimentation first in an increment of 10 K UPDATEs followed by an increment of 5 K UPDATEs to identify the transition point of BGPsec UPDATE at which QuaggaSRx shows the performance degradation.

8.4.1. SRxCryptoAPI's Validation Time among BGPsec Implementations

To evaluate the performance of cryptographic calculation with the SRxCryptoAPI library, we started an additional experiment analyzing the cryptography library's processing portions out of the entire processing time. Figure 47 compares the cost of cryptographic processing against BGPsec update processing in the QuaggaSRx and GoBGP-SRx implementations. Here we used a single BGPsec UPDATE in multiple runs and took the average to compare between both router platforms. The graph depicts the overall processing (red) next to time used only for the cryptographic processing (blue). Looking at the overall UPDATE processing time between GoBGP-SRx (817 microseconds) and QuaggaSRx (1 047 micro-seconds) the implementation of GoBGP-SRx only uses 78% of the processing time QuaggaSRx uses. Looking closer into the results depicted in Figure 47, both implementations – with a neglectable difference of 8 micro-seconds – use almost the same time for the cryptographic operation. What stands out is that QuaggaSRx does have a higher cost in non-cryptographic related UPDATE processing than GoBGP-SRx.

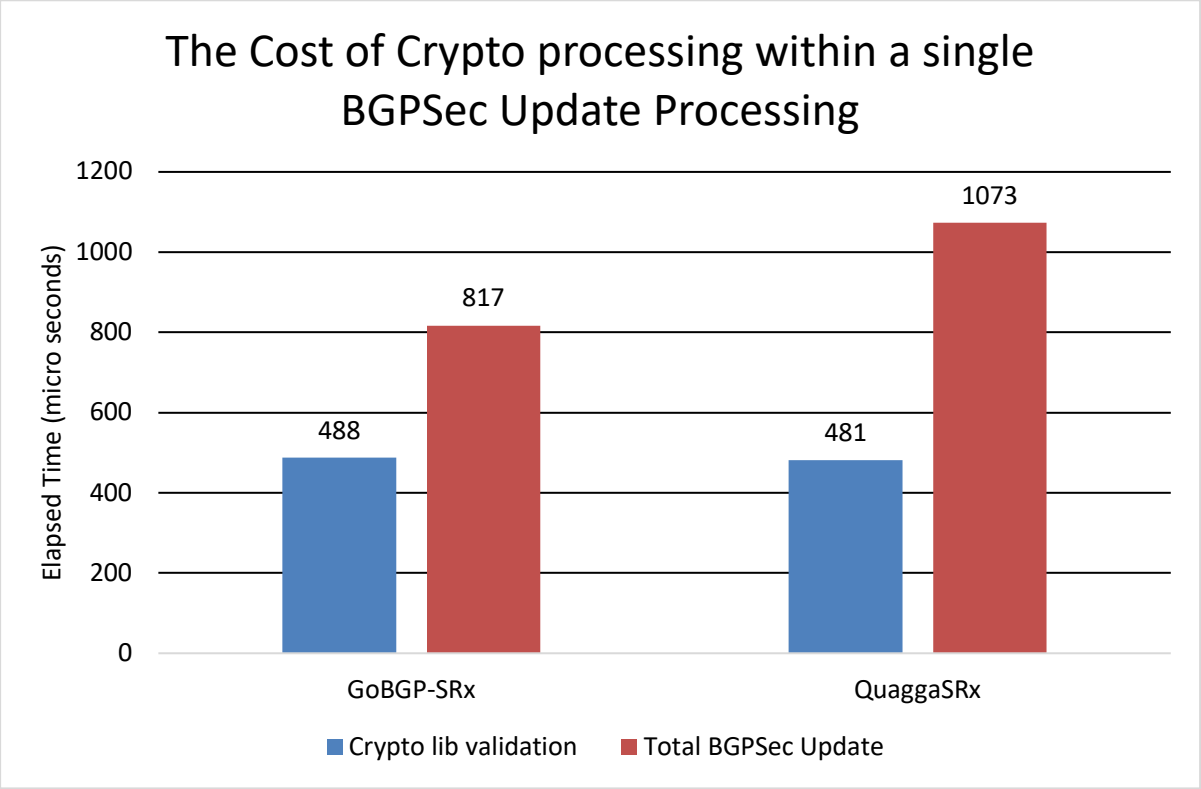


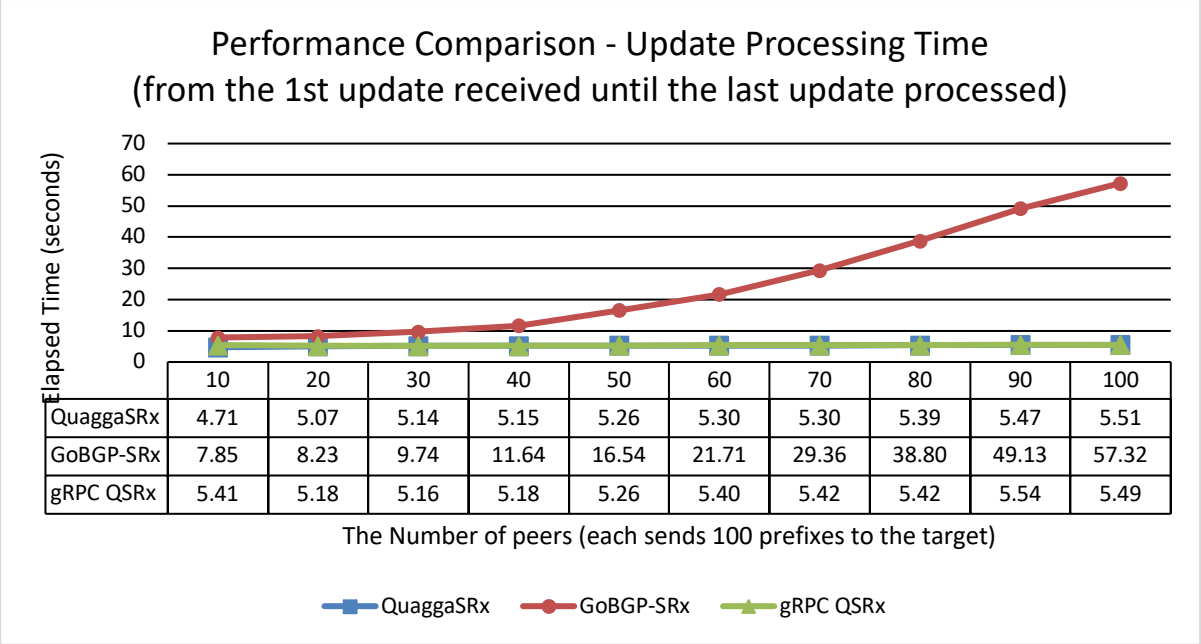
Figure 47 – Cryptographic processing cost vs. BGPsec UPDATE processing using GoBGP-SRx & QuaggaSRx

8.4.2. BGPsec Validation Processing Using Multiple Docker Based Peers

Earlier we saw (in Figure 46) that GoBGP-SRx performs better than QuaggaSRx when there is only a single peer. Here we address the question of what happens if the same amount of load comes from multiple peers. QuaggaSRx performed quite poorly compared to GoBGP-SRx at a load of 100 K UPDATEs in the case of all the load coming from one peer (see Figure 46). However, in the lower ranges of load (1 K, 10 K UPDATEs), QuaggaSRx performance was comparable to that of GoBGP-SRx. Looking back at Figure 45, while comparing BGP-4 router implementations, we noticed that GoBGP started showing problems when the number of peers

increased even though the total load was in the 1 K to 10 K range; the delay had shot up sharply for GoBGP as the number of peers increased. We decided to perform the same test (e.g. varying the number of peers) using QuaggaSRx, gRPC enabled QuaggaSRx, and GoBGP-SRx to see if the same behavior can be observed. We used the same configuration as previously described using Docker Containers, except this time we used ExaBGP-SRx instead of ExaBGP as a traffic generator.

Each peer announces 100 UPDATEs. The only changing variable is the number of peering sessions. As shown in Figure 48, as the number of peers increased, QuaggaSRx and gRPC-enabled QuaggaSRx did not change in the total processing time (e.g. the duration from the 1st update received to the last update processing completed). However, GoBGP-SRx processing time slowly increased up to 40 peers and after that, it increased far more rapidly from 40 to 100 peers. This is similar to what was seen in Figure 45 for BGP-4 comparing Quagga vs. GoBGP. This means that if the same amount of load (5 K to 10 K UPDATEs) comes from multiple peers (rather than one peer), then QuaggaSRx and gRPC-enabled Quagga SRx perform significantly better than GoBGP-SRx.



	10	20	30	40	50	60	70	80	90	100
QuaggaSRx	4.71	5.07	5.14	5.15	5.26	5.30	5.30	5.39	5.47	5.51
GoBGP-SRx	7.85	8.23	9.74	11.64	16.54	21.71	29.36	38.80	49.13	57.32
gRPC QSRx	5.41	5.18	5.16	5.18	5.26	5.40	5.42	5.42	5.54	5.49

Figure 48 – BGPsec performance comparison among QuaggaSRx, GoBGP-SRx, and gRPC enabled QuaggaSRx

8.5. Final Take Away

As takeaways based on the performance studies, we have some key observations to note. It appears that Quagga and GoBGP are optimized for different functions. Quagga is optimized for large numbers of peers, but not for large numbers of UPDATEs, whereas GoBGP is optimized for handling a large number of UPDATEs and performs poorly if the same large number of updates come from many peers (rather than one peer). It would be interesting to see how GoBGP and GoBGP-SRx would perform if GoBGP's poor handling of a large number of peering sessions is improved, and conversely, it would also be interesting to see how much better Quagga and QuaggaSRx can be if Quagga's internal data storage were optimized.

9. Deployment

This NIST BGP-SRx project started back in 2008 / 2009 when BGP Origin Validation and BGPsec Path Validation were still in their infancy. It became clear that BGP-OV requires BGP-PV to complete securing route announcements. The flaws in BGP-OV, the main one being that an attacker can forge the origin by pre-pending it to the path and therefore result in an BGP-OV being marked as a "valid" route, can only be undeniable detected by a mechanism that allows attesting the path the route took from originator to destination. For this BGPsec is required because it allows verifying each segment of the path the UPDATE traversed in. It also was clear that the infrastructure that is to be created will be complex and that there is a need for reference implementations, that allows testing the design of all the different mechanisms in parallel to the design to identify flaws, bottlenecks as well as performance improvements right during the design. That was the time when NIST decided to start the NIST BGP Secure Routing Extension, now known as NIST BGP-SRx.

Almost 12 years later, many RFCs were developed, the RPKI is actively being deployed, almost 300 000 (29%) global prefixes are validated as "valid" with almost 23% of all announced Address Spaces in /24 blocks (as of June 2021) participating in the global RPKI [26].

BGPsec was specified as RFC 8205 but since then sadly not much implementation has been done. While we are still working in wrapping up missing specifications such as BGPsec path validation signaling [27] which is, at the time of writing this document, an active IETF SIDROPS working group document, other specifications might be needed to facilitate the deployment of BGPsec.

This work demonstrated that it is very much possible to implement BGPsec. The regular hardware can accomplish easily the task of segment verification of routes and the way RFC 8205 is written allows temporary pausing validation until the router is able to handle the processing. As we describe in Section 10, deployment for BGPsec requires innovation. Implementations such as taraBGPsec and SRx-Server show that it is possible. An SRx-Server or similar implementation deployed in the cloud or centralized on-premise, and to be shared by multiple routers within the enclave can easily solve the overhead produced by BGPsec.

10. Final Thoughts and Future Plans

A project of this size always can have features added or implementations optimized. This reference implementation did help significantly in developing the necessary IETF standard RFCs. In particular, the BGPsec Protocol Specification was considerably influenced by performance issues identified within the reference implementation. For instance, the order of fields within the Secure Path section of the draft that became RFC 8205 was modified due to programmatic difficulties identified within this reference implementation.

For future work, especially in BGPsec, it seems evident that the chosen design of out-sourcing is the right path to go. Thus it is a logical next step to bring the SRx-Server-Protocol developed for this implementation to the IETF and propose it as a standard. Once this protocol is standardized, it will allow other community members to step up and possibly implement production-ready high-performance BGPsec validation servers. Also, the router vendors would not have to implement the complete BGPsec validation and could use this protocol to outsource

this task, similar to how the router-to-cache protocol (RFCs 6810 and 8210) does today by allowing to outsource the acquisition and validation of RPKI certificates.

Another thought is to standardize the SRxCryptoAPI that allows the modularization of BGPsec cryptography and opens the market further to vendors of high-performance cryptographic implementation independent of router or validation developers.

Additionally, we can see that proposing the SRx-Server-Protocol as a standard draft within the IETF could foster innovation in outsourcing BGPsec path validation. This project demonstrates that outsourcing BGPsec path validation is possible and reduced the burden of BGPsec processing from the router. Standardization of this protocol might kickstart other implementers to be innovative in this area. Old routers that do not have the capability of any cryptographic function can use this protocol to open a simple TCP session to an external validator. Vendors only need to implement the protocol and let others do the validation. It is similar to RPKI origin validation where validation caches perform the validation of certificates and compile a simple list that the router can use to match prefix and origin without ever performing any cryptographic function.

Regarding this reference implementation, we identified multiple areas of improvement / modification.

- Complete separation of BGP-OV and BGP-PV within QSRx.

  Currently, the final validation is calculated by combining BGP-OV and BGP-PV validation results (see Table 2). Adding policies that consider each validation result BGP-OV and BGP-PV separately instead of combined as it is done today, provides more freedom in scripting policies.

- Adding information on why the validation state of an UPDATE changed.

  RPKI V-to-I: Some ROA must have disappeared to render it invalid. The interesting portion is to identify if only a matching ROA disappeared or along with the disappearance, a new matching ROA also appeared that rendered the value invalid rather than not-found?

- Adding information to validation states such as “invalid but covered” would signal to the router to examine the FIB for less specific covering prefixes.

- Use BGPsec validation on parallel BGP peering sessions.

  An AS that mainly operates BGP-4 could spin up a parallel BGPsec session using a separate router. All routers could connect to the SRx-Server which is capable of BGPsec validation. Then with each received BGP-4 UPDATE, the router could send a verification request to the SRx-Server to query if a BGPsec update exists with the “same” AS-PATH and if yes then what is the BGPsec path’s validation result.

Here "same" AS-PATH detection algorithm could be implemented in different flavors:

- Consecutive:
  "same consecutive list of unique AS numbers" which would render "A A B C D D E" same as "A B C D E" ignoring concatenations.
- Exact:
  "A A B C D D E" same as "A A B C D D E" but **not** the same as "A B C D"

- Adding the signaling of BGPsec validation state to iBGP sessions. (The IETF SIDROPS working group currently has an active working group document in progress on this topic.)
- Adding the RPKI Cache Test harness as a component to BIO.

Acknowledgments

The authors are grateful to Antonio Izquierdo Manzanares for his review and comments.

Acronyms

Selected acronyms and abbreviations used in this paper are defined below.

AFRINIC	African Network Information Center
APNIC	Asia Pacific Network Information Center
ARIN	American Registry for Internet Numbers
BGP	Border Gateway Protocol
BGPsec	BGP security extension
BIO	BGPsec-IO traffic generator
BGP-PV	BGPsec Path Validation
eBGP	External BGP
ECDSA	Elliptic Curve Digital Signature Algorithm
iBGP	Internal BGP
IDS	Intrusion Detection System
IETF	Internet Engineering Task Force
IUT	Implementation Under Test
LACNIC	Latin America and Caribbean Network Information Center
NCCoE	National Cybersecurity Center of Excellence
RFC	Request for Comments
RIB	Routing Information Base
RIPE	Réseaux IP Européens
RIPE NCC	RIPE Network Coordination Center
ROA	Route Origin Attestation
BGP-OV	BGP Origin Validation
RPKI	Resource Public Key Infrastructure
RVC	RPKI Validation Cache
SCA	SRxCryptoAPI
SRx	Secure Routing Extension
SSVOPM	Signature and Verification Operations Parallelizing Manager
SUID	SRx update ID (Unique within the scope of SRx-Server)
TA	Trust Anchor
VRP	Validated ROA Payload

References

[1] RFC6811, “BGP Prefix Origin Validation”, 2013, P. Mohapatra, J. Scudder, D. Ward, R. Bush, R. Austein, https://tools.ietf.org/html/rfc6811

[2] RFC6810, “The Resource Public Key Infrastructure (RPKI) to Router Protocol”, 2013, R. Bush, R. Austein, https://tools.ietf.org/html/rfc6810

[3] RFC8097, “BGP Prefix Origin Validation State Extended Community”, 2017, P. Mohapatra, K. Patel, J. Scudder, D. Ward, R. Bush, https://tools.ietf.org/html/rfc8097

[4] RFC8210, “The Resource Public Key Infrastructure (RPKI) to Router Protocol – Version 1”, 2017, R. Bush, R. Austein, https://tools.ietf.org/html/rfc8210

[5] RFC8205, “BGPsec Protocol Specification”, 2017, M. Lepinski, K. Sriram, https://tools.ietf.org/html/rfc8205

[6] RFC8608, “BGPsec Algorithms, Key Formats, and Signature Formats”, 2019, S. Turner, O. Borchert, https://tools.ietf.org/html/rfc8608

[7] RFC4271, “A Border Gateway Protocol 4 (BGP-4)”, 2006, Y. Rekhter, T. Lee, S. Hares, https://tools.ietf.org/html/rfc4271

[8] Charter – Secure Inter-Domain-Routing WG (SIDR), 2006, https://datatracker.ietf.org/doc/charter-ietf-sidr/

[9] RFC8635, “Router Keying for BGPsec”, 2019, R. Bush, S. Turner, K. Patel, https://tools.ietf.org/html/rfc8635

[10] RFC8654, “Extended Message Support for BGP”, 2019, R. Bush, K. Patel, D. Ward, https://tools.ietf.org/html/rfc8636

[11] “Quagga Routing Suite”, https://www.quagga.net/

[12] “BIRD Internet routing daemon”, https://bird.network.cz/

[13] “The BGPsec enabled Bird Routing Daemon”, http://www.securerouting.net/tools/bird/

[14] “RSYNC”, https://rsync.samba.org

[15] RFC8182, “The RPKI Repository Delta Protocol (RRDP), 2017, T. Bruijnzeels, O. Muravskiy, B. Weber, R. Austein, https://tools.ietf.org/html/rfc8182

[16] American Registry for Internet Numbers (ARIN), https://www.arin.net

[17] Regional Internet Registry for Europe, the Middle East, and parts of Central Asia (RIPE), https://www.ripe.net

[18] Asia Pacific Network Information Centre (APNIC), https://www.apnic.net

[19] The Internet Addresses Registry for Latin America and Caribbean (LACNIC), https://www.lacnic.net

[20] The Internet Numbers Registry for Africa (AFRINIC), https://afrinic.net

[21] “SRx-Server-Protocol”, O. Borchert, K. Lee, P. Gleichmann https://github.com/usnistgov/NIST-BGP-SRx/raw/version5/srx-server/doc/srx-server-protocol-2-1.pdf

[22] “RPKI Validation State Unverified”, 2018, O. Borchert, D. Montgomery, https://tools.ietf.org/html/draft-borchert-sidrops-rpki-state-unverified-00

[23] “BGPsec Validation State Unverified”, 2018, O. Borchert, D. Montgomery, https://tools.ietf.org/html/draft-borchert-sidrops-bgpsec-state-unverified-00

[24] "High Performance BGP Security: Algorithms and Architectures", NANOG-69, 2017, M. Adalier, K. Sriram, O. Borchert, K. Lee, D. Montgomery https://archive.nanog.org/sites/default/files/1_Sriram_High_Performance_Bgp_v1.pdf (slides); https://www.youtube.com/watch?v=Yp03po5WJP0 (video)
[25] Implementation of Patricia Trie, Dave Plonka, http://pages.cs.wisc.edu/~plonka/Net-Patricia/
[26] "NIST RPKI Deployment Monitor", https://rpki-monitor.antd.nist.gov
[27] "BGPsec Validation Signaling", IETF SIDROPS working group draft, O. Borchert, D. Montgomery, D. Kopp, https://datatracker.ietf.org/doc/draft-sidrops-bgpsec-validation-signaling/
[28] FIPS-186-4, "Digital Signature Standard (DSS), 2013, National Institute of Standards and Technology, http://dx.doi.org/10.6028/NIST.FIPS.186-4
[29] NIST SP-1800-14A, "Protecting the Integrity of Internet Routing: Border Gateway Protocol (BGP) Route Origin Validation", June 2019, W. Haag, D. Montgomery, A. Tan, W. C. Baker, https://www.nccoe.nist.gov/sites/default/files/library/sp1800/sidr-nist-sp1800-14a-final.pdf
[30] RFC6979, "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", August 2013, T. Pornin, https://tools.ietf.org/html/rfc6979
[31] "Observations during Testing of Route Origin Validation", IETF SIDROPS working group proceedings, IETF 103, November 2018, O. Borchert, K. Lee, D. Montgomery, K. Sriram, https://www.ietf.org/proceedings/103/slides/slides-103-sidrops-observations-during-testing-of-route-origin-validation-00
[32] "Verification of AS_PATH Using the Resource Certificate Public Key Infrastructure and Autonomous System Provider Authorization", IETF SIDROPS working group draft, A. Azimov, E. Bogomazov, R. Bush, K. Patel, J. Snijders, September 10, 2020, https://tools.ietf.org/html/draft-azimov-sidrops-aspa-verification-01
[33] GoBGP: BGP implementation in Go, GitHub Repository, https://github.com/osrg/gobgp
[34] "gRPC, A high -performance, open-source universal RPC framework", https://grpc.io
[35] "Proxmox Virtual environment, open-source server management platform for enterprise virtualization", https://www.proxmox.com
[36] "Design and analysis of optimization algorithms to minimize cryptographic processing in BGP security protocols," Computer Communications, volume 106, pages 75-85, July 2017, V.K. Sriram, D. Montgomery, https://doi.org/10.1016/j.comcom.2017.03.007
[37] "Hurricane electric Internet Services", https://bgp.he.net/report/prefixes
[38] NIST RPKI Monitor, https://www.rpki-monitor.antd.nist.gov
[39] "BGP Update Reports", Online Weekly, G. Huston, http://bgp.pataroo.net/index-upd.html

Appendix A: Supplemental Materials

A.1: QuaggaSRx Configuration Settings

This appendix provides the comprehensive list of configuration settings added to the Quagga implementation during the integration of BGP-OV and BGP-PV. All added commands start with the keyword **srx** followed by the command and its parameters.

When entering the commands using the CLI, the following command sequence is used to enter the configuration section for each separate router instance.

```
enable
configure terminal
router bgp <asn>
```

A.1.1 QuaggaSRx Configuration Settings

Turn on/off additional SRx display information for default show commands.

```
[no] srx display
```

The SRx proxy id MUST be set prior to using the connect command. The SRx-Server uses the proxy ID to link updates to routers. This is can be either scripted as an IPv4 address or plain 4-byte integer decimal value. It is recommended to use the router-id as a proxy-id.

```
srx-proxy-id <id>
```

Configure the address of the server and its port without connecting.

```
srx set-server <host> <0..65535>
```

Connect the BGP server instance to the SRx-Server at the given location. The preferred method to connect is using “srx set-server” to configure the SRx-Server connection and calling “srx connect” without any parameters. The Quagga command “*show/write running-config*” displays the preferred sequence.

```
srx set-server <host> <0..65535>
```

Disconnect the BGP server instance from the SRx server. For this command, the “keep-window” setting is used.

```
srx disconnect
```

Specify the time in seconds the SRx is requested to hold information after it is deleted! This allows a router reboot without losing the validation result information within SRx.

```
srx keep-window
```

With QuaggaSRx 0.3.1 communicating origin validation results to peers via extended community as specified in RFC 8097 was introduced. The given integer value in the range of 0-255 is used to identify this extended community value number. This number should be 200 as specified in RFC 8097 but at the time of implementation the number was not yet assigned, and the implementation allowed to configure a value. By default, this enables communication in send and receive mode for all iBGP peers. In addition to the draft specification, QuaggaSRx allows extending the community into eBGP by adding the parameter '*include_ebgp*'. To turn off eBGP reconfigure the router using '*ibgp_only*'

```
srx extcommunity <0-255> ( ibgp_only | include_ebgp )
```

Disable RFC 8097 for signaling origin validation results.

```
no srx extcommunity <0-255> ( ibgp_only | include_ebgp )
```

RFC 8654 allows increasing the regular BGP update message size from 4K to 64K. The following command enables (or disables by using "*no*") this feature. The optional parameter "liberal" allows a more liberal approach as specified in earlier versions of the specification when it was in draft mode.

```
[no] srx capability extended [liberal]
```

The liberal mode means that even if the capability is not negotiated within the peers, QuaggaSRx will accept large UPDATE messages and not treat them as malformed UPDATEs.

A.1.2 QuaggaSRx BGP-OV and BGP-PV Configuration Settings

To enable or disable the policy processing within the decision process as well as activating or deactivating the ignore flag due to ignore-XXX policies, the evaluation must be enabled. In addition to enabling the evaluation of validation results, this command specifies the mode the evaluation is performed in. The mode *origin_only* indicates that only route origin validation results are used for calculating the UPDATEs validation result. The mode *bgpsec* indicates that both the route origin validation and the BGPsec path validation result will be used to determine the UPDATEs final validation result.

To disable the evaluation use "no srx evaluation".

```
[no] srx evaluation (origin_only | bgpsec [distributed])
```

Mode: *origin_only* (default)

With this setting, only origin validation is evaluated. Path validation results will still be requested and notifications from SRx will be processed for maintaining the correct data associated with each update, but the results of path validation will not be included in the evaluation of validation results.

The following results are possible with origin_only validation processing:

`valid`	A ROA exists that covers the announced prefix and origin.
`notfound`	No ROA exists for the announced prefix or a less specific of it.
`invalid`	A ROA exists that covers the announced prefix or a less specific prefix, but the origin AS does not match.
`undefined`	Validation is not yet performed.

QuaggaSRx introduces a fourth validation result type called "*undefined*". This result type allows distinguishing between an actual validation result and the status when no connection from the QuaggaSRx router to SRx-Server exists, or not enough information is available to make the final decision on the validation result for the update. As soon as QuaggaSRx can determine the outcome of the validation, then the validation result is set to the specific validation state.

Mode: *bgpsec* [*distributed*]

This evaluation mode activates origin validation and path validation. QuaggaSRx uses the validation results of origin validation and path validation to compute the final BGPSEC validation result (valid|invalid|undefined). Even though SRx-Server reports prefix-origin validation and path validation independently as soon as they are available. Note that the SRx-Server path validation refers only to the validation of the path signatures, NOT including the origin validation. QuaggaSRx merges the independent results of origin and path validation into one final BGPSEC validation.

Option: *distributed*

The option distributed specifies the location where the BGPsec path validation will take place. If provided, the BGPsec path validation will be done by SRx-Server, if omitted, the BGPsec path validation will be performed locally within the router itself. In the latter case, all keys must be pre-installed in the router. To use the validation cache for retrieving the router keys (public keys), the validation must be performed by the SRx-Server.

The final validation result is explained in detail in Table 2 of Section 3.4.

A.1.3 QuaggaSRx Policy Configuration

QuaggaSRx allows configuring the default value for BGP-OV and BGP-PV. These values are used unless validation results are received via extended community string as specified in RFC 8097 or until the final validation result is computed. The following two commands allow specifying the default values for BGP-OV and BGP-PV.

```
[no] srx set-origin-value ( valid | notfound | invalid | undefined )
[no] srx set-path-value ( valid | invalid | undefined )
```

Policy processing is discussed in detail in Section 3.5. Here we only list the configuration settings for completeness.

```
[no] srx policy (ignore-notfound | ignore-invalid | ignore-undefined)
[no] srx policy local-preference (valid | notfound | invalid) <value>
                                 [add | subtract]
[no] srx policy prefer-valid
```

A.1.4 Configure a BGPsec Peering Session

BGPsec requires some more configuration to perform as expected. For that, QuaggaSRx allows two modes to do just that.

1. Perform all BGPsec operations within the router itself, or
2. Use SRx-Server to perform BGPsec operations.

For internal operation, QuaggaSRx uses the SRxCryptoAPI, which must be configured properly (see Section 5.2). QuaggaSRx also uses SCA as storage for the private key.

QuaggaSRx allows to store two keys but will only use a single key at a time. To specify each key the following configuration command will be used.

```
srx bgpsec ski (0 | 1) (1..255) (key-ski)
```

With version 0.4.2 QuaggaSRx allows having 2 separate private keys installed. In addition, each key can be assigned its separate algorithm suite identifier. The key SKI is the hexadecimal representation of the 20-octet long SKI (40 hexadecimal ASCII characters) that specifies the private key. The private key will be loaded using the SRxCryptoAPI.

The following command specifies, which of the two BGPsec key configurations is activated.

```
srx key (0 | 1) active
```

BGPsec will be configured on a per-peer basis. The following command allows specifying the correct peer/neighbor configuration.

```
neighbor A.B.C.D bgpsec (send | receive | both)
```

Currently, QuaggaSRx provides BGPsec configuration for IPv4 only!

A.1.5 CLI Display Commands

For the display, QuaggaSRx seamlessly integrates validation information into the standard "*show [ip] bgp"* commands. The additional information must be enabled or disabled within using the "*srx display*" command as described in Appendix A.1.1.

There are two separate display sections. The first one is embedded within the regular Quagga display. Available are commands such as "show ip bgp" or the more detailed version of it, "show ip bgp a.b.c.d/e", where a.b.c.d/e represents a BGP prefix. The latter provides additional outputs such as validation results, update ID, or additional BGPsec path information – just to name a few.

Command: **show ip bgp**

```
bgpd# show ip bgp
BGP table version is 0, local router ID is 10.0.6.50
Status codes: s suppressed, d damped, h history, * valid, > best, i - internal,
              r RIB-failure, S Stale, R Removed
Validation:    v - valid, n - notfound, i - invalid, ? - undefined
SRx Status:    I - route ignored, D - SRx evaluation deactivated
SRxVal Format: validation result (origin validation, path validation)
Origin codes: i - IGP, e - EGP, ? - incomplete

   Ident    SRxVal SRxLP Status Network        Next Hop   Metric  LocPrf Weight Path
*> 289682FD v(v,v)              10.80.0.0/16   10.80.0.1       0              0 80 ?
*  328B5F6E i(i,v)          I   10.80.0.0/24   10.70.0.1       0              0 70 90 ?
```

Command: **show ip bgp 10.80.0.0/16**

```
bgpd# show ip bgp 10.80.0.0/16
BGP routing table entry for 10.80.0.0/16
Paths: (1 available, best #1, table Default-IP-Routing-Table)
  Advertised to non peer-group peers:
  10.0.6.60 10.0.6.70
  80
    SRx Information:
      Update ID: 0xA9945EA7
      Validation:
        prefix-origin: valid
        path:   valid
        bgpsec: valid  (combination of prefix-origin and path validation)
      PathType: BGPSEC-Path ( 1 signature blocks, each with 1 path segments)
        signature block #1: algorithm suite id 1
        path segment 1: as=80; pcount=1
          signature segment [1]: block 1, ski=18494DAA1B2DFD80636AE943D9DC9FF42C1AF9D9
    10.80.0.1 from 10.0.6.80 (10.80.0.1)
      Origin incomplete, metric 0, localpref 100, valid, external, best
      Last update: Tue Sep  8 21:06:56 2020
```

The second display section is used to display the srx configuration. This command is located in the router's configuration section. The configuration section can be reached using the command combination:

```
bgpd> enable
bgpd# configure terminal
bgpd(config)# router bgp <asn>
bgpd(config-router)
```

Once in the router configuration the Quagga using the command '*show srx-config*' will display the current srx configuration settings.

```
bgpd(config-router)# show srx-config
SRx-Server configuration settings:
  srx-server.....: 127.0.0.1
  port...........: 17900
  proxy-id.......: 0x0A000632 (10.0.6.50 - 167773746)
  keep-window....: 900
  connected......: true
BGPSEC configuration settings:
  active key.....: 0
  Private key 0 - active
  - algorith-id..: 1
  - ski..........: FB5AA52E519D8F49A3FB9D85D495226A3014F627
  - DER-loaded...: yes
  Private key 1 - inactive
  - algorith-id..: 0
  - ski..........: 0000000000000000000000000000000000000000
  - DER-loaded...: no
SRx Router Configuration settings:
  evaluation.....: bgpsec (prefix-origin and path processing) using srx-server
  default value..: (origin)   ? = undefined
  default value..: ( path )   ? = undefined
  policy.........: ignore-invalid
                   prefer-valid
  ext community..: 200
  incl-eBGP......: true
bgpd(config-router)#
```

A.2: SRx Server and Tools

The SRx-Server codebase also provides a rich set of test harness tools. These tools were originally developed to test the client or server-side of network connectivity. This Appendix will list the configurations for all, the SRx-Server as well as the SRx-Server-Tools.

A.2.1: SRx Server Configuration

The SRx-Server configuration is located on the file systems /etc folder. The following list shows the complete configuration file.

```
verbose  = true;
loglevel = 5;
#log     = "/var/log/srx_server.log";

sync    = true;
port    = 17900;
```

```
console: {
  port = 17901;
  password = "x";
};

rpki: {
  host = "localhost";
  # Default port (RFC6810) is 323 but needs root privilege on the server
  # side
  #port = 50001;
  port = 323;
  # supports 2 versions: 0 => RFC6810, 1 => RFC8210
  router_protocol = 1;
};

bgpsec: {
  # Allows to set a configuration file for path validation
  #srxcryptoapi_cfg = "<configuration file>";

  # Synchronize the logging settings of SCA with the logging settings of
  # SRx-Server. If set to false the SCA configuration takes precedence
  sync_logging = true;
};

mode: {
  no-sendqueue = true;
  no-receivequeue = false;
};

mapping: {
#The configuration allows 255 pre-configurations. client_0 is invalid
  client_1  = "2";
  client_10 = "10.0.0.1";
  client_25 = "10.1.1.2";
};
```

A.2.2: SRx-Server Telnet Commands

Once logged into the SRx-Server telnet session, the command **help** produces the output shown below. The telnet session does not provide up and down arrow history and no command completion.

```
Available commands are:
=======================================================
 close, quit, exit     Close this console!
 shutdown <password>   Shutdown the SRx Server!
 log-level [number]    Set/display the log level of the server.
                       3=ERROR, 5=NOTICE, 6=INFO, 7=DEBUG
 rtr-sync [proxyID]    Send synchronization request to
                       the provided proxy or all.
 show-version          Display the full version number of the SRx-server.
 show-srxconfig        Display the configuration of the srx server
 show-update <cmd>     Display update data according to
                       the command string.
      cmd:= id <id>    Show the update with the ID (hex).
 show-proxies          Display the list of proxies.
```

```
num-updates             Display the number of updates stored in update
                        cache!
num-prefixes            Display the number of prefixes stored
                        in the prefix cache!
num-proxies             Display the number of proxies attached
command-queue           Displays the content of the command queue.
dump-pcache             Dump the prefix cache to command line
                        of SRx!
dump-ucache             Dump the update cache to command line
                        of SRx!
!! [<parameter>]        Repeat last command with optional new
                        parameter if specified, otherwise
                        old parameter!
```

A.2.3: RPKI_RTR_SVR – The RPKI Validation Cache Test Harness

The development of SRx-Server required to also develop test tools for proper protocol implementation. This is especially important for the RPKI side of the implementation. At the time the initial implementation started not many, if any, RPKI validation caches were around. The RPKI Prefix Origin Validation was still in its design stage, but to be able to implement a reference implementation in parallel to the protocol specification it became clear very early in the development that proper test tools are needed. The test tool required needed to abstract the full complexity of RPKI. The RPKI and its inner workings were not important for the SRx-Server development. However, the outcome of the RPKI Validation was. And for that, we needed a tool that allowed generating RPKI traffic as specified in the router to cache protocol, which became RFC 6810 for ROA information distribution and the later replacement RFC 8210 that added BGPsec key distribution. The outcome was the RPKI-RTR Cache Test Harness. This tool grew over time to an RPKI traffic generator that allows using a scripting language to automate RPKI traffic.

The following sections show screen captures of the software, which implements RFC 6810 as well as RFC 8210 and allows testing the protocol mechanics as well as sending ROA or key announcements and withdrawals and starting prewritten scripts.

The RPKI-RTR Cache Test Harness implements the protocols RFC 6810 and RFC 8210 and was successfully tested with CISCO and Juniper commercial router implementations.

The following command line parameters are provided:

```
Syntax: rpkirtr_svr [options] [port [script]]
  options:
    -f <script>  A script that has to be executed as soon as
                 the server is started.
    -D <level>   Set the logging level ERROR(3) to DEBUG(7)

  For backward compatibility, a script also can be added after a
  port is specified.! - For future usage, use -f <script> to
  specify a script!
  If No port is specified the default port 323 is used.

RPKI Cache Test Harness Version 0.5.1.1
```

Below is the list of CLI commands that can be used. All these commands can be used in a script file that can be executed from within the CLI or passed to the Test Harness using the parameter ***-f <script>***.

```
$ rpkirtr_svr 50000
Start RPKI Cache Test Harness using port 50000
RPKI Cache Test Harness Version 0.5.1.1
>> help
RPKI Cache Test Harness Version 0.5.1.1

Display Commands:
-----------------
  - verbose
                Turns verbose output on or off
  - cache
                Lists the current cache's content
  - version
                Displays the version of this tool!
  - sessionID
                Display the current session id
  - help [command]
                Display this screen or detailed help for the
                given command!
  - credits
                Display credits information!

Cache Commands:
-----------------
  - keyLoc <location>
                The key volt location.
  - empty
                Empties the cache
  - sessionID <number>
                Generates a new session id.
  - append <filename>
                Appends a prefix file's content to the cache
  - add <prefix> <maxlen> <as>
                Manually add a whitelist entry
  - addNow <prefix> <maxlen> <as>
                Manually add a whitelist entry without any
                delay!
  - addKey <as> <cert file>
                Manually add a RPKI Router Certificate
  - remove <index> [end-index]
                Remove one or more cache entries
  - removeNow <index> [end-index]
                Remove one or more cache entries without any
                delay!
  - error <code> <pdu|-> <message|->
                Issues an error report. The pdu contains all
                real fields comma separated.
  - notify
                Send a SERIAL NOTIFY to all clients.
  - reset
                Send a CACHE RESET to all clients.
  - echo [text]
```

```
                Print the given text on the console window.
  - waitFor <client-IP>
                Wait until the client with the given IP connects.
                This function times out after 60 seconds.
  - pause [prompt]
                Wait until any key is pressed. This is mainly
                for scripting scenarios. If no prompt is used,
                the default prompt will be applied!

Program Commands:
-----------------
  - quit, exit, \q
                Quits the loop and terminates the server.
                This command is allowed within scripts but only
                as the very last command otherwise, it will be
                ignored!
  - clients
                Lists all clients
  - run <filename>
                Executes a file line-by-line
  - sleep <seconds>
                Pauses execution


>>
```

In addition, the CLI allows code completion. This is achieved by pressing the TAB key. The first example below demonstrate TAB without any other input. This shows all supported commands.

```
>>
*           addNow     credits    exit       pause      reset      verbose
add         append     echo       help       quit       run        version
addKey      cache      empty      keyLoc     remove     sessionID  waitFor
addKeyNow   clients    error      notify     removeNow  sleep
```

The second example demonstrated code completion.

```
>> a
add         addKey     addKeyNow  addNow     append
```

The CLI allows to switch the TAB command from code completion to file browser by pressing "*****". This is helpful for starting scripts or specifying the *keyLoc* position

```
>> RPKI/raw-keys-ski/
18494DAA1B2DFD80636AE943D9DC9FF42C1AF9D9.cert
8E232FCCAB9905C3D4802E27CC0576E6BFFDED64.cert
18494DAA1B2DFD80636AE943D9DC9FF42C1AF9D9.der
8E232FCCAB9905C3D4802E27CC0576E6BFFDED64.der
18494DAA1B2DFD80636AE943D9DC9FF42C1AF9D9.pem
8E232FCCAB9905C3D4802E27CC0576E6BFFDED64.pem
>>
```

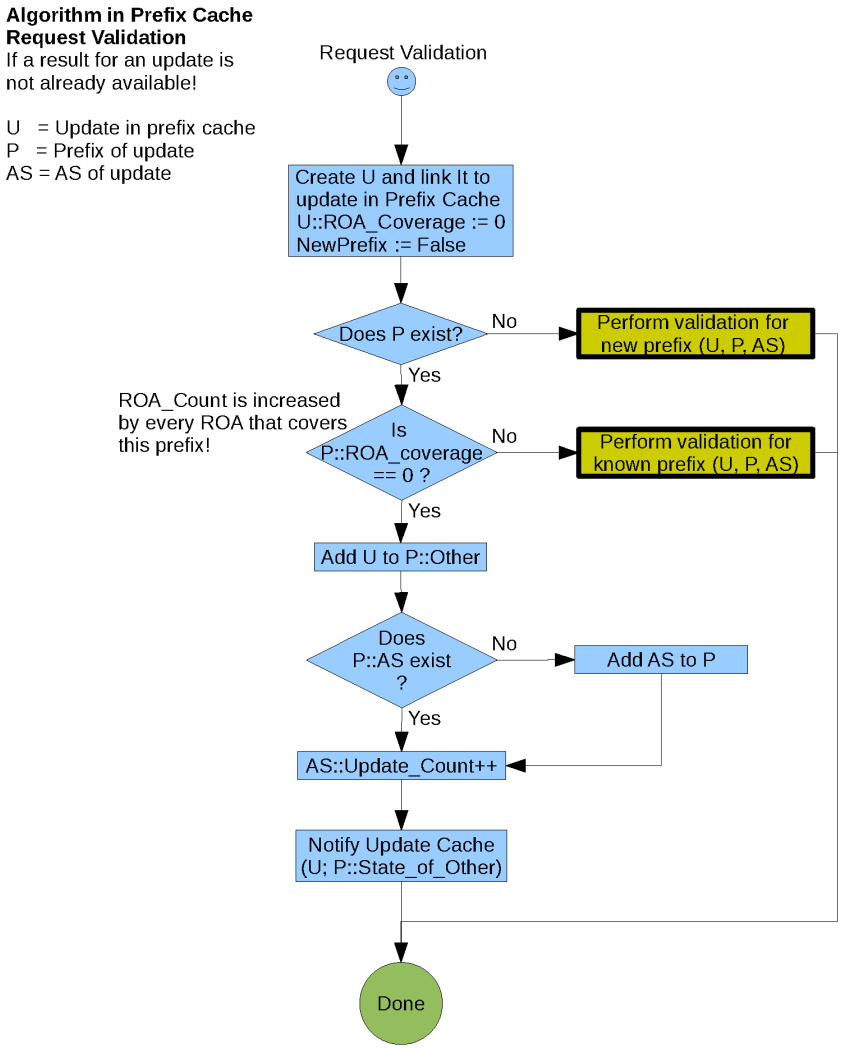
The below script shows the content of an example that pre-fills the Cache Test Harness with ROA and BGPsec key entries. A script also can have interactive and automated components. These components are *sleep <seconds>*, *waitFor <clientIP>*, and *pause [prompt]*. The command waitFor allows to wait for a client to connect, but it will time out and continue operation after 60 seconds. The *pause* command allows for interactive input. This is useful in cases where we are interested in having the capability to stop in between to analyze the router implementation.

```
echo Set keyLoc to /.raw-keys/ski
keyLoc ./raw-keys-ski
echo Load public keys...
addKey 2 47F23BF1AB2F8A9D26864EBBD8DF2711C74406EC.cert
addKey 5 3A7C104909B37C7177DF8F29C800C7C8E2B8101E.cert
addKey 7 8BE8CA6579F8274AF28B7C8CF91AB8943AA8A260.cert
addKey 50 FB5AA52E519D8F49A3FB9D85D495226A3014F627.cert
addKey 60 FDFEE7854889F25BF6ECB88AFAF39CE0EBC41E08.cert
addKey 70 C38D869FF91E6307F1E0ABA99F3DA7D35A106E7F.cert
addKey 80 18494DAA1B2DFD80636AE943D9DC9FF42C1AF9D9.cert
addKey 90 63729E346F7D10E3D037BCF365F9D19E074884E6.cert
echo done.
echo Load ROA's...
add 10.60.0.0/16 20 60
add 10.70.0.0/16 20 70
add 10.80.0.0/16 20 80
add 10.90.0.0/16 20 90
add 172.16.2.0/24 24 2
add 172.16.5.0/24 24 5
add 172.16.7.0/24 24 7
echo done.
echo Show cache content:
cache
echo
echo Send notification to clients to speed things up
notify
```

A.2.4: Prefix Cache Algorithmic Flow Charts

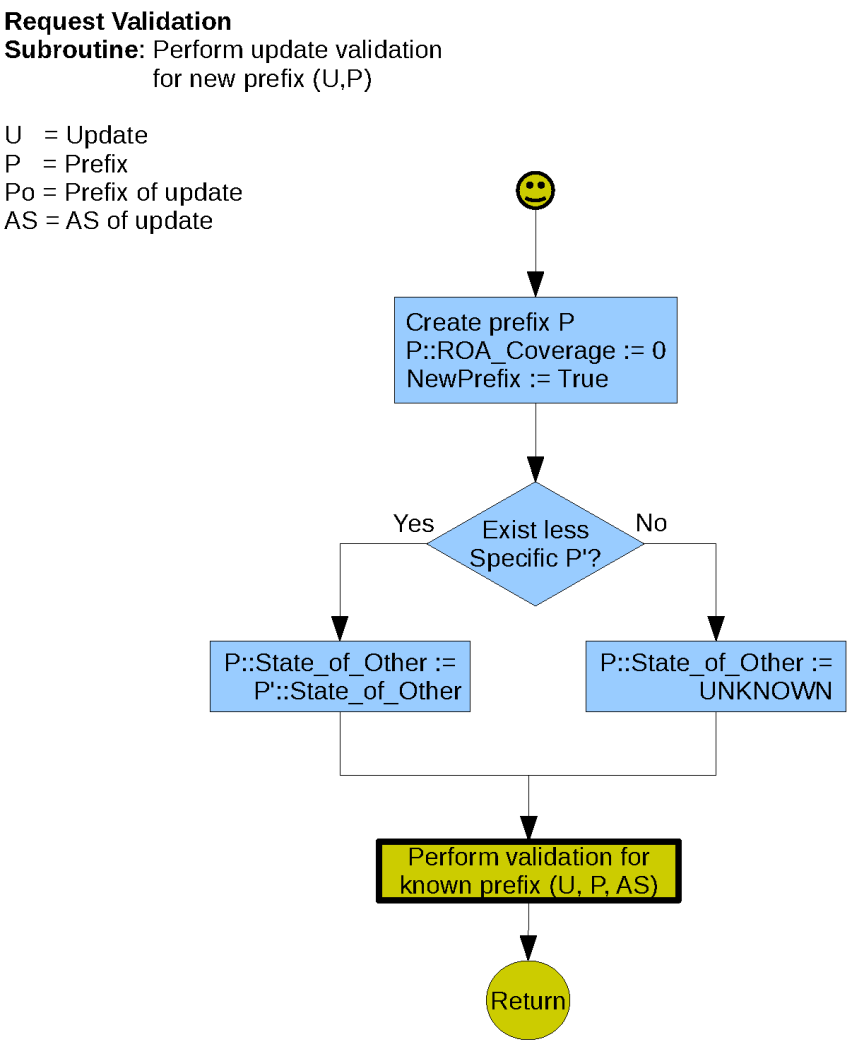
This appendix contains the list of flow charts for the internal workings of the SRx-Server Prefix Cache. The data structure allows optimized search and validation/revalidation. The flow charts are drawn so that the “smiley” on top is the start. A “blue smiley” represents a main function start, a “yellow-greenish smiley” represents a subroutine. Diamonds represent conditions and rectangles represent an action. Rectangles encapsulating other structs do represent loops. Rectangles with yellow-greenish background represent a “Goto” link to another flow chart. Yellow-greenish circles with the wording “Return” represent the return to the Goto that was called to link to this subroutine. Green circles with “Done” represent the end of the function.

Flow Chart 1 describes the request for an Update that is not yet validated. This Flow Chart (FC) will use two subroutines (SR) depending on the existence of the Prefix in the Update or if it exists, if it is already covered by a ROA.



Flow Chart 1 – Initial request validation

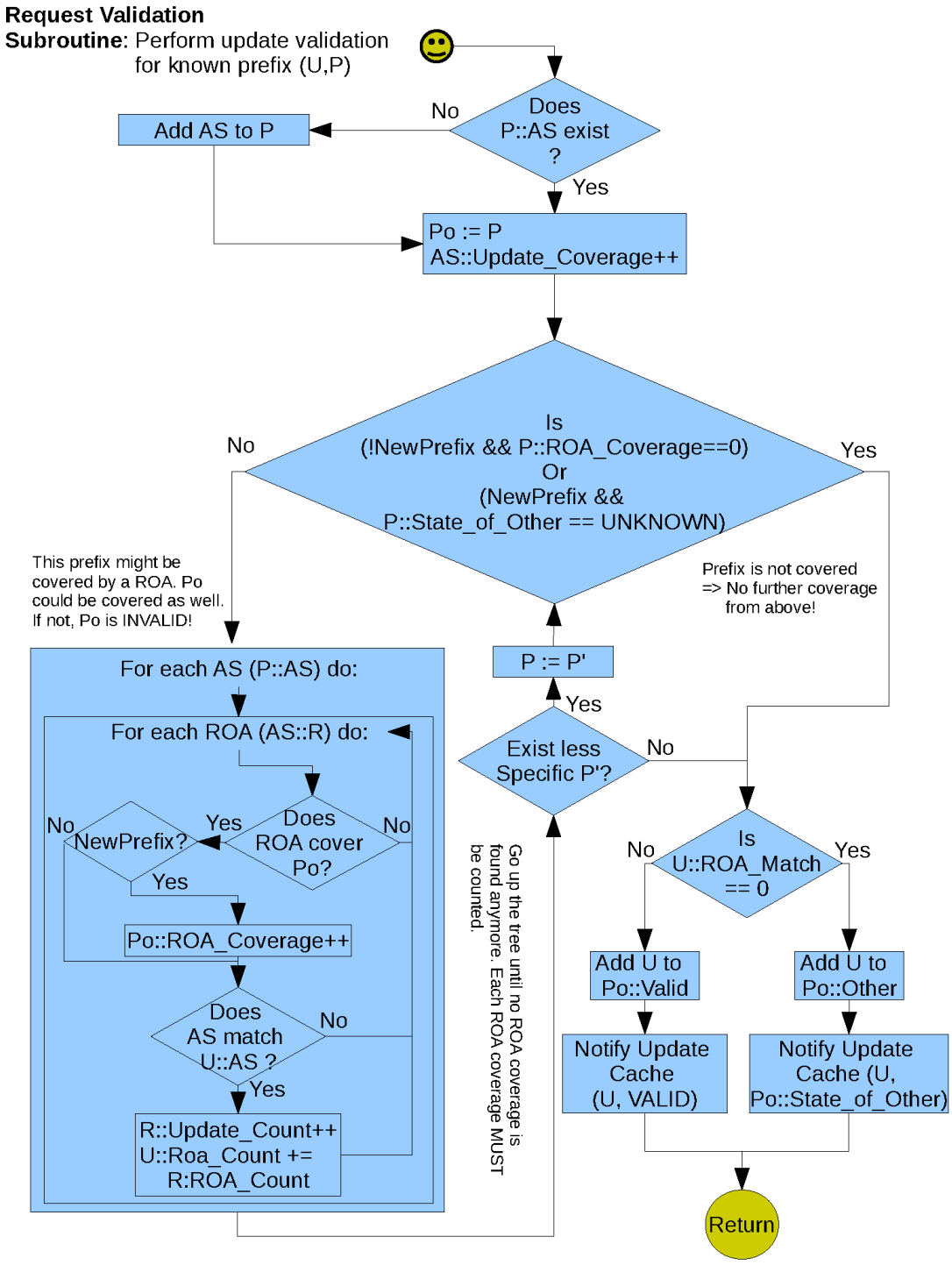
Flow Chart 2 is the SR called during the initial Validation Request once it was identified that the Prefix of the given Update is not known yet.



Flow Chart 2 – Request validation – SR: Perform validation for new prefix

Once the process depicted in Flow Chart 2 added the prefix to the Prefix Cache and initialized the attributes the SR Validation for known prefixes is called. Once the SR validation returns, this SR will return to its caller as well.

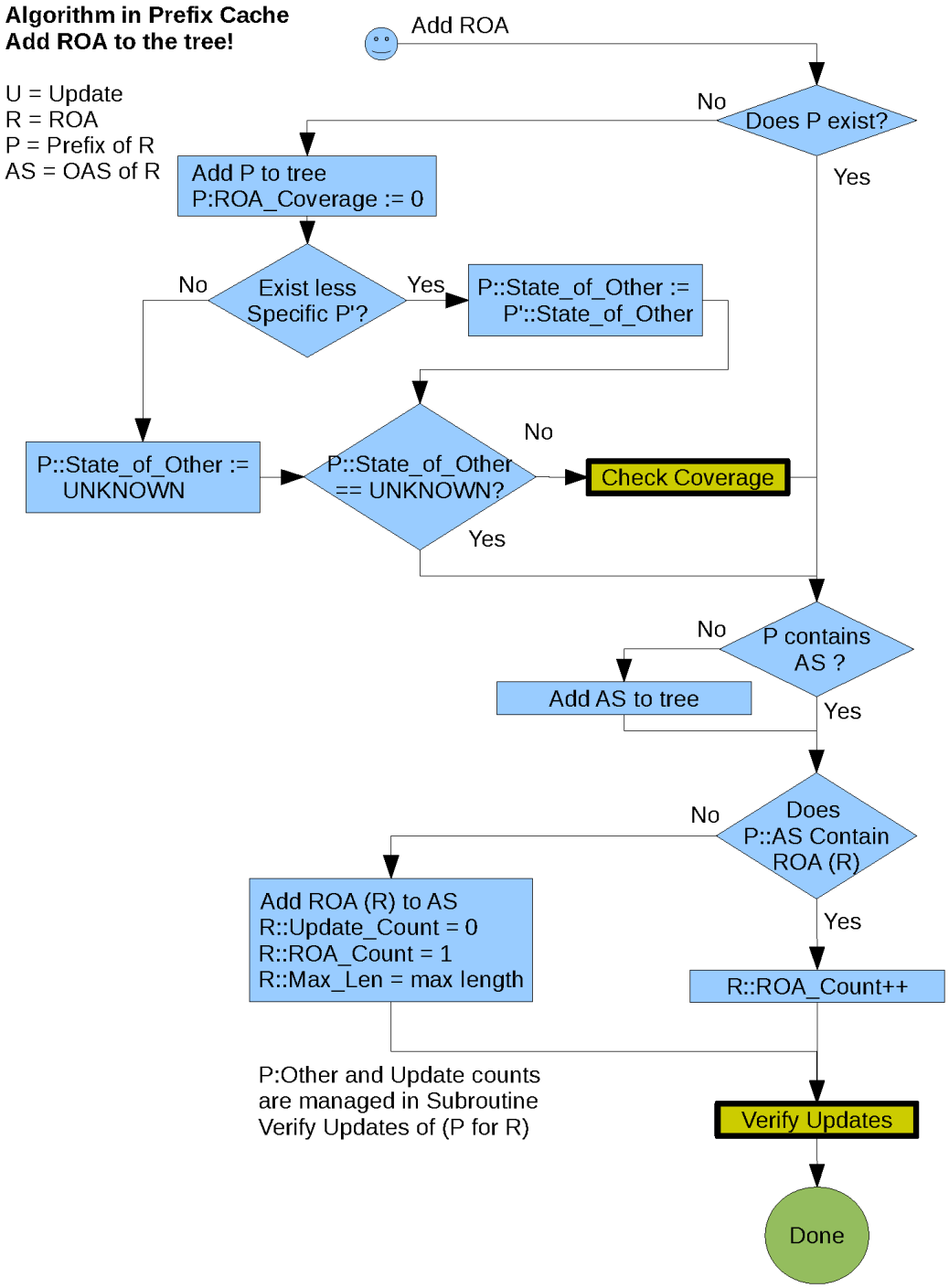
Flow Chart 3 shows the SR that is called for validation requests for known prefixes.



Flow Chart 3 – Request validation – SR: Perform validation for known prefix

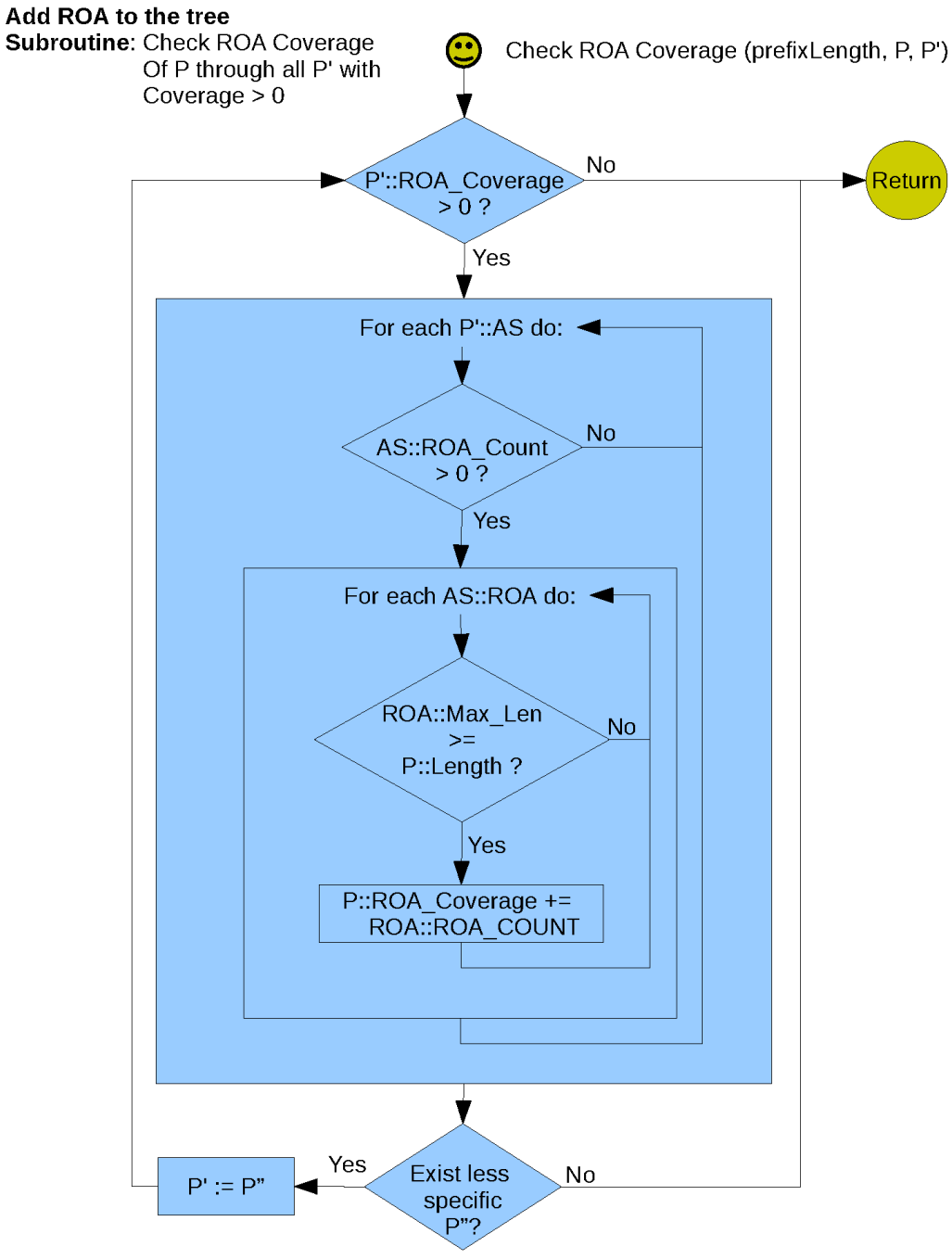
This SR determines the final validation state of the prefix. Once this is done, the subroutine returns to the caller.

Flow Chart 4 describes the mechanism of adding ROAs to the Prefix Cache. Similarly, to Update validations, the first step is to identify if the prefix exists. Once the prefix is added, the subroutine for validation is called to assure all affected updates will be re-validated.



Flow Chart 4 – Add ROA to Prefix Cache

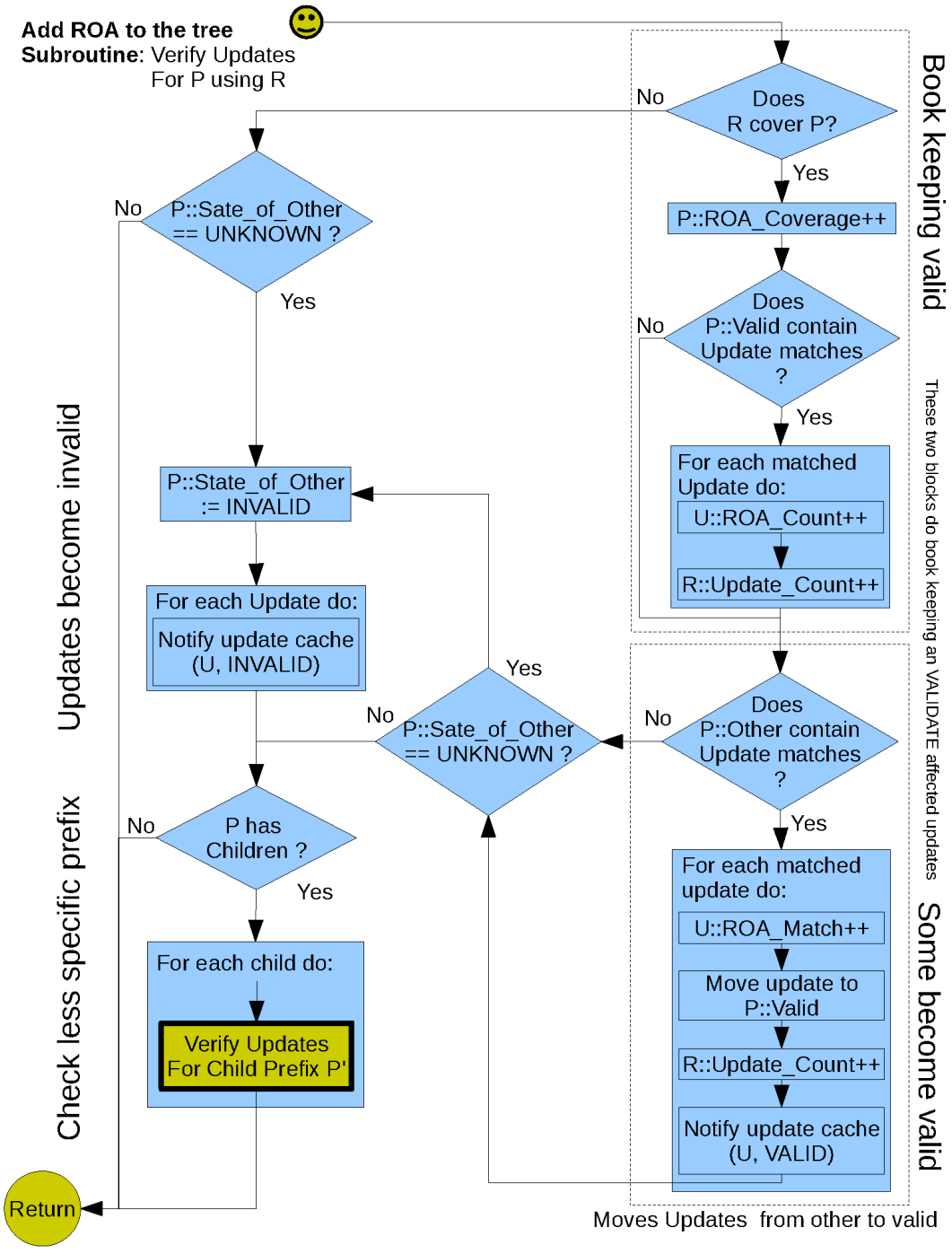
Flow Chart 5 depicts the recursive SR for checking if the prefix P is covered by a ROA installed in the less specific Prefix P' or its lesser specific prefix P" until the lesser specific prefix P" is not covered.



Flow Chart 5 – Add ROA – SR: Check ROA coverage

This method walks up the prefix tree until the first prefix is found that is not covered anymore or the root is reached.

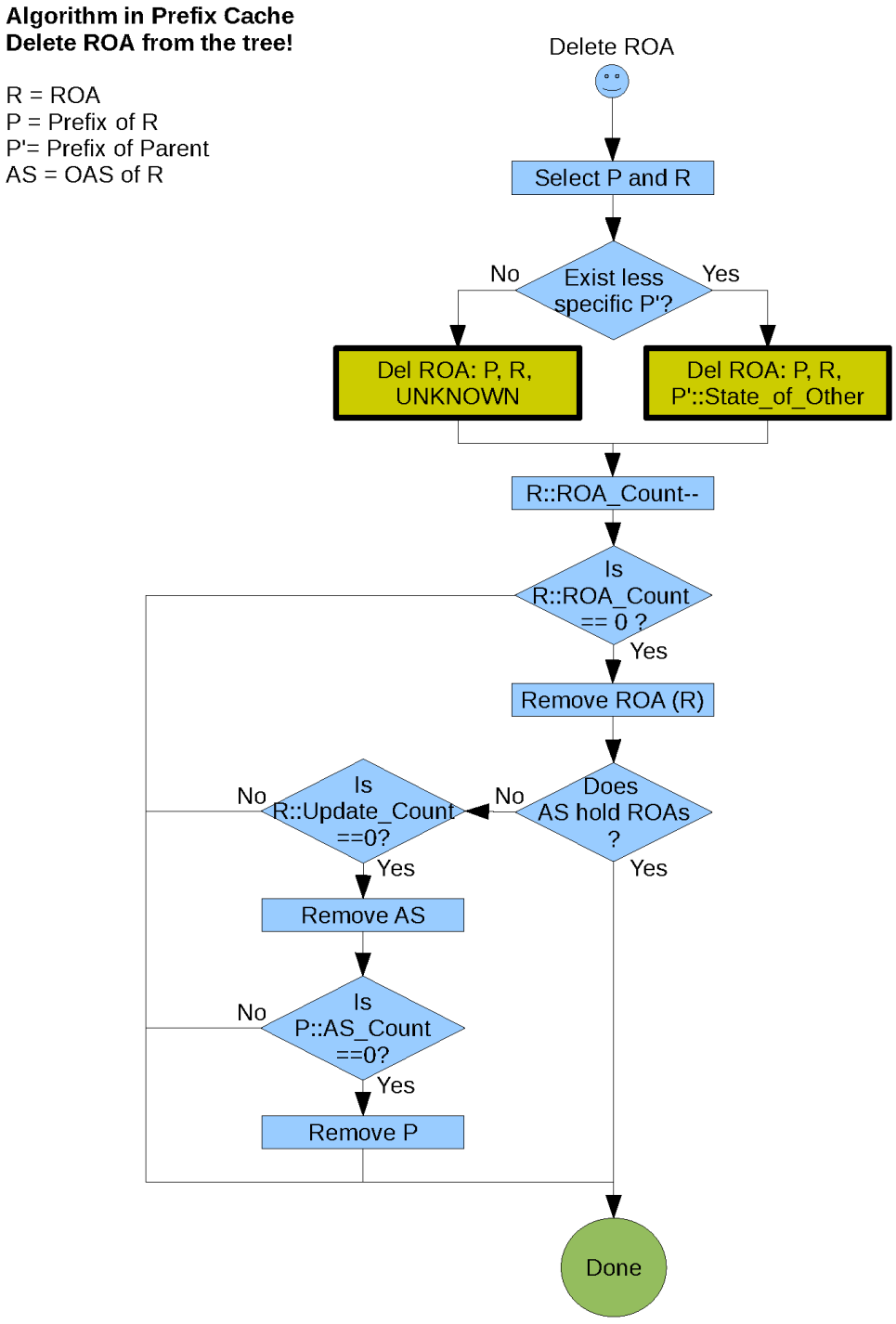
Flow Chart 6 displays the SR verifying updates for a given Prefix and ROA. This flow describes how the updates are identified going towards the more specific prefixes.



Flow Chart 6 – Add ROA – SR: Verify Updates for P using R

Each Update's ROA Match counter that is matched by the ROA prefix and Max length does get the match counter increased. This helps later to identify if an update during removal of a ROA will be moved into the *Other* list.

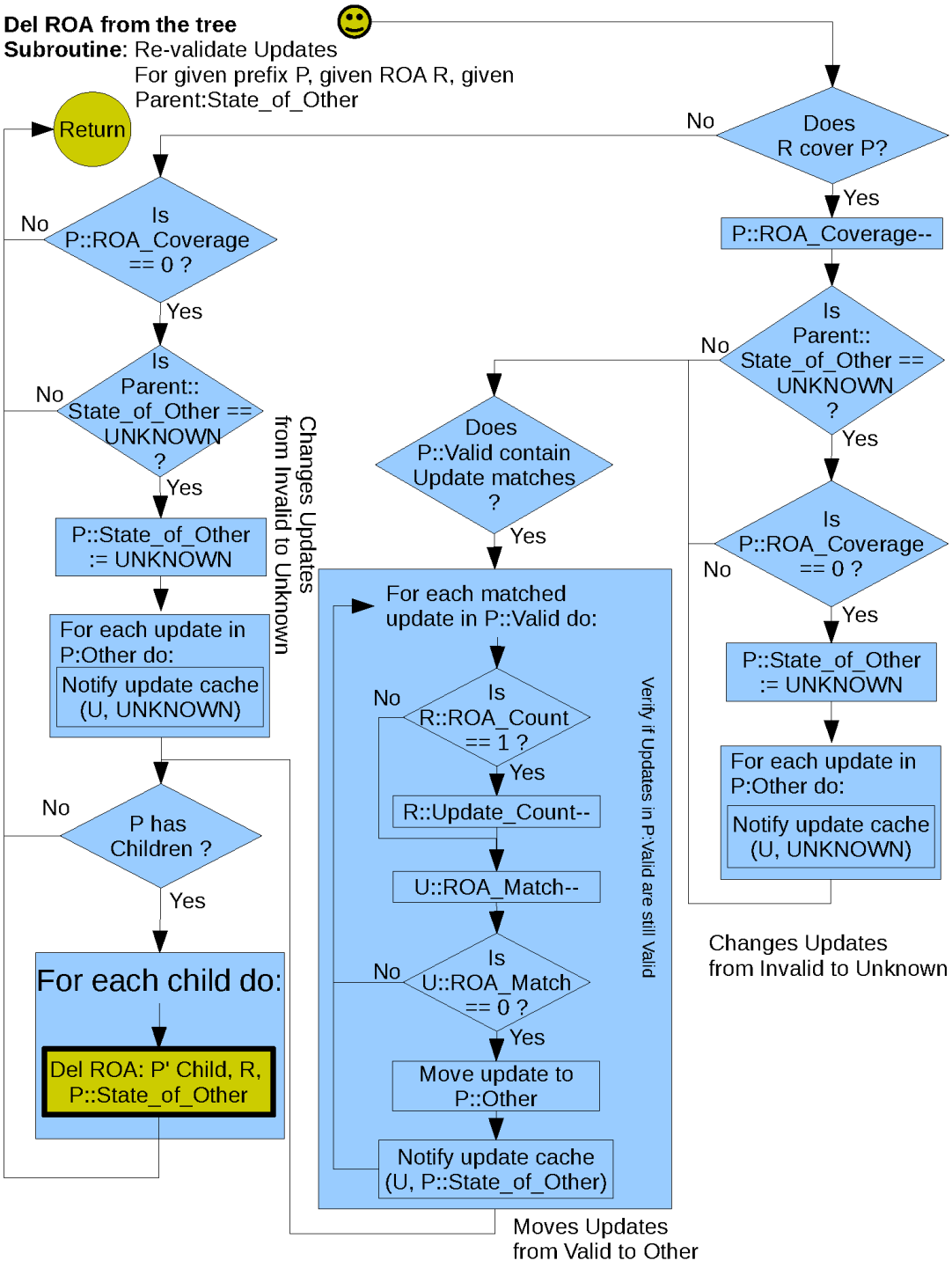
Flow Chart 7 describes the action of removing a ROA from the Prefix Cache.



Flow Chart 7 – Delete ROA

This function not only removed the ROA but also cleans the Prefix Cache. It checks if the AS used by this ROA still is attached to other ASes or updates, if not, the AS gets removed as well.

Flow Chart 8 describes the process of re-validations. It is called by Delete ROA described in Flow Chart 7. One very important fact to remember is that if one of the validation states of "State_of_Other" is INVALID, all State_of_Other values for child prefixes (more specific prefixes) must be INVALID as well.



Flow Chart 8 – Delete ROA – SR: Revalidate Updates

A.3: BGPsec-IO Configuration Examples and Tools

BGPsec-IO provides the functionality to completely generate a configuration file by using the command **bgpsecio -C <filename>** that is completely self-explanatory. The reason for that is rooted in the complexity and feature richness of BIO, which allows its user to immediately see all available options.

Next to the configuration file (which will be specified using the parameter **-f <filename>)**, BIO provides a rich set of command-line parameters. With the exception of UPDATEs, all command line parameters are applied after the configuration file is loaded. UPDATEs specified using the command line are the first ones added.

A.3.1: BGPsec-IO Configuration File

The following listing represents the auto-generated configuration file of bgpsec-io.

```
#BGPSEC-IO Configuration file. Auto generated by bgpsecio V0.2.1.1

ski_file    = "/var/lib/key-volt/ski-list.txt";
ski_key_loc = "/var/lib/key-volt/";

preload_eckey = false;

# Choose from the following types "BGP", "CAPI", "GEN-B", and "GEN-C"
mode = "BGP";

# Maximum combined number of updates to process. Script 0 for MAX INT
max = 0;

# Allow to force the usage of the flag for extended length being set.
only_extended_length = true;

# bin = "<binary input file>";
# out = "<binary output file>";
# Append data to the out file.
appendOut = "false";

# Allow to specify a configuration file for srx-crypto-api, if this is not
# specified, the default srx-crypto-api configuration (determined by the
# API) will be used.
#capi_cfg = "<configuration file>";

# Multiple sessions possible (at a later time)
session = (
  {
    asn        = 64;
    bgp_ident  = "10.0.1.64";
    hold_timer = 180;

    # Allows to specify specific session IP.
    # If not specified, the bgp_ident value is used!
    #local_addr = "10.0.1.64";


    # Allows to specify next hop address. If not,
    # specified the bgp identifier is used instead!
```

```
#next_hop_ipv4 = "10.0.1.64";
# Required for sending IPv6 updates.
#next_hop_ipv6 = "0:0:0:0:0:ffff:a00:140";

peer_asn   = 32;
peer_ip    = "10.0.1.32";
peer_port  = 179;

# Run forever or until the peer shuts down.
disconnect = 0;

# Enable BGP convergence measurement framework.
convergence = false;

# Allow to enable/disable extended message capability.
ext_msg_cap = true;
# Allow to enable/disable liberal behavior when
# receiving extended message capability.
ext_msg_liberal = true;
# Overwrite draft / RFC specification and force.
# sending extended message regardless if negotiated or not.
#ext_msg_force = true;

# Configure BGP capabilities.
#cap_as4 = true;

# Configure BGPSEC capabilities.
bgpsec_v4_snd = true;
bgpsec_v4_rcv = true;
bgpsec_v6_snd = true;
bgpsec_v6_rcv = true;

# Updates for this session only
# (path prefix B4 specifies BGP-4 only update!)
# <prefix>[,[[B4]? <asn>[p<repitition>]]*[ ]*[I|V|N]?]
update = (  "10.0.0.0/24"
          , "10.1.0.0/24, B4 10 20p3 30"
          , "10.0.1.0/24, 10 20p3 30"
          , "10.0.2.0/24, 10 20 40 50"
          , "10.0.3.0/24, 10 20 60 70V"
         );

# Enable/Disable adding global updates to this session.
incl_global_updates = true;

# Allow prefix packing for BGP-4 scripted updates wherever possible.
prefixPacking = false;

algo_id = 1;

# Choose from the following signature modes (CAPI|BIO|BIO-K1|BIO-K2)
signature_generation = "BIO";
```

```
    #In case the signature generation does fail, the
    #following settings are possible (DROP| FAKE| BGP4)
    null_signature_mode = "FAKE";
    fake_signature      = "1BADBEEFDEADFEED" "2BADBEEFDEADFEED"
                          "3BADBEEFDEADFEED" "4BADBEEFDEADFEED"
                          "5BADBEEFDEADFEED" "6BADBEEFDEADFEED"
                          "7BADBEEFDEADFEED" "8BADBEEFDEADFEED"
                          "ABADBEEFFACE";
    fake_ski            = "0102030405060708" "090A0B0C0D0E0F10"
                          "11121314";

    # Allow printout of send and received BGP/BGPsec traffic.
    printOnSend    = false;
    # Or more detailed as a filter
    #printOnSend = {
    #  open         = true;
    #  update       = true;
    #  keepalive    = true;
    #  notification = true;
    #  unknown      = true;
    #};

    printOnReceive    = false;
    # Or more detailed as a filter
    #printOnReceive = {
    #  open         = true;
    #  update       = true;
    #  keepalive    = true;
    #  notification = true;
    #  unknown      = true;
    #};

    #printSimple     = false;

    printPollLoop  = false;

    # For CAPI Mode.
    printOnInvalid = false;

  }
# Currently multi sessions are not supported, that is
# the reason the following section is commented out!
#  ,{
      # Here script another session
      # Minimum configuration
      # asn = 64;
      # bgp_ident = 10.0.1.64;
      # peer_asn = 32;
      # peer_ip = 10.0.1.32;
#  }

);

# global updates for all sessions
# <prefix>[,[B4]?[ <asn>[p<repetition>]]*[ ]*[I|V|N]?]
update = (
        );
```

A.3.2: BGPsec-IO Command Line Parameters

The following listing presents all available parameters that are available to this time. The latest list can be generated by calling **bgpsecio -?**

```
Syntax: bgpsecio [parameters]
This program allows receiving updates via pipe stream, one update per
line.

 Parameters:
 ===========
  -?, -h, -H, --help
          This screen!
  -V, --version
          Display the version number.
  -f <config>, --config <config>
          config: The configuration file.
  -y <config>, --capi_cfg <config>
          config: An alternative SRxCryptoAPI configuration file.
  -u <prefix, path>, --update <prefix, path>
          prefix: Prefix to be announced.
          path: The list of AS numbers (right most is origin).
                The path can contain pCount values using <asn>p<value>
                  to create p repetitions of asn.
                In case the path contains the value 'I', 'V', or 'N'
                  an extended community string will be added with
                  the RPKI validation state I:invalid, V:valid, or
                  N:not-found (no difference between iBGP or eBGP
                To define BGP-4 only path, start path with B4 for BGP-4!
  -s <filename>, --ski_file <filename>
          Name of the SKI file generated by qsrx-publish
  -l <directory>, --ski_key_loc <directory>
          Specify the location where the keys and certificates are
          located.
  -m <type>, --mode <type>
          Enable the operational mode:
          type BGP: run BGP player
          type CAPI: run as SRxCryptoAPI tester.
          type GEN: Generate the binary data.
  -a <asn>, --asn <asn>
          Specify the own AS number.
  -i <IPv4>, --bgp_ident <IPv4>
          The BGP identifier of the BGP daemon.
  -t <time>, --hold_timer <time>
          The hold timer in seconds (0 or >=3).
  -A <asn>, --peer_asn <asn>
          The peer as number.
  -I <IPv4>, --peer_ip <IPv4>
          The IP address of the peer.
  -P <port>, --peer_port <port>
          The port number of the peer.
  -M, --no_mpnlri
          DEPRECATED.
          Disable MPNLRI encoding for IPv4 addresses.
          If disabled prefixes are encoded as NLRI only.
```

```
-e, --no_ext_msg_cap
        Disable the usage of messages larger than 4096 bytes.
        This includes the capability exchange.(Default enabled)
-L, --no_ext_msg_liberal
        Reject extended messages if not properly negotiated.
--ext_msg_force
        Force sending extended messages regardless if capability
        is negotiated. Allows debugging the peer.
-d <time>, --disconnect <time>
        The minimum time in seconds the session stays up after
        the last update was sent. The real disconnect time is
        somewhere between <time> and <holdTime> / 3.
        A time of 0 "zero" disables the automatic disconnect.
-T, --convergence
        Enable BGP convergence statistics to be displayed for
        updates received.
-E, --no_preload_eckey
        Disable pre-computation of EC_KEY structure during
        loading of the private and public keys.
-b <filename>, --bin <filename>
        The filename containing the binary input data. Here
        only the first configured session will be used.
-o <filename>, --out <filename>
        The filename where to write the output to - Here only
        the first configured session will be used.
        Requires GEN mode!!
-O, --appendOut
        If specified, the generated data will be appended to
        given outfile. In case the outfile does not exist, a
        new one will be generated.
        Requires GEN mode!!
-U, --max
        Allows to restrict the number of updates generated.
-C <filename>
        Generate a configuration file. The configuration file
        uses the given setup (parameters, configuration file)
        or generates a sample file if no configuration is
        specified.
-n <interface>
        Use the interface to determine the local IP address.
        This setting is only used in combination with the
        creation of a configuration file.
```

```
Configuration file only parameters:
===================================
 incl_global_updates
         Enable/Disable adding global updates to this session.
         Default: true
 cap_as4
         Enable/Disable the usage of 4-byte ASN.
         Default: true (enable)
 bgpsec_v4_rcv
         Specify if bgpsec-io can receive IPv4 BGPSEC traffic.
         Default: true
 bgpsec_v4_snd
         Specify if bgpsec-io can send IPv4 BGPSEC traffic.
         Default: true
 bgpsec_v6_rcv
         Specify if bgpsec-io can receive IPv6 BGPSEC traffic.
         Default: true
 bgpsec_v6_snd
         Specify if bgpsec-io can send IPv6 BGPSEC traffic.
         Default: false
 local_addr
         Specify the IP address used for this session. In case
         no local IP is specified the BGP identifier is
         used.
 signature_generation
         Specify the signature generation mode:
         mode CAPI: Use CAPI to sign the updates.
         mode BIO: Use internal signature algorithm (default).
         mode BIO-K1: Same as BIO except it uses a static k.
         mode BIO-K2: Same as BIO except it uses a static k.
         The signature modes BIO-K1 and BIO-K2 both use a k
         which is specified in RFC6979 Section A.2.5
         BIO-K1 uses k for SHA256 and msg=sample.
          k=A6E3C57DD01ABE90086538398355DD4C3B17AA873382B0F24D6129493D8AAD60
         BIO-K2 uses k for SHA256 and msg=test.
          k=D16B6AE827F17175E040871A1C7EC3500192C4C92677336EC2537ACAEE0008E0
 only_extended_length
         Force usage of extended length also for BGPSEC
         path attributes with a length of less than 255 bytes.
 null_signature_mode
         Specify what to do in case no signature can be
         generated. Example: no key information is found.
         Valid values are (DROP|FAKE|BGP4).
 fake_signature
         This string contains the fake signature in hex format.
         The signature must not be longer than 255 bytes.
         (2 HEX characters equals one byte!).
 fake_ski
         This string contains the fake ski for not found keys.
         The SKI MUST consist of 20 bytes.
         (2 HEX characters equals one byte!).
```

```
printOnSend, printOnReceive
        Each BGP update packet send/received will be printed on
        standard output in Wireshark form.
        Use this setting for debug only!!
        Both settings can be used in two different forms:
        (1) Set =true|false to this for all message types.
        (2) Use as sub configuration to fine-tune each message.
            Using this form sets all message types to false and
            they must be individually set to true.
            = { msg-type = true|false; ... };
            Valid message types are:
            open
                    Printing of bgp OPEN messages.
            update
                    Printing of bgp UPDATE messages.
            keepalive
                    Printing of bgp KEEPALIVE messages.
            notification
                    Printing of bgp NOTIFICATION messages.
            unknown
                    Printing of future bgp messages.
printSimple
        Print BGP messages in simple format (true) of in
        Wireshark format (false).
printPollLoop
        Print information each time the poll loop runs.
printOnInvalid
        Print status information on validation result invalid.
        This setting only affects the CAPI mode.
```

A.3.3: BGPsec-IO Tools

The following list shows the tools that are provided with BGPsec-IO. These tools are Linux shell scripts that help to generate and retrieve data.

- **bio-traffic.sh**
  This script shows how to generate large numbers of UPDATEs. It can generate two different types of traffic data:

  - For the configuration file as configuration UPDATE list. This can be easily used by removing the global Update section from the configuration file and then redirecting the output of this tool into the configuration file. Example: sh bio-traffic.sh >> bgpsecio.conf
  - Generating a stream that can be "piped" to BGPsec-IO via the command-line interface. Each Update in its own line.

  It allows specifying the AS path to be used as well as provides a minimal amount of configuration setting for the prefixes that will be generated.

  The tool provides the following command line parameters:

```
bio-traffic.sh [-b <#> <val> ] [-p <pfxlen>] [-s2 <step>]
               [s3 <step>] [-P <path>] [-c ] [-l]

  Parameters:
    -b <#> <val>: Specify the byte portion of the start prefix
    -p <pfx-len>: Specify the prefix length (default: 24)
    -s2 <step>  : Specify the increment of byte 2
    -s3 <step>  : Specify the increment of byte 3
    -P <path>   : Path = <as#> <as1> <as2> ... path to be added
    -c          : Max number of updates
    -l          : List mode, one update per line.
    -?, ?, -h   : This screen.
```

- **get-from-quagga.sh**

  This tool allows to connect to QuaggaSRx via telnet, execute the given command and exit the telnet session. It is useful to automate experimentation and verify the RIB-IN on the QuaggaSRx end.

- **mrt_to_bio.sh**

  This script formats the output or bgpdump[9], an external tool used to convert MRT files to ASCII files. The formatted string is compatible to be "piped" into BGPsec-IO.

[9] https://github.com/RIPE-NCC/bgpdump

A.4: SRx Server Protocol

This appendix includes the protocol used to communicate between the SRx-Proxy and the SRx-Server. This protocol is currently in its 2nd iteration.

srx-server-protocol-2.1 O. Borchert
NIST-BGP SRx 5.1 K. Lee
P. Gleichmann
NIST
Experimental Sept 2020

Secure Router Extension (SRx) Server Protocol
srx-server-protocol

Abstract

This document facilitates the off-loading of RPKI-based security operations such as Route Origin Validation (ROV) and BGPsec path validation (BPV) onto external systems such as the NIST developed Secure Routing Extension (SRx) Server. It describes the communication between the SRx Server and its proxy thin client integrated, within a BGP router or Policy module. The SRx Server provides an interface to the RPKI/ROA Validation Cache using the RPKI to Router protocol [RFC8610] as well as a BGPsec path validation engine.

Status of This Memo

This document specifies a protocol design for the NIST internal reference implementation for ROA processing [RRFC6811] and BGPSEC processing [RFC8205] on the router side. This document describes an experimental protocol which is NOT an Internet standard. Comments and suggestions are welcome and to be send to the author of this document.

Table of Contents

1. Requirements language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Introduction

This document describes the communication between SRx and its proxy. This protocol is of interest to those who decide to implement their own proxy module and therefore choose to communicate directly with the server.

The srx-server-protocol is a TCP based, not encrypted protocol that is intended to be used within a trusted and secure environment, hence it is not needed to add an extra layer of security that would only increase the data volume. If security is desired, it can be tunneled through using ssh.

The SRx server itself does not provide BGPSEC validation in the sense of a combined origin and path validation. SRx provides both validation types independent from one another and therefore the validation requests within this protocol are NOT requests for ROA validation and BGPSEC validation, the requests and result notifications are focused on each component, origin validation and path validation. The consumer can decide on how to interpret / combine the results according to the implementation chosen.

3. Glossary

SRx:
: Secure Routing Extension, a framework that allows to out-source the processing of route origin validation and path validation as well as path signing.

Proxy:
: The thin client of SRx embedded in the router, policy module or other software that will use the SRx. In case the router chooses to implement the proxy itself, it will take on the role of the proxy.

Router:
: A BGP router that uses the SRx to receive validation information about updates.

Policy Module:
A software that generates BGP routing policies that are feed into the BGP router. This policy module might use the SRx to generate/modify policies. In such case the router does not need to use SRx.

Validation Cache:
The validation cache is responsible for performing ROA/RPKI validation. Changes in the validation cache are signaled to the SRx using [RFC8210]

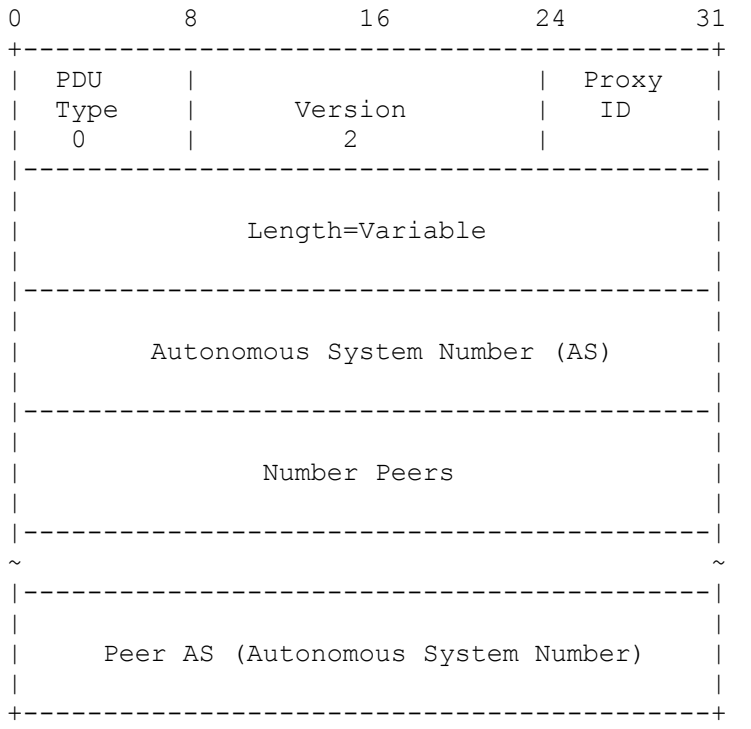
4. Protocol Data Units (PDU)

4.1. Session Packets

This chapter deals with the session handshake and session tear down.

4.1.1 Hello

The proxy connects to the client and negotiates a session. The negotiation is done by sending a hello packet to the SRx server. The server will answer with a Hello Response packet that contains the connection status and proxy identifier. The packet MUST contain at least one "Peer AS". This information allows the SRx to precalculate signatures while in IDLE mode. The precalculation will only be performed for updates received, not for updates that are originated by the proxy AS. The default prepend count of the own AS number is "1". The "Sign Request" packet allows to specify a different prepend count.



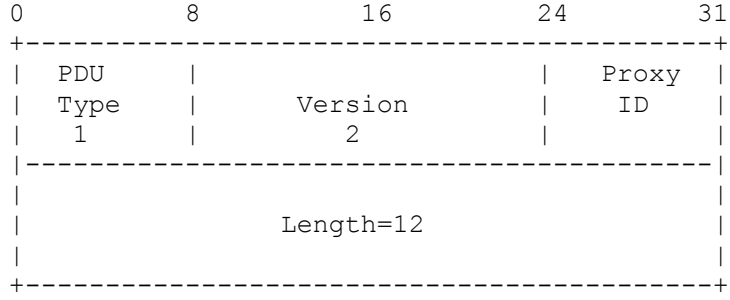
Each connection between a proxy and SRx MUST have a unique identifier. Each proxy can have only one AS number. The router implementation MUST NOT share one proxy instance with multiple internal router instances. Each router instance MUST have its own

proxy.

The SRx answers to the hello packet either with a "Hello Response" message or with an error packet. In case the provided proxy identifier has the value "0" zero, SRx will generate one and return it back using the Hello Response message. Otherwise the SRx returns the provided proxy identifier.

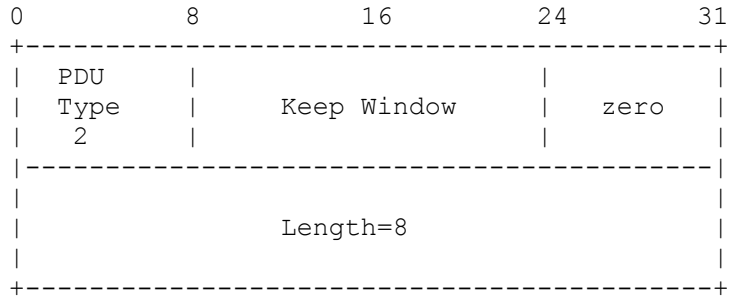
4.1.2 Hello Response

The Hello Response packet finalizes the handshake. The proxy MUST use the proxy identifier provided within this packet. This value should be the same as the provided one using the Hello packet except the initial value was set to "0" zero. In this case the SRx generated the identifier. In case of a conflict the server assigns a new value.



4.1.3 Goodbye PDU

The Goodbye message is sent to orderly disconnect the session between SRx and proxy. This packet is used by both, SRx as well as its proxy.



The field Keep Window is a request for both sides to not remove data associated to this session. This value is in seconds and is a request only.

4.2 Origin and Path Validation Request Communication

SRx provides two different services. The first one is the validation of updates received, the second one is the signing of BGP updates selected by the router and send out to its peers.

(1) Origin and Path Validation:

The Verify Request will be performed using two packet types, one for IPv4 prefixes and the other for IPv6 packages. The Verify Request messages are used to request the verification of an update.

Within each request the proxy provides SRx with a predefined validation result (Sections 5.7 & 5.8) to allow SRx to return a preliminary result as soon as SRx generated a unique ID for this update. This ID is the communication interface between SRx, the proxy and the router / Policy module.

The validation packets are used for both request types, Origin validation request as well as Path validation request. Both requests can be either combined into one single packet or send as two separate requests. Eventually all requests for the same update will result in the same update identifier.

Updates that are originated by the router do not need to be verified. Nevertheless, these updates need an update ID to be able to use the SRx signing mechanism. In this case the update MUST be processed using a validation request with the exception that none of the "Validation Type" bits is set. This will prevent the update from being further processed in regard to validation. This is considered the "Safe Only" mode.

(2) Path Signing:

The path signing request starts the signing operation within SRx. SRx is able to precompute signatures for updates that underwent BGPSEC validation. Updates that are originated by the router can not be precomputed due to the fact that one element the signature does cover is the origination time stamp. This will be taken in the moment of sending. All other updates can be precomputed by SRx during idle times.

If the precomputation of update signatures is performed it is up to the SRx implementation to decide if by default the own AS is used once or multiple times. It is recommended though to precompute without any traffic engineering. The signing request can make these requests on an individual base using the

"Prepared Counter" field of the request package.

SRx only monitors verification results only for updates for which it received at least once a validation request. This means in case a validation request was made for origin validation but not path validation, the monitoring is performed for origin validation only. In case an additional request for the same updates is received but this time for path validation, SRx will start adding path validation to the monitor. In this case it will monitor both validations.

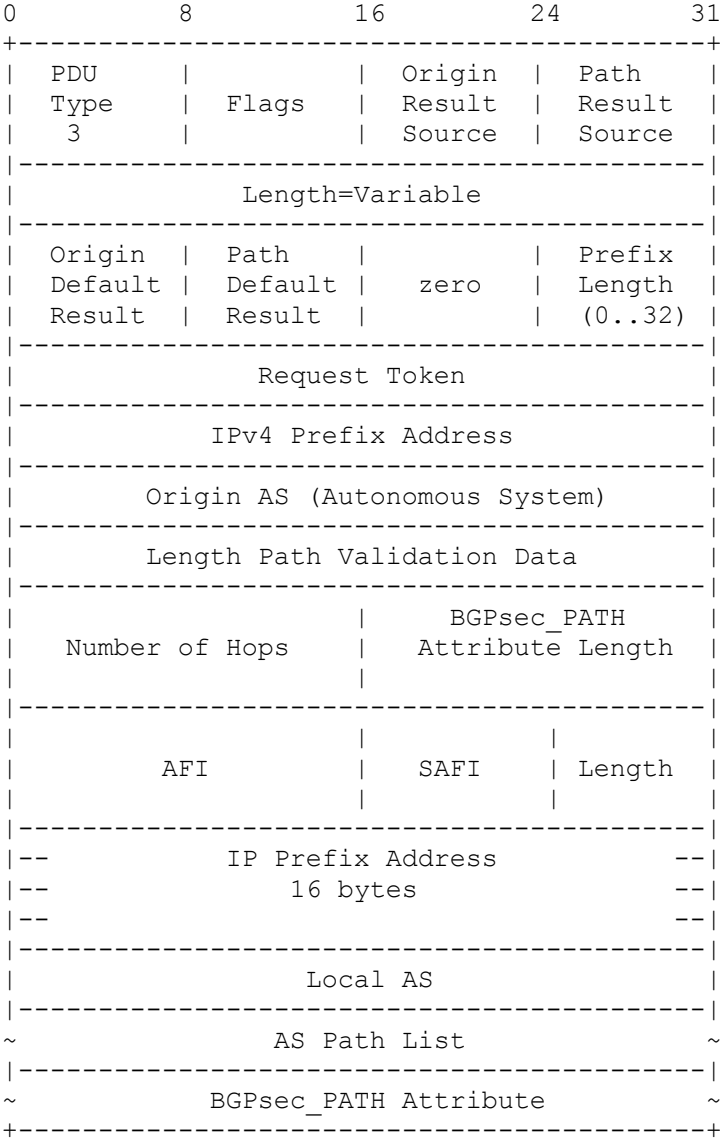
This allows to just store updates by submitting a validation request but leaving the field "Flags" empty. This allows the proxy to receive an update id that can be used for later signature requests. (Important for self-originated updates!)

Validation requests normally result in validation receipts. This is necessary to allow the SRx to return the generated update ID. Therefore, the proxy needs to wait for the receipt after each update request. This might create an unnecessary processing delay on the routers side, especially in situations where the update id is already known. In such situations, receipts can be omitted. The SRx then will only send a notification in case a change in validation result occurred. In case the proxy omits all receipts, the proxy has to generate the "update ID" on its own. In this case the proxy MUST assure to use the identical algorithm for generating the ID as the SRx otherwise they will not properly communicate to each other.

In case the validation request uses the receipt flag, the request token allows the proxy to match the request to the notification. Notifications use this flag to assign the receipt to its validation request.

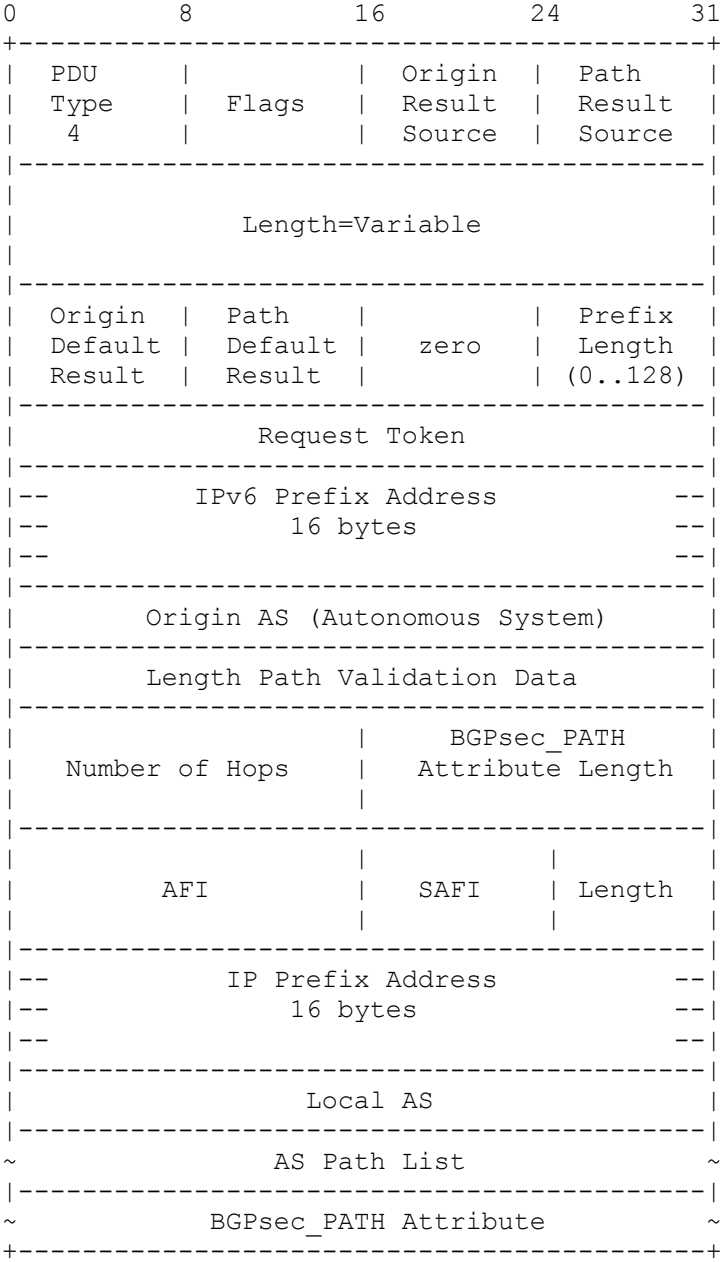
4.2.1 Verify Request IPv4

The following PDU describes the verification request packet for IPv4 Prefix / Update validation.



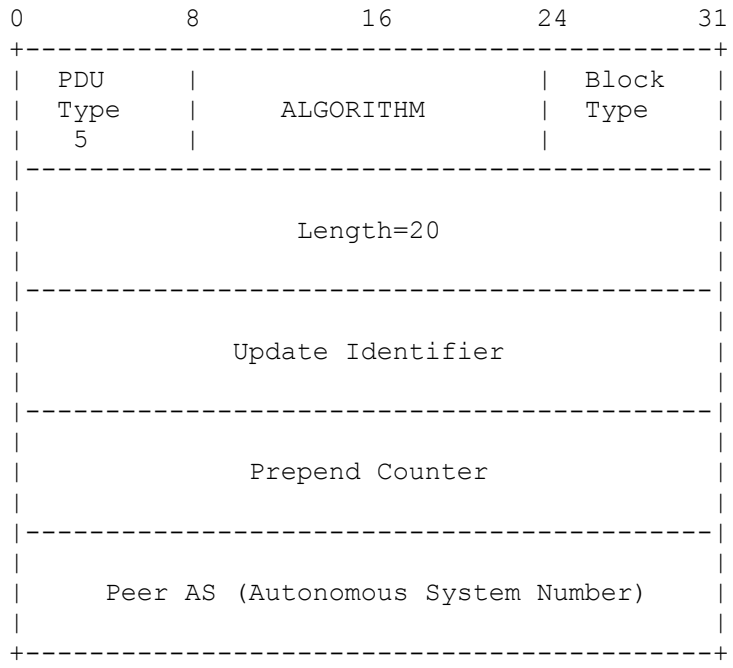
4.2.2 Verify Request IPv6

The following PDU describes the verification request packet for IPv6 Prefix / Update validation.



4.2.3 Sign Request

This PDU is used to sign the as path and its attributes. This request needs only to reference the peer AS the update will be send to. SRx is capable of precomputing default signatures; signatures with no additional content other than already provided during initial validation request. The field "Prepend Counter" allows traffic engineering in form of path-length within the signature generation.



4.3. SRx to Proxy Result Notification"

The PDU's described here are used by SRx to communicate the requested result as well as changes within the results to the proxy and therefore, to the router or policy module. Notifications can occur due to multiple events:

(1) Origin / Path Validation:

Each request for validation will result instantaneously in a result notification. For this kind of notification, the "Receipt" flag is set. This result does not only return a "preliminary/final" result, it also returns the unique update ID that is used for all future communication regarding this particular update.

Repeated validation requests of the exact same update MUST result in the same update ID.

(2) Signature Request:

Different to the validation request, the signature request will result in a signature validation. The router must decide if it only sends updates fully signed or if it allows sending unsigned packages followed by resending the update again once the signature is fully computed.

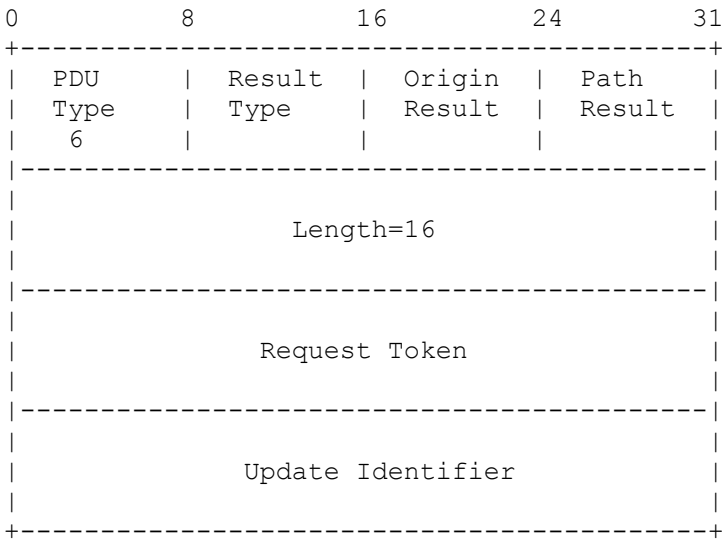
(3) Validation Changes of ROA's or Signature keys:

SRx monitors the state of updates in such as it received information about ROA expiration, revocations, key expiration etc. These events can change the validation result of a prior processed update. In case a validation state of an update changes, SRx MUST send a notification message to the proxy. This notification messages MUST NOT have the "Receipt" flag set.

4.3.1 Verify Notification

This packet is used by SRx to communicate the validation result to the proxy. The results communicated using this packed must reflect the validation result of SRx as soon as possible. This means that as soon as SRx results are available these results MUST be used, the results provided during the last request are ignored.

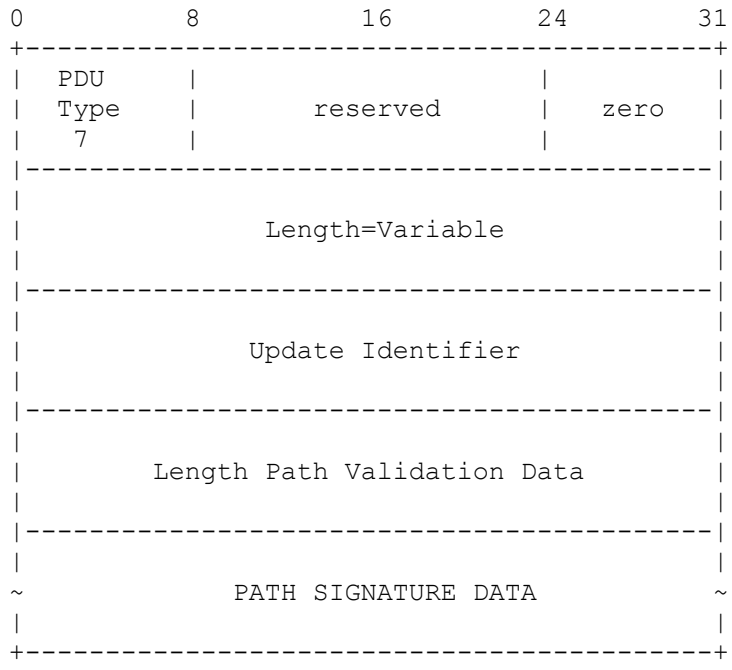
The "Receipt" flag specifies if this notification MUST be handles as validation receipt or validation result notification.



The "Request Token" field will only be used in case this notification is flagged as "Receipt". This token will help the proxy side to match the receipt to the request.

4.3.2 Signature Notification

This packet is used by SRx to communicate the BGPSEC portion of the update back to the proxy. Depending on the sign request type the data returned contains either only the latest signature block or all signature blocks. This allows the user of the proxy (e.g. router) to either strip all BGPSEC information from the update and keep the memory consumption low or keep the data traffic to a minimum by only transmitting the new signature block. In both cases SRx keeps all BGPSEC related data cached.



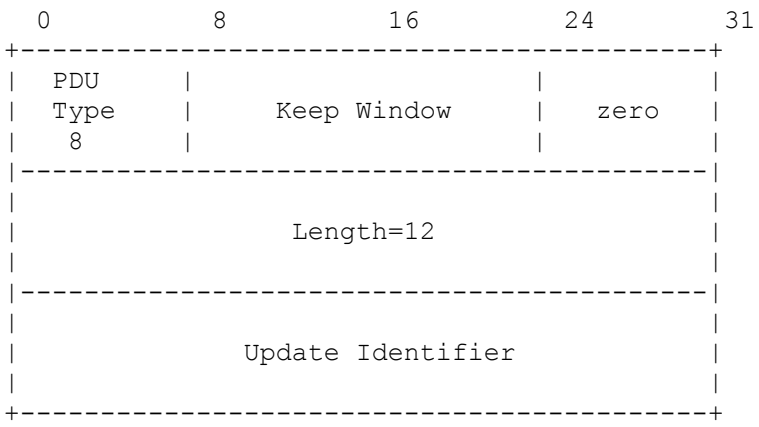
4.4 SRx Maintenance and Error Handling

PDU's specified in this section define maintenance packets. They are used to allow synchronization, failure notification, housekeeping, and peer configuration. SRx can implement the functions anticipated behind these messages but does not need to do so. Both, SRx as well as the proxy MUST accept packages of this type according to the communication schematics.

4.4.1 Delete Update

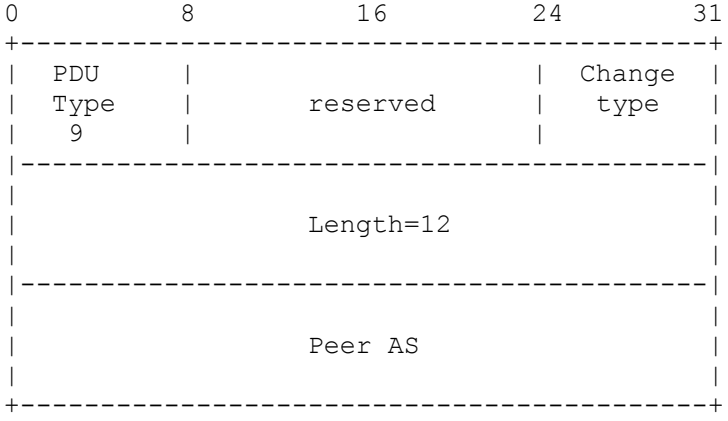
This packet is used to inform the SRx that the specified update is not available anymore in the router. SRx itself does not need to react on this but it will allow SRx to free up resources.

Furthermore, the proxy makes sure it will NOT receive any further notifications related to this update. The "Keep Window" field allows to inform SRx that the update might be requested again within the "Keep Window" time. SRx is not obligated to follow this request. This field is purely a performance setting.



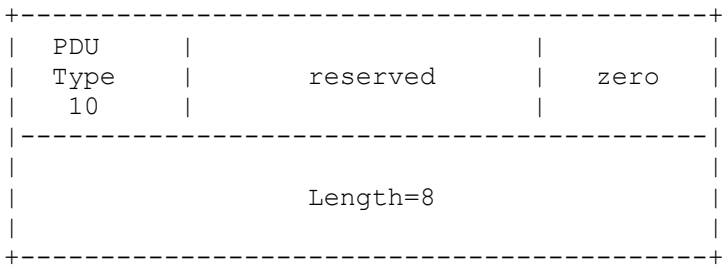
4.4.2 Peer Change

This packet is used by proxy to indicate a change in the peer configuration.



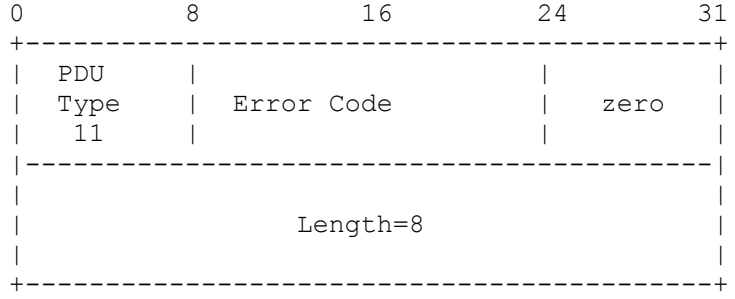
4.4.3 Synchronization Request

This packet is used if SRx assumes that the connection with the proxy is out of sync. This could be due to a session restart or other problems. It is IMPORTANT that the proxy performs a synchronization once the server requested one to assure that SRx has a complete view on the data of proxy. The synchronization is performed by sending a validation request for each update located within the RIB in of the BGP router. Due to the fact that this could be a very expensive operation the SRx implementation should be conservative in the usage of this request.



4.4.4 Error Packet

This packet is used by SRx in case an error occurred. All errors are considered fatal and are followed by a goodbye if possible.



5. PDU Data Fields

This section describes the data fields used within each PDU.

5.1. Proxy Identifier (Proxy ID)

The Proxy Identifier is a unique 4 byte value that allows the SRx to map internal values to the proxy itself. A preferred value is the IPv4 address of the proxy.

During the handshake the identifier is allowed to have the initial value of "0" zero provided by the proxy. In this case the SRx will generate a SRx wide unique identifier for this proxy. Each identifier can only be mapped to one proxy at a time. In case the proxy provides an identifier during handshake and this identifier is currently mapped to an existing session an error will be produced and the handshake fails.

5.2. Number Peer AS

The number of BGP peers the proxies user (most likely the BGP router) has. In case the value of this number is odd, the last two bytes of the packet MUST be filled with "0" zero. The number of peers MUST be greater or equals to "1" one.

5.3. Peer AS (Autonomous System)

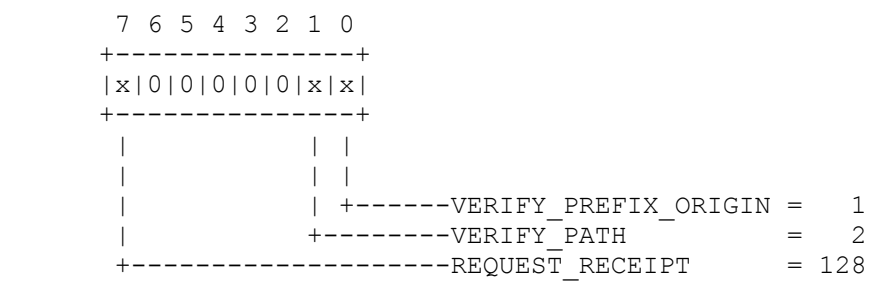
Contains the AS number of a peer. This is used to allows SRx to precompute signatures for the moment when the proxy requests a path signature for a particular selected path. This operation can be performed by SRx during idle times.

5.4. Change Type

The change type is used to indicate if the specified peer as is removed or added to router configuration. Adding a peer is done by setting the value to "1" one, removing is done by setting the value to "0" zero.

5.5. Flags

The flags field is bit coded and informs the SRx server what to do.



In case none of the bits is set the update will only be stored. This is needed for updates originated by the router that uses the proxy but do not need any validation applied to.

VERIFY_PREFIX_ORIGIN (1)

Request Prefix Origin validation. With this flag set the SRx will perform an origin validation. In case the SRx already performed this validation it is expected to return its result. No new validation needs to be performed. Changes in validation status due to expired or revoked ROA's will be triggered by the validation cache.

VERIFY_PATH (2)

Requests for patch validation. With this flag set the SRx will perform a path validation on the passed update. This setting also indicates that SRx must monitor changes within the key validity. In case a key is revoked or expired its signatures will become invalid and this change will be signaled to the router / proxy via a Validation Notification massage.

REQUEST_RECEIPT (128)

This flag indicates the proxy requests a notification receipt. A receipt is similar to a notification except that the SRx MUST send it regardless if the provided given validation state matches the validation result generated by SRx during a prior validation request. The request receipt is most importantly used to receive the unique update id. In case the proxy generates the update IS it MUST use the same algorithm as SRx to prevent communication problems.

5.6 Origin Result Source / Path Result Source

These two fields specify what to do with the provided default result values. It is possible that the BGP router might call the validation and provides a prior calculated validation result. this can be as a result of a SYNC request from the SRx server itself. It also could be used after a session reboot between SRx and router.

This field can take the following values:

0: SRX
1: ROUTER
2: IGP
3: UNKNOWN

IGNORE: Do not provide default result in case no result is available yet.

All the others: In case the SRx does not already have a validation result it will return the predefined value. In case the validation comes back with a different validation result than the provided one, SRx will notify the router of the change.

5.7. Default Origin Result

A predefined validation result that if used is returned to the proxy in case no current result is available.

The following values can be used:

0: VALID
SRx knows about a ROA that covers this pair of Prefix/Origin.

1: UNKNOWN
The prefix is not covered by any ROA nor is a ROA known that describes a less specific prefix.

2: INVALID
A ROA exists that covers either this prefix or a less specific prefix but none includes the given prefix.

3: UNDEFINED
SRx does not have any result available and no default value was provided.

5.8. Default Path Result

A predefined validation result that if used is returned to the proxy in case no current result is available.

The following values can be used:

0: VALID
SRx could validate the path according to the specifications.

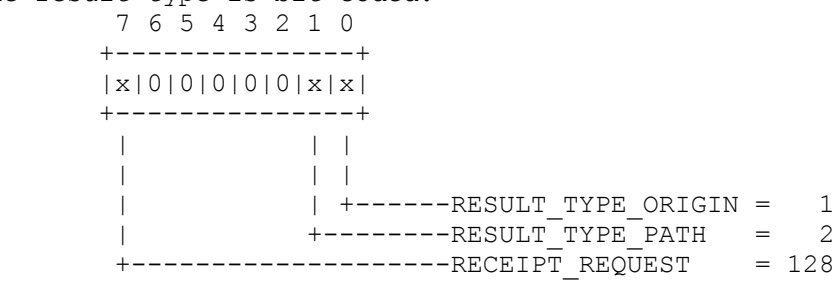
2: INVALID
SRx could not validate the path according to the specifications.

3: UNDEFINED
SRx does not have any result available and no default value was provided.

5.9. Result Type

Identifies which result to use. This field MUST NOT be "0" zero filled.

The result type is bit coded.



5.10. Update Identifier

Specifies a unique id within the router that is used to identify the update when talking between proxy and SRx. It is expected that the identical update results in the exact same Update Identifier at all times.

5.11. Receipt Token

This field is maintained by the proxy only. It is not used as system wide identifier. the receipt token helps the proxy to assign a receipt notification to the initiating validation request and only if receipts are requested. Otherwise this field will be zero. This token facilitates the timeout management of requests. The client decides on how to tread notifications that do NOT match any requests as well as how to generate this token. SRx simply copies the value from the verification request into the request notification.

5.12. Origin AS

The originator of the update. Will be ignored if origin validation is turned off (VERIFY_PREFIX_ORIGIN not set).

5.13. IPv4 Prefix / IPv6 Prefix

The IPv4 or IPv6 prefix (In Network order).

5.14. Prefix Length

The length of the IP prefix. This value can be 0..32 for IPv4 and 0..128 for IPv4. All other values are resulting in an error response.

5.15. ALGORITHM

BGPsec path validation [RFC8205] only allows two algorithms at the same time for the reason of algorithm change. In case the requested algorithm is not supported, an "Algorithm Not Supported" exception MUST be thrown.

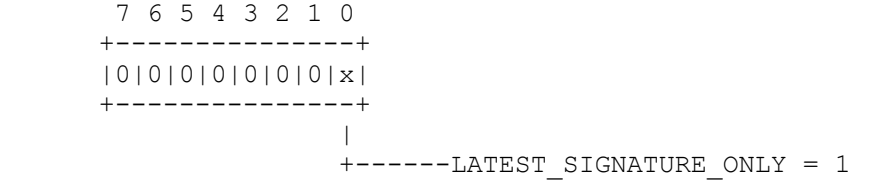
1: ALGORITHM_1:
The first older algorithm that is either the only currently active algorithm or about to be replaced by the newer algorithm specified as ALGORITHM_2.

2: ALGORITHM_2:
The newer algorithm that is intended to replace the older algorithm ALGORITHM_1.

0xFFFF:
This algorithm is just for test purpose until the BGPsec specification is finished. A request for this algorithm MUST not produce an "Algorithm Not Supported" error.

5.16. Block Type

The Block Type specifies if the return value of the signature request MUST result in the complete set of BGPSEC data or only the latest signature. If the bit LATEST_SIGNATURE_ONLY is set to "1" one, only the latest signature is transmitted.



5.17. Prepend Counter

By default, SRx uses the current AS only once for (pre)calculating signatures. For traffic engineering reasons a particular update can contain the same prefix multiple times. The "Prepend Counter" allows the calculation of signatures over multiple entities of the own prepended AS.

5.18. Length Path Validation Data

Specifies the length of the data length provided needed for path validation. For ROV validation only, no path data is required. This filed allows to optimize the data volume by setting it to zero were

the AS path is not important - ROV only. in case this value is set to zero "0", no further data MUST be read to this package.

5.18.1 Number of Hops

Specifies the number of hops within the path. This umber of hops must include repetitions because it is used to calculate the 4 Byte per entry AS Path List (see 5.18.7)

5.18.2 BGPsec_PATH Attribute Length

This specifies the complete size of the BGPsec_PATH attribute in bytes.

5.18.3 AFI

The Address Family Identifier as specified in RFC 4760

5.18.4 SAFI

The Subsequent Address Family Identifier in RFC 4760

5.18.5 Length

The prefix length in bytes ((prefix_length in bits + 7) div 8). This value is not used to specify the size of the prefix address field size, it is used to identify how many bytes are used within the prefix address filed.

5.18.6 The Prefix Address

The prefix field in network order. The size of this field is 16 bytes regardless of AFI type.

5.18.7 Local AS

The local AS number of the receiver of the UPDATE

5.18.8 AS Path List

An array containing the AS path as consecutive 4 byte ASNs. The right most AS is the originator. AS concatenations must be reflected as concatenations as well.

5.18.9 BGPsec_PATH Attribute

The BGPsec_PATH attribute as specified in RFC 8205. In case the UPDATE was a BGP4 UPDATE, this value will be omitted and the BGPsec_PATH Attribute Length field specified in 5.18.2 must be set to "0" zero.

5.20. Keep Window

Keep Window allows to request the SRx to not delete the data assigned to the current connection after the connection is terminated. If used, the keep widow specifies the time in seconds the proxy user needs to reconnect back to the SRx server. This SHOULD be the reboot time of the BGP router, approx. 15 minutes = 900 seconds. The SRx itself does not need to follow the request but is recommended to reduce the recalculation time.

5.21. Error Code

All fatal errors MUST result in a Goodbye message followed by closing the connection. Errors are only sent out from SRx, SRx itself does not except any errors. In case SRx receives an Error it MUST return an error packet with error code 2 followed by a Goodbye message.

0: Wrong Protocol Version (fatal):
The handshake fails due to a conflict of version number between the speakers.

1: Duplicate Proxy Identifier (fatal):
The handshake fails due to a conflict of the proxy identifier. An other currently active proxy is using the proxy identifier provided.

2: Invalid Packet (fatal):
This error is sent when SRx receives a packet that is either unknown or unexpected. An example could be twice a Hello Packet, a Hello Packet with insufficient number of peers provided, an Error packet send from proxy to SRx, or other.

3: Internal error (fatal):
The SRx has an internal error (memory, etc.) and is forced to abort the communication. This error might be followed by a goodbye message if possible. Otherwise the client can shut down the connection and try to reconnect after some time. For instance, 30 seconds.

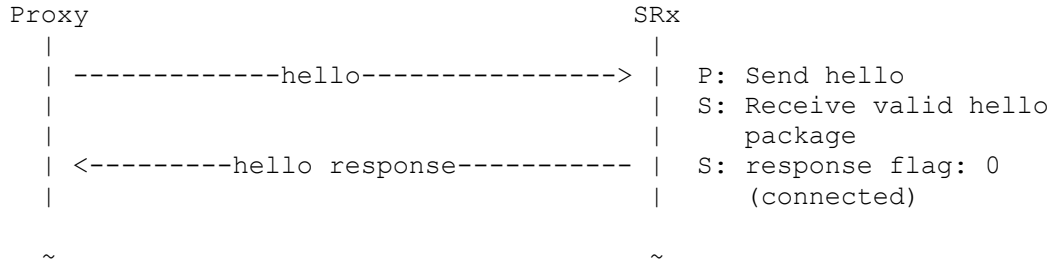
4: Algorithm Not Supported:
This error is thrown when the given algorithm for path signing / validation is not supported. This error is not considered to be fatal. It signals the proxy to resend the signing request using an alternative algorithm.

5: Update Not Found:
This error is thrown when a signing request for an update cannot be processed because the update can not be found within the database of SRx.

Even though this error is not considered to be fatal, it should result in a "Synchronization Request" to allow a synchronization between both parties.

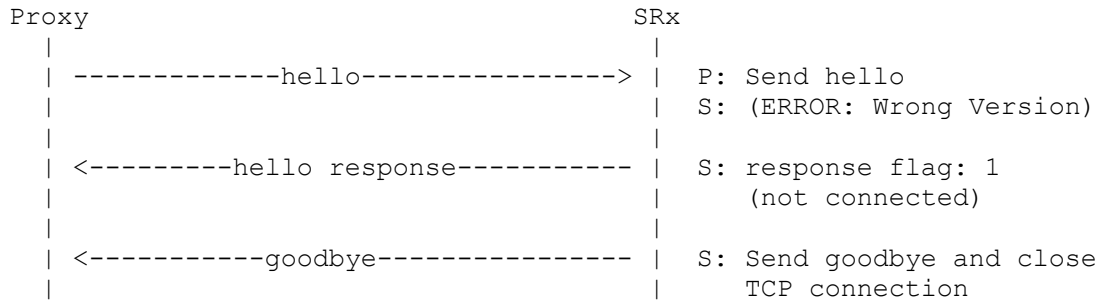
6. Communication

6.1. Establish a Connection

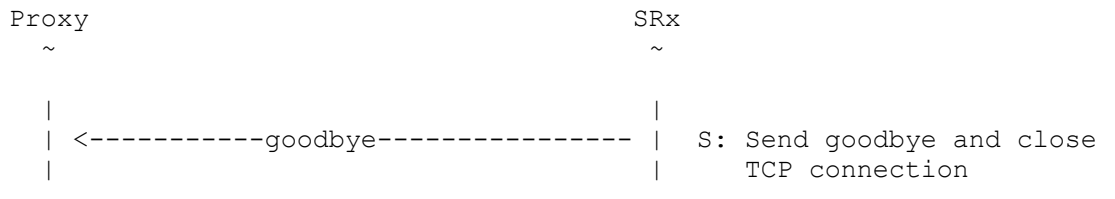


The time between sending a "hello" packet and receiving a "hello response" SHOULD not exceed 30 seconds.

In case of an error - for instance wrong version number - SRx will send an error message as response followed by a Goodbye message.

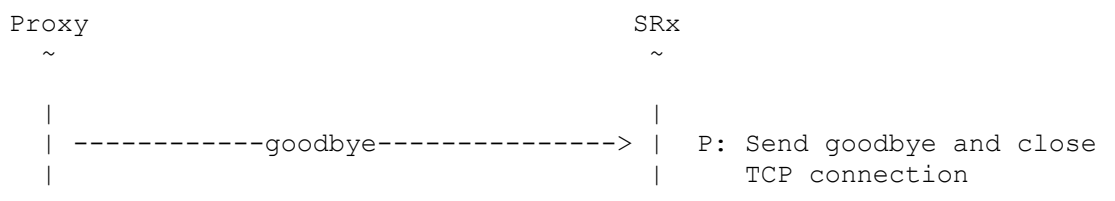


6.2. SRx server closes the connection



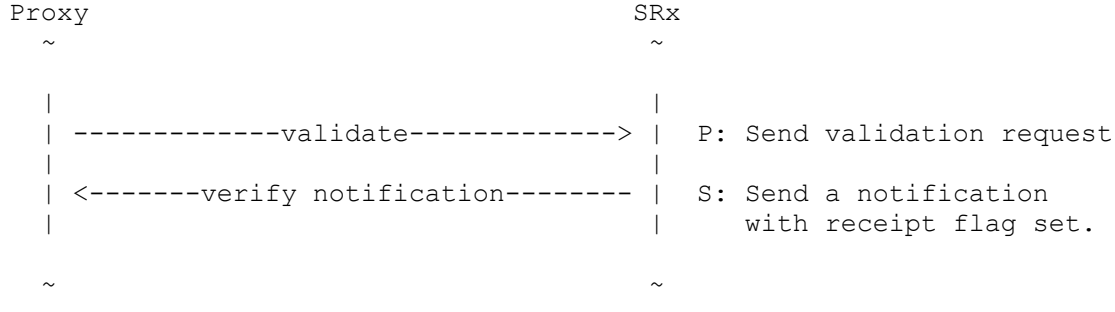
Once the goodbye message is sent out, the server can immediately tear down the connection.

6.3. Proxy closes the connection



Once the goodbye message is sent out, the proxy can immediately tear down the connection. SRx can free up all resources or keep them up for some grace period in case the connection will be reopened. The Keep Window field of the goodbye message specifies a time in seconds the data should be held.

6.4. Create a validation request



The validation request performs two major actions:

A: Initiate a request for Origin / Path validation
B: Generate a unique update id.

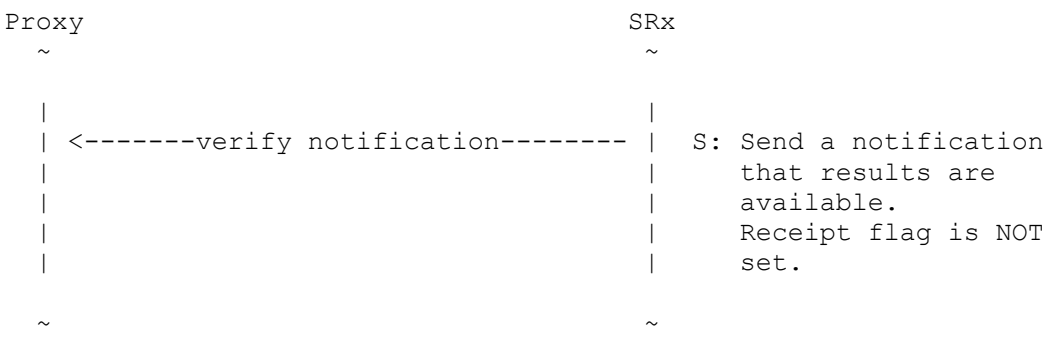
With each request the proxy provides validation results for both the requested origin validation result as well as the requested BGPsec

path validation result to SRx. SRx uses this results in case no other validation result is available.

Once SRx completed the validation and the result differs from the previous results a result notification message is send to the proxy.

In case the validation request did not specify a validation method, no further validation is started and in such case no validation notification will be sent to the proxy. This is useful in regard to updates that originate from the router that implements/uses the proxy instance. In case an update is originated the update Id is also necessary for path signing.

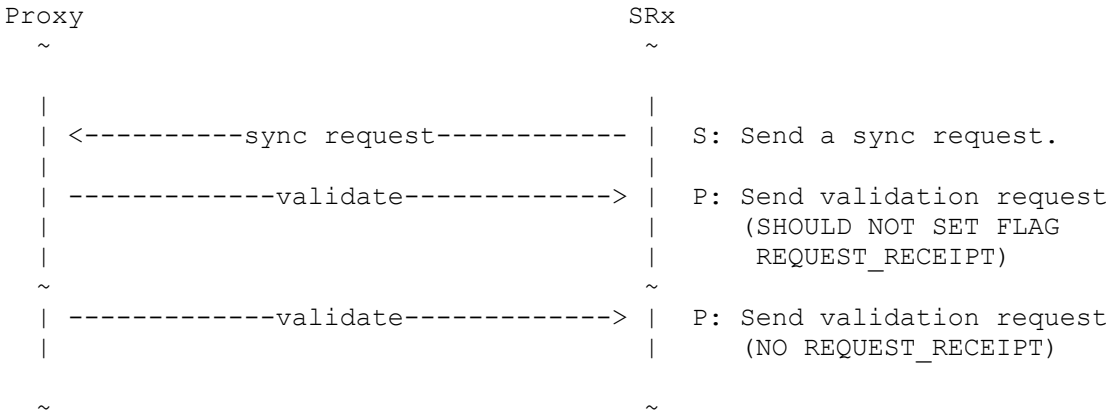
6.5. Create a validation result notification



A verify notification is send for validation prior requests only. In case a verification was requested for origin validation only, only changes within origin validation are performed. Same with path validation.

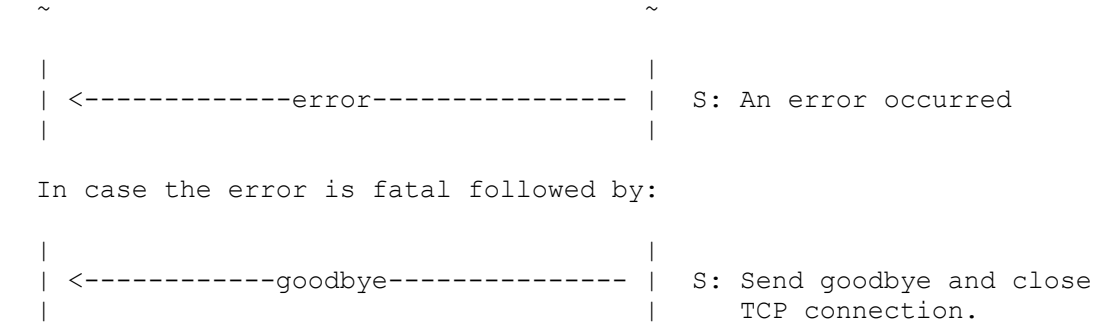
6.6. Synchronization Request

The synchronization request is initiated by the SRx due to the believe that SRx and proxy might be out of sync. The proxy need not to answer this request but it is strongly recommended. Furthermore it is recommended that the proxy provides the results once received from SRx to reduce the back traffic due to new notifications. In addition it is recommended that the proxy SHOULD NOT set the "receipt flag" to prevent receiving receipts for already known update results. In case a result differs from the provided default result, SRx will send notifications. In case nothing changed in the validation no further traffic has to be processed and therefore no change should be triggered within the decision process of the router.



6.7. Error Communication

At any point in time if an error occurs, SRx initiates an error message indicating the error. In case the error is considered fatal the connection MUST be closed.



Errors that are considered fatal require a reboot of the session.

7. Implementation Suggestions

This section deals with implementation modes this protocol tries to address. In general, it is thought that a router might access the SRx through a proxy API. The proxy API then needs to implement this protocol to talk to SRx and communicate validation requests and request results between the router and SRx. This communication can be performed in two modes, synchronous and asynchronous.

Regardless of the mode the proxy is operated in some parts are and will always be asynchronous. The synchronization defined here is in regards to validation requests and their initial result value. The router itself receives four values from SRx that need to be added to each update entry within the router.

Update Identifier
The Update Identifier (Section 5.10)

Validation Result
Origin (Section 5.7) Path (Section 5.9)

8. Acknowledgements

The Author wants to thank Doug Montgomery and Sebastian Spiess for their input.

9. References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4760] Bates, T., Chandra, R., Katz, D., and Y. Rekhter, "Multiprotocol Extensions for BGP-4", RFC 4760, DOI 10.17487/RFC4760, January 2007, <https://www.rfc-editor.org/info/rfc4760>.

[RFC6811] Mohapatra, P., Scudder, J., Ward, D., Bush, R., and R. Austein, "BGP Prefix Origin Validation", RFC 6811, DOI 10.17487/RFC6811, January 2013, <https://www.rfc-editor.org/info/rfc6811>.

[RFC8205] Lepinski, M., Ed., and K. Sriram, Ed., "BGPsec Protocol Specification", RFC 8205, DOI 10.17487/RFC8205, September 2017, <https://www.rfc-editor.org/info/rfc8205>.

[RFC8210] Bush, R. and R. Austein, "The Resource Public Key Infrastructure (RPKI) to Router Protocol, Version 1", RFC 8210, DOI 10.17487/RFC8210, September 2017, <https://www.rfc-editor.org/info/rfc8210>.

[RFC8208] Turner, S. and O. Borchert, "BGPsec Algorithms, Key Formats, and Signature Formats", RFC 8208, DOI 10.17487/RFC8208, September 2017, <https://www.rfc-editor.org/info/rfc8208>.

Authors' Addresses

Oliver Borchert
NIST
100 Bureau Drive
Gaithersburg, MD 20899
United States of America

Email: oliver.borchert@nist.gov

Kyehwan Lee
NIST
100 Bureau Drive
Gaithersburg, MD 20899
United States of America

Email: kyehwanl@nist.gov

Patrick Gleichmann
NIST
100 Bureau Drive
Gaithersburg, MD 20899
United States of America

Email: patrick.gleichmann@nist.gov

Appendix B: Notes