The RPKI Documentation

Alex Band, Tim Bruijnzeels, Martin Hoffmann and the RPKI Community

Sep 13, 2019
1 Introduction
   1.1 About this Documentation .......................................................... 3
   1.2 About Resource Public Key Infrastructure ........................................ 3
   1.3 Organisation of this Documentation ........................................... 4

2 FAQ
   2.1 RPKI Mechanism ........................................................................ 5
   2.2 Operations and Impact ................................................................. 7
   2.3 Miscellaneous ............................................................................ 9

3 Introduction
   3.1 Internet Number Resource Allocation ............................................ 11
   3.2 Mapping the Resource Allocation Hierarchy into the RPKI ................... 12
   3.3 X.509 PKI Considerations .......................................................... 14

4 Internet Routing
   4.1 BGP Best Path Selection .............................................................. 15
   4.2 Routing Errors .......................................................................... 18
   4.3 Mitigation of Routing Errors ....................................................... 18
   4.4 The Internet Routing Registry ..................................................... 18

5 Securing BGP
   5.1 Route Origin Validation .............................................................. 21
   5.2 Path Validation .......................................................................... 23

6 Implementation Models
   6.1 Hosted RPKI .............................................................................. 25
   6.2 Delegated RPKI ......................................................................... 27

7 Using RPKI Data
   7.1 Connecting to the Trust Anchor .................................................... 29
   7.2 Fetching and Verifying ................................................................. 29
   7.3 Validating Routes ....................................................................... 30
   7.4 Local Overrides .......................................................................... 31
   7.5 Feeding Routers ......................................................................... 31

8 Router Support ............................................................................. 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Hardware Solutions</td>
<td>33</td>
</tr>
<tr>
<td>8.2</td>
<td>Software Solutions</td>
<td>33</td>
</tr>
<tr>
<td>9</td>
<td>Resources</td>
<td>35</td>
</tr>
<tr>
<td>9.1</td>
<td>RPKI Insights and Statistics</td>
<td>35</td>
</tr>
<tr>
<td>9.2</td>
<td>Operational Experiences</td>
<td>35</td>
</tr>
<tr>
<td>9.3</td>
<td>Examples of BGP Hijacks</td>
<td>36</td>
</tr>
<tr>
<td>9.4</td>
<td>IETF Documents</td>
<td>36</td>
</tr>
<tr>
<td>10</td>
<td>Software Projects</td>
<td>37</td>
</tr>
<tr>
<td>10.1</td>
<td>Relying Party Software</td>
<td>37</td>
</tr>
<tr>
<td>10.2</td>
<td>Certificate Authority Software</td>
<td>37</td>
</tr>
<tr>
<td>10.3</td>
<td>Supporting Tools</td>
<td>38</td>
</tr>
<tr>
<td>11</td>
<td>Krill</td>
<td>39</td>
</tr>
<tr>
<td>11.1</td>
<td>Installation</td>
<td>39</td>
</tr>
<tr>
<td>11.2</td>
<td>Running Krill</td>
<td>41</td>
</tr>
<tr>
<td>11.3</td>
<td>Running Krill with Docker</td>
<td>43</td>
</tr>
<tr>
<td>11.4</td>
<td>Using the Krill CLI</td>
<td>46</td>
</tr>
<tr>
<td>12</td>
<td>Routinator</td>
<td>53</td>
</tr>
<tr>
<td>12.1</td>
<td>Installation</td>
<td>53</td>
</tr>
<tr>
<td>12.2</td>
<td>Initialisation</td>
<td>56</td>
</tr>
<tr>
<td>12.3</td>
<td>Running Interactively</td>
<td>58</td>
</tr>
<tr>
<td>12.4</td>
<td>Running as a Daemon</td>
<td>60</td>
</tr>
<tr>
<td>12.5</td>
<td>Configuration</td>
<td>63</td>
</tr>
<tr>
<td>12.6</td>
<td>Monitoring</td>
<td>65</td>
</tr>
<tr>
<td>13</td>
<td>RTRlib</td>
<td>69</td>
</tr>
<tr>
<td>13.1</td>
<td>About</td>
<td>69</td>
</tr>
<tr>
<td>13.2</td>
<td>Installation</td>
<td>70</td>
</tr>
<tr>
<td>13.3</td>
<td>RTRlib Command Line Tools</td>
<td>73</td>
</tr>
</tbody>
</table>
Welcome to the documentation of the Resource Public Key Infrastructure (RPKI), the community-driven technology based on open standards that is aimed at making Internet routing more secure. If you are new to this documentation, we recommend that you read the introduction page to get an overview of what this documentation has to offer.

The table of contents below and in the sidebar should let you easily access the documentation for your topic of interest. You can also use the search function in the top left corner.

**Note:** This documentation is an open source project maintained by the RPKI team at NLnet Labs, with contributions from the network operator community around the world. We always appreciate your feedback and improvements.

You can submit an issue or pull request on the GitHub repository, or post a message on the RPKI mailing list. If you are interested in providing a translation for this project, please read [this guide](#) to get started.

The main documentation is organised into the following sections:
Welcome to the documentation of the Resource Public Key Infrastructure (RPKI). Here, we aim to offer an overview of the RPKI technology itself, as well as the tools that NLnet Labs and others are developing for it.

This page gives a broad overview of the RPKI and how it can help make Internet routing using the Border Gateway Protocol (BGP) more secure. This way, you will learn how RPKI can benefit your organisation, as well as helping others to be more secure on the Internet.

1.1 About this Documentation

This documentation is continuously written, corrected and edited by the RPKI team at NLnet Labs. An initial version was written by Alex Band, Tim Bruijnzeels and Martin Hoffmann. Over time, additions from the network operator community, researchers and interested parties around the world were contributed. The documentation is edited via text files in the reStructuredText markup language and then compiled into a static website/offline document using the open source Sphinx and ReadTheDocs tools.

Note: You can contribute to the RPKI documentation by opening an issue or sending patches via pull requests on the GitHub source repository.

All the contents are under the permissive Creative Commons Attribution 3.0 (CC-BY 3.0) license, with attribution to “The RPKI team at NLnet Labs and the RPKI community”.

1.2 About Resource Public Key Infrastructure

RPKI allows holders of Internet number resources to make verifiable statements about how they intend to use their resources. To achieve this, it uses a public key infrastructure that creates a chain of resource certificates that follows the same structure as the way IP addresses and AS numbers are handed down.
RPKI is used to make Internet routing more secure. It is a community-driven system in which open source software developers, router vendors and all five Regional Internet Registries (RIRs) participate, i.e. ARIN, APNIC, AFRINIC, LACNIC and RIPE NCC.

Currently, RPKI is used to let the legitimate holder of a block of IP addresses make an authoritative statement about which AS is authorised to originate their prefix in the BGP. In turn, other network operators can download and validate these statements and make routing decisions based on them. This process is referred to as route origin validation (ROV). This provides a stepping stone to provide path validation in the future.

### 1.3 Organisation of this Documentation

This documentation is organised into three main sections:

- The *General* section contains this introduction as well as information about the licensing, authors, etc. It also contains the *FAQ*.

- The *RPKI Technology* section explains the RPKI technology and standards in order for you to get a good sense of the requirements and moving parts. It will help you choose the right RPKI solution for your organisation, with regards to generating, publishing and using RPKI data.

- The *RPKI Tools* section is about various open source projects that are maintained to support RPKI. These include projects maintained by NLnet Labs, as well as projects maintained by others in the community.
2.1 RPKI Mechanism

2.1.1 What is RPKI and why was it developed?

The global routing system of the Internet consists of a number of functionally independent actors (Autonomous Systems) which use BGP (Border Gateway Protocol) to exchange routing information. The system is very dynamic and flexible by design. Connectivity and routing topologies are subject to change. Changes easily propagate globally within a few minutes. One weakness of this system is that these changes cannot be validated against information existing outside of the BGP protocol itself.

RPKI is a way to define data in an out-of-band system such that the information that are exchanged by BGP can be validated to be correct. The RPKI standards were developed by the IETF (Internet Engineering Task Force) to describe some of the resources of the Internet’s routing and addressing scheme in a cryptographic system. These information are public, and anyone can get access to validate their integrity using cryptographic methods.

2.1.2 I thought we were all using the IRR to check route origin, why do we need RPKI now?

If you’ve been involved in default-free zone Internet engineering for any length of time, you’re probably familiar with RPSL, a routing policy specification language originally defined in RFC2280 back in 1998. While RPSL has created considerable early enthusiasm and has seen some traction, the Internet was rapidly growing at the time, and the primary focus was on data availability rather than data trustworthiness. Everyone was busy opportunistically documenting the
The minimal policy that was necessary to “make things work” with the policy specification language parsing scripts of everyone else so that something would finally ping!

Over time, this has created an extensive repository of obsolete data of uncertain validity spread across dozens of route registries around the world. Additionally, the RPSL language and supporting tools have proven to be too complex to consistently transpose policy into router configuration language - resulting in most published RPSL data being neither sufficiently accurate and up to date for filtering purposes, nor sufficiently comprehensive or precise for being the golden master in router configuration.

RPKI aims to complement and expand upon this effort focusing primarily on trustworthiness, timeliness, and accuracy of data. RPKI ROAs are hierarchically delegated by RIRs based on strict criteria, and are cryptographically verifiable. This offers the Internet community an opportunity to build an up to date and accurate information of IP address origination data on the Internet.

2.1.3 Why are we investing in RPKI, isn’t it easier to just fix the Internet Routing Registry (IRR) system?

The main weakness of the IRR is that it is not a globally deployed system and it lacks the authorisation model to make the system water tight. The result is that out of all the information on routing intent that is published, it is difficult to determine what is legitimate, authentic data and what isn’t. RPKI solves these two problems, as you can be absolutely sure that an authoritative, cryptographically verifiable statement can be made by any legitimate IP resource holder in the world.

2.1.4 Is it true that BGP4 is just not up to the task any longer?

Unfortunately it’s practically impossible to replace BGP right now. We should, however, work on fixing the broken parts and improving the situation.

2.1.5 As RPKI relies on X.509 PKI, isn’t this the same problem with untrustworthy SSL/TLS Certificate Authorities all over again?

Instead of relying on a large number of CAs subject to variable auditing standards which come pre-installed in a browser or an operating system, RPKI relies on just five Trust Anchors, run by the Regional Internet Registries. These are well established and openly governed organisations. Each operator that wishes to get an RPKI resource certificate already has a contractual relationship with one or more of the RIRs.

2.1.6 What is the value of RPKI based BGP Origin Validation without Path Validation?

While Path Validation is a desirable characteristic, the existing RPKI origin validation functionality addresses a large portion of the problem surface.

Existing operational and economic incentives ensure that the most important prefixes for each network are seen via the shortest AS path possible. One such example are network operators setting a higher local preference for prefixes learned via an Internet exchange or private peers (“peerlock”). This reduces the risk that an invalid route could win the BGP route selection process even if it originates from an impersonated but correct origin AS.

For transit providers, direct interconnections and short AS paths are a defining characteristic, positioning them ideally to act on RPKI data and accept only valid routes for redistribution.
Furthermore, operational experience suggests that the vast majority of route hijacks are unintentional rather than malicious, and are caused by ‘fat-fingering’, where an operator accidentally originates a prefix they are not the holder of. Origin Validation would mitigate many of these problems.

While a malicious party willing to intentionally impersonate the origin AS could still take advantage of the lack of Path Validation in some circumstances, widespread RPKI Origin Validation implementation would make such instances easier to pinpoint and address.

2.1.7 When comparing the ROA data set to the announcements my router sees, what are possible outcomes?

In short, routes can have the state Valid, Invalid, or NotFound (a.k.a. Unknown).

- **Valid**: The route announcement is covered by at least one ROA
- **Invalid**: The prefix is announced from an unauthorised AS or the announcement is more specific than is allowed by the maximum length set in a ROA that matches the prefix and AS
- **NotFound**: The prefix in this announcement is not covered (or only partially covered) by an existing ROA

To understand how more specifics, less specifics and partial overlaps are treated, please refer to section 2 of RFC 6811.

2.1.8 I’ve heard the term “route leak” and “route hijack”. What’s the difference?

A route leak is a propagation of one or more routing announcements that are beyond their intended scope. That is an announcement from an Autonomous System (AS) of a learned BGP route to another AS is in violation of the intended policies of the receiver, the sender, and/or one of the ASes along the preceding AS path.

A route hijack is the unauthorised origination of a route.

Note that in either case, the cause may be accidental or malicious and in either case, the result can be path detours, redirection, or denial of services. For more information, please refer to RFC 7908.

2.1.9 If a ROA is cryptographically invalid, will it make my route invalid?

An invalid ROA means that the object did not pass cryptographic validation and is therefore discarded. The statement about routing that was made within the ROA is simply not taken into consideration. An invalid route on the other hand, is the result of a valid ROA, specifically one that had the outcome that a prefix is announced from an unauthorised AS or the announcement is more specific than is allowed by the maximum length set in a ROA that matches the prefix and AS.

2.2 Operations and Impact

2.2.1 Will my router have a problem with all of this cryptographic validation?

No, routers do not do any cryptographic operations to perform Route Origin Validation. The signatures are checked by external software, called Relying Party software or RPKI Validator, which feeds the processed data to the router over a light-weight protocol. This architecture causes minimal overhead for routers.
2.2.2 Does RPKI reduce the BGP convergence speed of my routers?

No, filtering based on an RPKI validated cache has a negligible influence on convergence speed. RPKI validation happens in parallel with route learning (for new prefixes which aren’t yet in cache), and those prefixes will be marked as valid, invalid, or notfound (and the correct policy applied) as the information becomes available.

2.2.3 Why do I need rsync on my system to use a validator?

In the original standards, rsync was defined as the main means of distribution of RPKI data. While it has served the system well in the early years, rsync has several downsides:

- When RPKI relying party software is used on a client system, it has a dependency on rsync. Different versions and different supported options, such as --contimeout, cause unpredictable results. Furthermore, calling rsync is inefficient. It’s an additional process and the output can only be verified by scanning the disk.
- Scaling becomes more and more problematic as the global RPKI data set grows and more operators download and validate data, as with rsync the server in involved in processing the differences.

To overcome these limitations the RRDP protocol was developed and standardised in RFC 8182, which relies on HTTPS. RRDP was specifically designed for scaling and allows CDNs to participate in serving the RPKI data set globally, at scale. In addition, HTTPS is well supported in programming languages so development of relying party software becomes easier and more robust.

Currently, RRDP is implemented on the server side by the RIPE NCC and APNIC. It is considered as a work item for 2019 by ARIN. Most RPKI Validator implementations either already have RRDP support, or have it on the short term roadmap.

2.2.4 The five RIRs provide a Hosted RPKI system, so why would I want to run a Delegated RPKI system myself instead?

The RPKI system was designed to be a distributed system, allowing each organisation to run their own CA and publish the certificate and ROAs themselves. The hosted RIR systems are in place to offer a low entry barrier into the system, allowing operators to gain operational experience before deciding if they want to run their own CA.

For many operators, the hosted system will be good enough, also in the long term. However, organisations who for example don’t want to be dependent on a web interface for management, who manage address space across multiple RIR regions, or have BGP automation in place that they would like to integrate with ROA management, can all choose to run a CA on their own systems.

2.2.5 Should I run a validator myself, when I can use an external data source I found on the Internet?

The value of signing the authoritative statements about routing intent by the resource holder comes from being able to validate that the data is authentic and has not been tampered with in any way.

When you outsource the validation to a third party, you lose the certainty of data accuracy and authenticity. Conceptually, this is similar to DNSSEC validation, which is best done by a local trusted resolver.

Section 3 of RFC 7115 has an extensive section on this specific topic.
2.2.6 How often should I fetch new data from the RPKI repositories?

According to section 3 of RFC 7115 you should fetch new data at least every 4 to 6 hours. At the moment, the publication of new ROAs in the largest repositories takes about 10-15 minutes. This means fetching every 15-30 minutes is reasonable, without putting unnecessary load on the system.

2.2.7 What if the RPKI system becomes unavailable or some other catastrophe occurs, will my (signed) prefixes become unreachable to others? Will other prefixes my routers learned over BGP become unreachable for me?

RPKI provides a positive statement on routing intent. If all RPKI validator instances become unavailable and all certificates and ROAs expire, the validity state of all routes will fall back to NotFound, as if RPKI were never used. Routes with this state should be accepted according to section 5 of RFC 7115, as this state will unfortunately be true for the majority of routes.

2.2.8 What if the Validator I use crashes and my router stops getting a feed. What will happen to the prefixes I learn over BGP?

All routers that support Route Origin Validation allow you to specify multiple Validators for redundancy. It is recommended that you run multiple instances, preferably from independent publishers and on separate subnets. This way you rely on multiple caches.

In case of a complete failure, all routes will fall back to the NotFound state, as if Origin Validation were never used.

2.2.9 I don’t want to rely on the RPKI data set in all cases, but I want to have my own preferences for some routes. What can I do?

You can always apply your own, local overrides on specific prefixes/announcements and override the RPKI data you fetch from the repositories. Specifying overrides is in fact standardised in RFC8416, “Simplified Local Internet Number Resource Management with the RPKI (SLURM)”.

2.2.10 Is there any point in signing my routes with ROAs if I don’t validate and filter myself?

Yes, signing your routes is always a good idea. Even if you don’t validate yourself someone else will, or in worst case someone else might try to hijack your prefix. Imagine what could happen if you haven’t signed your prefixes…

2.3 Miscellaneous

2.3.1 What is the global adoption and data quality of RPKI like?

There are several initiatives that measure the adoption and data quality of RPKI:

- RPKI Analytics, by NLnet Labs
- Global certificate and ROA statistics, by RIPE NCC
- Cirrus Certificate Transparency Log, by Cloudflare
- The RPKI Observatory, by nusenu
2.3.2 I want to use the RPKI services from a specific RIR that I'm not currently a member of. Can I transfer my resources?

The RPKI services that each RIR offers differ in conditions, terms of service, availability and usability. Most RIRs have a transfer policy that allow their members to transfer their resources from one RIR region to another. Organisations may wish to do this so that they bring all resources under one entity, simplifying management. Others may do this because they are are looking for a specific set of terms with regards to the holdership of their resources. Please check with your RIR for the possibilities and conditions for resource transfers.

2.3.3 Will RPKI be used as a censorship mechanism allowing governments to make arbitrary prefixes unroutable on a whim?

Unlikely. In order to suppress a prefix, it would be necessary to both revoke the existing ROA (if one is present) and publish a conflicting ROA with a different origin.

These characteristics make using RPKI as a mechanism for censorship a rather convoluted and uncertain way of achieving this goal, and has broad visibility (as the conflicting ROA, as well as the Regional Internet Registry under which it was issued, will be immediately accessible to everyone). A government would be much better off walking into the data center and confiscate your equipment.

2.3.4 What are the long-term plans for RPKI?

With RPKI Route Origin Validation being deployed in more and more places, there are several efforts to build upon this to offer out-of-band Path Validation. Autonomous System Provider Authorisation (ASPA) currently has the most traction in the IETF, defined in these drafts: draft-azimov-sidrops-aspa-profile and draft-azimov-sidrops-aspa-verification.
Resource Public Key Infrastructure (RPKI) revolves around the right to use Internet number resources, such as IP addresses and autonomous system (AS) numbers.

In this PKI, the legitimate holder of a block of IP addresses or AS numbers can obtain a resource certificate. Using the certificate, they can make authoritative, signed statements about the resources listed on it. To understand the structure of RPKI and its usage, we must first look at how Internet number resources are allocated globally.

### 3.1 Internet Number Resource Allocation

Before being formalised within an organisation, the allocation of Internet number resources, such as IP addresses and AS numbers, had been the responsibility of Jon Postel. At the time, he worked at the Information Sciences Institute (ISI) of the University of Southern California (USC). He performed the role of Internet Assigned Numbers Authority (IANA), which is presently a function of the Internet Corporation for Assigned Names and Numbers (ICANN).

Initially, the IANA function was performed globally, but as the work volume grew due to the expansion of the Internet, Regional Internet Registries (RIRs) were established over the years to take on this responsibility on a regional level. Until the available pool of IPv4 depleted in 2011, this meant that periodically, a large block of IPv4 address space was allocated from IANA to one of the RIRs. In turn, the RIRs would allocate smaller blocks to their member organisations, and so on. IPv6 address blocks and AS numbers are allocated in the same way.

Today, there are five RIRs responsible for the allocation and registration of Internet number resources within a particular region of the world:

- The African Network Information Center (AFRINIC) serves Africa
- The American Registry for Internet Numbers (ARIN) serves Antarctica, Canada, parts of the Caribbean, and the United States
- The Asia-Pacific Network Information Centre (APNIC) serves East Asia, Oceania, South Asia, and Southeast Asia
- The Latin America and Caribbean Network Information Centre (LACNIC) serves most of the Caribbean and all of Latin America
• The Réseaux IP Européens Network Coordination Centre (RIPE NCC) serves Europe, the Middle East, Russia, and parts of Central Asia.

In the APNIC and LACNIC regions, Internet number resources are in some cases allocated to National Internet Registries (NIRs), such as NIC.br in Brazil and JPNIC in Japan. NIRs allocate address space to its members or constituents, which are generally organised at a national level. In the rest of world, the RIRs allocate directly to their member organisations, typically referred to as Local Internet Registries (LIRs). Most LIRs are Internet service providers, enterprises, or academic institutions. LIRs either use the allocated IP address blocks themselves, or assign them to End User organisations.

### 3.2 Mapping the Resource Allocation Hierarchy into the RPKI

As illustrated, the IANA has the authoritative registration of IPv4, IPv6 and AS number resources that are allocated to the five RIRs. Each RIR registers authoritative information on the allocations to NIRs and LIRs, and lastly, LIRs record to which End User organisation they assigned resources.

In RPKI, resource certificates attest to the allocation by the issuer of IP addresses or AS numbers to the subject. As a result, the certificate hierarchy in RPKI follows the same structure as the Internet number resource allocation hierarchy, with the exception of the IANA level. Instead, the five RIRs each run a root CA with a trust anchor from which a chain of trust for the resources they each manage is derived.

The IANA does not operate a single root certificate authority (CA). While this was originally a recommendation from the Internet Architecture Board (IAB) to eliminate the possibility of resource conflicts in the system, they reconsidered after operational experience in deployment had caused the RIRs to conclude that the RPKI system would be less brittle using multiple overlapping trust anchors.
Fig. 3.2: The service regions of the five Regional Internet Registries

Fig. 3.3: Internet number resource allocation hierarchy
3.3 X.509 PKI Considerations

The digital certificates used in RPKI are based on X.509, standardised in RFC 5280, along with extensions for IP addresses and AS identifiers described in RFC 3779. Because RPKI is used in the routing security context, a common misconception is that this is the Routing PKI. However, certificates in this PKI are called resource certificates and conform to the certificate profile described in RFC 6487.

**Note:** X.509 certificates are typically used for authenticating either an individual or, for example, a website. In RPKI, certificates do not include identity information, as their only purpose is to transfer the right to use Internet number resources.

In addition to RPKI not having any identity information, there is another important difference with commonly used X.509 PKIs, such as SSL/TLS. Instead of having to rely on a vast number of root certificate authorities which come pre-installed in a browser or an operating system, RPKI relies on just five trust anchors, run by the RIRs. These are well established, openly governed, not-for-profit organisations. Each organisation that wishes to get an RPKI resource certificate already has a contractual relationship with one or more of the RIRs.

In conclusion, RPKI provides a mechanism to make strong, testable attestations about Internet number resources. In the next sections, we will look at how this can be used to make Internet routing more secure.
To understand how RPKI is used to make Internet routing more secure, we must first look at how routing works, what the weaknesses are and which elements RPKI can currently help protect against.

The global routing system of the Internet consists of a number of functionally independent actors called autonomous systems (AS), which use the Border Gateway Protocol (BGP) to exchange routing information.

An autonomous system is a set of Internet routable IP prefixes belonging to a network or a collection of networks that are all managed and supervised by a single entity or organisation. An AS utilises a common routing policy controlled by the entity and is identified by a globally unique 16 or 32-bit number. The AS number (ASN) is assigned by one of the five Regional Internet Registries (RIRs), just like IP address blocks.

The Border Gateway Protocol manages the routed peerings, prefix advertisement and routing of packets between different autonomous systems across the Internet. BGP uses the ASN to uniquely identify each system. In short, BGP is the routing protocol for AS paths across the Internet. The system is very dynamic and flexible by design. Connectivity and routing topologies are subject to change, which easily propagate globally within a few minutes.

Fundamentally, BGP is based on mutual trust between networks. When a network operator configures the routers in their AS, they specify which IP prefixes to originate and announce to their peers. There is no authentication or authorisation embedded within BGP. In principle, an operator can define any ASN as the origin and announce any prefix, also one they are not the holder of.

### 4.1 BGP Best Path Selection

BGP routing information includes the complete route to each destination. BGP uses the routing information to maintain a database of network reachability information, which it exchanges with other networks. For each prefix in the routing table, BGP continuously and dynamically makes decisions about the best path to reach a particular destination. After the best path is selected, the route is installed in the routing table.

Though there are many factors at play, two of them are the most important to keep in mind throughout the next sections: the preference for shortest path and most specific IP prefix.
4.1.1 Preference for Shortest Path

Out of all the possible routes that a router has in its Routing Information Base (RIB), BGP will always prefer the shortest path to its destination, minimising the amount of hops. When two matching prefixes are announced from two different networks on the Internet, BGP will route traffic to the destination that is topologically closest. This is an important feature of BGP, but when configuration errors occur, it can also be the cause of reachability problems.

![Diagram of network showing preference for shortest path](image)

Fig. 4.1: When the announcement of a prefix is an exact match, the shortest path wins

4.1.2 Preference for Most Specific Prefix

Regardless any local preference, path length or any other attributes, when building the forwarding table, the router will always select most specific IP prefix available. This behaviour is important, but creates the possibility for almost any network to attract someone else’s traffic by announcing an overlapping more specific.

With this in mind, there are several problems that can arise as a result of this behaviour.
Fig. 4.2: Regardless of the path length, the announcement of a more specific prefix always wins
4.2 Routing Errors

Routing errors on the Internet can be classified as route leaks or route hijacks. RFC 7908 provides a working definition of a BGP route leak:

A route leak is the propagation of routing announcement(s) beyond their intended scope. That is, an announcement from an Autonomous System (AS) of a learned BGP route to another AS is in violation of the intended policies of the receiver, the sender, and/or one of the ASes along the preceding AS path. The intended scope is usually defined by a set of local redistribution/filtering policies distributed among the ASes involved. Often, these intended policies are defined in terms of the pair-wise peering business relationship between autonomous systems.

A route hijack, also called prefix hijack, or IP hijack, is the unauthorised origination of a route.

Note: Route leaks and hijacks can be accidental or malicious, but most often arise from accidental misconfigurations. The result can be redirection of traffic through an unintended path. This may enable eavesdropping or traffic analysis and may, in some cases, result in a denial of service or black hole.

Routing incidents occur every day. While several decades ago outages and redirections were often accidental, in recent years they have become more malicious in nature. Some notable events were the AS 7007 incident in 1997, Pakistan’s attempt to block YouTube access within their country, which resulted in taking down YouTube entirely in 2008, and lastly, the almost 1,300 addresses for Amazon Route 53 that got rerouted for two hours in order to steal cryptocurrency, in 2018.

4.3 Mitigation of Routing Errors

One weakness of BGP is that routing errors cannot be easily be deduced from information within the protocol itself. For this reason, network operators have to carefully gauge what the intended routing policy of their peers is. As a result, it is imperative that networks employ filters to only accept legitimate traffic and drop everything else.

There are several well known methods to achieve this. Certain backbone and private peers require a valid Letter of Agency (LOA) to be completed prior to allowing the announcement or re-announcement of IP address blocks. A more widely accepted method is the use of Internet Routing Registry (IRR) databases, where operators can publish their routing policy. Both methods allow other networks to set up filters accordingly.

4.4 The Internet Routing Registry

The Internet Routing Registry (IRR) is a distributed set of databases allowing network operators to describe and query for routing intent. The IRR is used as a verification mechanism of route origination and is widely, though not universally, deployed to prevent accidental or intentional routing disturbances.

The notation used in the IRR is the Routing Policy Specification Language (RPSL), which was originally defined in RFC 2280 in 1998. RPSL is a very expressive language, allowing for an extremely detailed description of routing policy. While IRR usage had created considerable early enthusiasm and has seen quite some traction, the Internet was rapidly growing at the time. This meant that the primary focus was on data availability rather than data trustworthiness.

In later years, it was considered a good practice to extensively document how incoming and outgoing traffic was treated by the network, but nowadays the most prevalent usage is to publish and query for route objects, describing from which ASN a prefix is intended to be originated:
As explained earlier, only the Regional Internet Registries have authoritative information on the legitimate holder of an Internet number resource. This means that the entries in their IRR databases are authenticated, but they are not in any of the other routing registries. Over time, this has created an expansive repository of obsolete data of uncertain validity, spread across dozens of routing registries around the world.

Additionally, the RPSL language and supporting tools have proven to be too complex to consistently transpose policy into router configuration language. This resulted in most published RPSL data being neither sufficiently accurate and up to date for filtering purposes, nor sufficiently comprehensive or precise for being the golden master in router configuration.

In conclusion, the main weakness of the IRR is that it is not a globally deployed system and it lacks the authorisation model to make the system water tight. The result is that out of all the information on routing intent that is published, it is difficult to determine what is legitimate, authentic data and what isn’t.

RPKI solves these problems, as you can be absolutely sure that an authoritative, cryptographically verifiable statement can be made by any legitimate IP resource holder in the world. In the next sections we will look at how this is achieved.
Now that we’ve looked at how the RPKI structure is built and understand the basics of Internet routing, we can look at how RPKI can be used to make BGP more secure.

RPKI provides a set of building blocks allowing for various levels of protection of the routing system. The initial goal is to provide route origin validation, offering a stepping stone to providing path validation in the future. Both origin validation and path validation are documented IETF standards. In addition, there are drafts describing autonomous system provider authorisation, aimed at providing a more lightweight, incremental approach to path validation.

5.1 Route Origin Validation

With route origin validation (ROV), the RPKI system tries to closely mimic what route objects in the IRR intend to do, but then in a more trustworthy manner. It also adds a couple of useful features.

Origin validation is currently the only functionality that is operationally used. The five RIRs provide functionality for it, there is open source software available for creation and publication of data, and all major router vendors have implemented ROV in their platforms. Various router software implementations offer support for it, as well.

5.1.1 Route Origin Authorisations

Using the RPKI system, the legitimate holder of a block of IP addresses can use their resource certificate to make an authoritative, signed statement about which autonomous system is authorised to originate their prefix in BGP. These statements are called Route Origin Authorisations (ROAs).

The creation of a ROA is solely tied to the IP address space that is listed on the certificate and not to the AS numbers. This means the holder of the certificate can authorise any AS to originate their prefix, not just their own autonomous systems.

Maximum Prefix Length

In addition to the origin AS and the prefix, the ROA contains a maximum length (maxLength) value. This is an attribute that a route object in RPSL doesn’t have. Described in RFC 6482, the maxLength specifies the maximum
length of the IP address prefix that the AS is authorised to advertise. This gives the holder of the prefix control over the level of deaggregation an AS is allowed to do.

For example, if a ROA authorises a certain AS to originate 192.0.1.0/24 and the maxLength is set to /25, the AS can originate a single /24 or two adjacent /25 blocks. Any more specific announcement is unauthorised by the ROA. Using this example, the shorthand notation for prefix and maxLength you will often encounter is 192.0.1.0/24-25.

**Warning:** According to RFC 7115, operators should be conservative in use of maxLength in ROAs. For example, if a prefix will have only a few sub-prefixes announced, multiple ROAs for the specific announcements should be used as opposed to one ROA with a long maxLength.

**Liberal usage of maxLength opens up the network to a forged origin attack.** ROAs should be as precise as possible, meaning they should match prefixes as announced in BGP.

In a forged origin attack, a malicious actor spoofs the AS number of another network. With a minimal ROA length, the attack does not work for sub-prefixes that are not covered by overly long maxLength. For example, if, instead of 10.0.0.0/16-24, one issues 10.0.0.0/16 and 10.0.42.0/24, a forged origin attack cannot succeed against 10.0.666.0/24. They must attack the whole /16, which is more likely to be noticed because of its size.

### 5.1.2 Route Announcement Validity

When a network operator creates a ROA for a certain combination of origin AS and prefix, this will have an effect on the RPKI validity of one or more route announcements. Once a ROA is validated, the resulting object contains an IP prefix, a maximum length, and an origin AS number. This object is referred to as validated ROA payload (VRP).

When comparing VRPs to route announcements seen in BGP, RFC 6811 describes their possible statuses, which are:

- **Valid** The route announcement is covered by at least one VRP. The term _covered_ means that the prefix in the route announcement is equal, or more specific than the prefix in the VRP.

- **Invalid** The prefix is announced from an unauthorised AS, or the announcement is more specific than is allowed by the maxLength set in a VRP that matches the prefix and AS.

- **NotFound** The prefix in this announcement is not, or only partially covered by a VRP.
Anyone can download and validate the published certificates and ROAs and make routing decisions based on these three outcomes. In the Using RPKI Data section, we’ll cover how this works in practice.

5.2 Path Validation

Currently, RPKI only provides origin validation. While BGPsec path validation is a desirable characteristic and standardised in RFC 8205, real-world deployment may prove limited for the foreseeable future. However, RPKI origin validation functionality addresses a large portion of the problem surface.

For many networks, the most important prefixes can be found one AS hop away (coming from a specific peer, for example), and this is the case for large portions of the Internet from the perspective of a transit provider - entities which are ideally situated to act on RPKI data and accept only valid routes for redistribution.

Furthermore, the vast majority of route hijacks are unintentional, and are caused by ‘fat-fingering’, where an operator accidently originates a prefix they are not the holder of.

Origin validation would mitigate most of these problems, offering immediate value of the system. While a malicious party could still take advantage of the lack of path validation, widespread RPKI implementation would make such instances easier to pinpoint and address.

With origin validation being deployed in more and more places, there are several efforts to build upon this to offer out-of-band path validation. Autonomous system provider authorisation (ASPA) currently has the most traction in the IETF, and is described in these drafts: draft-azimov-sidrops-aspa-profile and draft-azimov-sidrops-aspa-verification.
RPKI is designed to allow every resource holder to generate and publish cryptographic material on their own systems. This is commonly referred to as delegated RPKI. To offer a turn-key solution, each RIR also offers a hosted RPKI system in their member portals. Both models have their own advantages, based on the specific requirements of the organisation using the system.

No matter what implementation model you choose, it always a good idea to publish ROAs for your BGP announcements. Even when you are still evaluating how to deploy RPKI within your organisation, the benefits are immediate. Others can already filter based on what you publish, offering protection for you and other Internet users. For example, in case someone inadvertently announces your address space from their AS, it will be flagged as Invalid and dropped by everyone who has deployed route origin validation.

**Important:** Once you start authorising announcements with RPKI, it is imperative that ROAs are created for all route origins from the prefixes you hold, including more specifics announced by other business units or customers. In addition, RPKI should become a standard part of operations, ensuring staff is trained and ROAs are continually monitored and maintained.

### 6.1 Hosted RPKI

In 2008, when the five RIRs committed to start offering RPKI services, it was clear that there would be an early adopters phase for a considerable amount of time. Given the past experiences with IPv6 and DNSSEC uptake, the RIRs decided to offer a hosted RPKI solution to lower the entry barrier into the technology. This way, organisations could easily get operational experience with the technology, without having to manage a certificate authority themselves.

Hosted RPKI offers a fair balance between ease-of-use, maintenance and flexibility. It allows users to log into their RIR member portal and request a resource certificate, which is securely hosted on the servers of the RIR. All cryptographic operations, such as key roll overs, are automated. The certificates and ROA are published in repositories hosted by the RIR. In short, there is nothing that the user has to manage, apart from creating and maintaining ROAs.

The functionality and user interfaces of the hosted RPKI implementations vary greatly across the five RIRs. Despite these variations, if you are an organisation with a single ASN and a handful of statically announced IP address blocks that are not delegated to customers, hosted RPKI is sufficient for most use cases.
Fig. 6.1: Example of the Hosted RPKI interface of the RIPE NCC
6.1.1 Functional differences across RIRs

This section provides an overview of the functionality each RIR provides to help users manage RPKI, which is summarised in the table below.

First, the table indicates if the RPKI system supports setting up delegated RPKI, so users can run their own certificate authority if they want. When using the hosted RPKI system, there is an overview if multiple users can be authorised to manage ROAs, and whether they can authenticate using two-factors.

To make management of ROAs easier, some systems provide a list of all announcements with certified address space that are seen by BGP route collectors, such as the RIPE Routing Information Service (RIS). ROAs have an explicit start and end validity date, but in some cases it is possible to automatically renew the ROAs, so that they are valid for as long as there is an entry in the web interface. In addition, it may be possible to synchronise the management of “route” objects in the IRR with the ROAs that are created. An application programming interface (API) may be provided to make batch processing easier.

To improve retrieval of published RPKI data by relying party software, the RPKI Repository Delta Protocol (RRDP) protocol was developed. Support for this standard is listed as well.

Lastly, nonrepudiation refers to the inability for a party to dispute or deny having performed an action.

<table>
<thead>
<tr>
<th></th>
<th>APNIC</th>
<th>AFRINIC</th>
<th>ARIN</th>
<th>LACNIC</th>
<th>RIPE NCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for delegated RPKI</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes¹</td>
</tr>
<tr>
<td>Multi-user support</td>
<td>Yes</td>
<td>Yes²</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Two-factor authentication</td>
<td>Yes</td>
<td>No</td>
<td>Yes³</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>BGP route collector suggestions</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes⁴</td>
<td>Yes</td>
</tr>
<tr>
<td>Auto-renew ROAs</td>
<td>Yes²</td>
<td>No</td>
<td>No</td>
<td>Yes⁵</td>
<td>Yes</td>
</tr>
<tr>
<td>Match “route” objects with ROAs</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>API</td>
<td>No</td>
<td>No</td>
<td>Yes⁶</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Publication via RRDP</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Nonrepudiation</td>
<td>No</td>
<td>No</td>
<td>Yes⁷</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

A final differentiator is the publication interval of each RIR repository. Please keep in mind that once a ROA is created by a user in one of the hosted systems, it can take between several minutes up to multiple hours before the object is published and available for download, depending on the RIR system you use.

6.2 Delegated RPKI

Operators who prefer more control and have better integration with their systems can run their own child CA. This is model is usually referred to as delegated RPKI.

In this model, the certificate authority that manages object signing is functionally separated from the publication of cryptographic material. This means that an organisation can run a CA and either publish themselves, or delegate this responsibility to a third party, such as a hosting company or cloud provider.

There may be various reasons for organisations to choose this model. For example, this may be useful for organisations that need to be able to delegate RPKI to their customers or different business units, so that that they can run their a CA on their systems and manage ROAs themselves.

¹ Coming June 2019.
² Currently upon request only.
³ Requires a client X.509 certificate to use RPKI.
⁴ Requires a ROA Request Key Pair.
⁵ Explicit opt-in feature.
⁶ Only possible to create ROAs; no list, update or delete.
Alternatively, enterprises who manage large amounts of address space across various RIRs, may not want to manage ROAs in up to five different web interfaces. Instead, they might prefer to be operationally independent from the RIR and manage everything from within one package that is tightly integrated with IP address management and provisioning systems.

Lastly, in the LACNIC and APNIC regions there are several National Internet Registries who provide registration services on a national level to their members and constituents. They also need to be operationally independent and run a certificate authority as a child of their RIR.
Validation is a key part of any public key infrastructure. The value from signing comes with validation, and should always be done by the party relying on the data. If validation is outsourced to a third party, you can never be certain if the data is complete, or tampered with in any way.

Operators who want to deploy route origin validation in their BGP decision making process have to fetch and validate all of the published RPKI data. As with any PKI, you have to start with one or more entities you are prepared to trust. In the case of RPKI, these are the five Regional Internet Registries.

### 7.1 Connecting to the Trust Anchor

When you want to retrieve all RPKI data, you connect to the trust anchor that each RIR provides. The root certificate contains pointers to its children, which contain pointers to their children, and so on. These certificates, and other cryptographic material such as ROAs, can be published in the repository that the RIR provides, or a repository operated by an organisation who either runs delegated RPKI themselves, or hosts a repository as a service. As a person who wants to fetch and validate the data, formally known as a relying party, it is not a concern where data is published. By simply connecting to the trust anchor, the chain of trust is followed automatically.

The RIR trust anchor is found through a static trust anchor locator (TAL), which is a very simple file that contains a URL to retrieve the trust anchor and a public key to verify its authenticity. The reason the TAL exists is because it’s very likely that the contents of the self signed root certificate change, due to resource transfers between RIRs. By using a TAL, the data in the trust anchor can change, without it needing to be redistributed.

### 7.2 Fetching and Verifying

Various open source relying party software packages, also known as RPKI validators, are available in order to download, verify and process RPKI data. Please note that most RPKI validators come preinstalled with TALs for all RIRs except the one for ARIN, as they require users to first review and agree to their Relying Party Agreement.

When the validator runs, it will start retrieval at each of the RIR trust anchors and follows the chain of trust to fetch all published certificates and ROAs. Fetching data is currently done via rsync. RIRs and software developers are
The RPKI Documentation

gradually migrating to the RPKI Repository Delta Protocol (RRDP) for retrieval, standardised in RFC 8182. This protocol uses HTTPS, which makes development and implementation easier, and opens up possibilities for Content Delivery Networks to participate in serving RPKI data.

Once the data has been downloaded, the validator will verify the signatures on all objects and output the valid route origins as a list. Each object in this list contains an IP prefix, a maximum length, and an origin AS number. This object is referred to as validated ROA payload (VRP). The collection of VRPs is known as the validated cache.

Note: Objects that do not pass cryptographic verification are discarded. Any statements made about route origins are not considered, as if a ROA was never published. As a result, they will not affect any route announcements.

Please note that objects that do not pass cryptographic verification are sometimes referred to as ‘invalid ROAs’, but we like to avoid this term because validity is used elsewhere in a different context.

Fetching and verification of data should be performed periodically, in order to process updates. Though the standards recommend retrieval at least once every 24 hours, current operational practice recommends that processing updates every 30 to 60 minutes is reasonable.

7.3 Validating Routes

As explained in the Route Origin Validation section, when comparing VRPs to the route announcements seen in BGP, it will have an effect on their RPKI validity state. They can be:

Valid The route announcement is covered by at least one VRP. The term covered means that the prefix in the route announcement is equal, or more specific than the prefix in the VRP.

Invalid The prefix is announced from an unauthorised AS, or the announcement is more specific than is allowed by the maxLength set in a VRP that matches the prefix and AS.

NotFound The prefix in this announcement is not, or only partially covered by a VRP.

Please carefully note the use of the word validity. Because RPKI revolves around signing and verifying cryptographic objects, it’s easy to confuse this term with the validity state of a BGP announcement. As mentioned, it can occur that a ROA doesn’t pass cryptographic verification, for example because it expired. As a result, it is discarded and will not affect any BGP announcement. In turn, only a validated ROA payload—sometimes referred to as ‘valid ROA’—can make a BGP announcement Valid or Invalid.

A route announcement may be covered by several VRPs. For example, there may be a VRP for the aggregate announcement, which overlaps with a customer announcement of a more specific prefix from a different AS. A route announcement will be Valid as long as there is one covering VRP that authorises it.

Based on the three validity outcomes, operators can make an informed decision what to do with the BGP route announcements they see. As a general guideline, announcements with Valid origins should be preferred over those with NotFound or Invalid origins. Announcements with NotFound origins should be preferred over those with Invalid origins.

As origin validation is deployed incrementally, the amount of IP address space that is covered by a ROA will gradually increase over time. Therefore, accepting the NotFound validity should be done for the foreseeable future.

Important: For route origin validation to succeed in its objective, operators should ultimately drop all BGP announcements that are marked as Invalid. Before taking this step, organisations should first analyse the effects of doing this, to avoid unintended results. Initially accepting Invalid announcements and giving them a lower preference, as well as tagging them with a BGP community is a good first step to measure this.
7.4 Local Overrides

Sometimes there is an operational need to accept Invalid announcements temporarily. Local overrides allow you to manage your own exceptions to the validated cache. This ensures that you remain in full control of the VRPs used by your routers. For example, if an Invalid origin is the result of a misconfigured ROA, you may accept it until the operator in question has resolved the issue. A format named SLURM is available for this, which is standardised in RFC 8416.

SLURM provides several ways to achieve exceptions. First, you can add a VRP specifically for the affected route by specifying the correct ASN, prefix and maximum length. Secondly, you can filter out an existing VRP, thereby moving the route back to NotFound state. In general, the former is the safer way, as it deals better with changing ROAs. Lastly, it is possible to allow all routes from a certain ASN or prefix. It is advised to use overrides with care, as liberal usage may have unintended consequences.

7.5 Feeding Routers

The validated cache can be fed directly into RPKI-capable routers via the RPKI to Router Protocol (RPKI-RTR), described in RFC 8210. Many routers, including Cisco, Juniper, Nokia, as well as BIRD and OpenBGPD support processing the validated cache. Alternatively, most validators can export the cache in various useful formats for processing outside of the router, in order to set up filters.

![Diagram of RPKI publication, data retrieval, validation and processing](image)

Fig. 7.1: RPKI publication, data retrieval, validation and processing

Note that your router does not perform any of the cryptographic validation, this is all handled by the relying party software. In addition, using RPKI causes minimal overhead for routers and has a negligible influence on convergence speed. Validation happens in parallel with route learning for new prefixes which are not yet in the cache. Those prefixes will be marked as Valid, Invalid, or NotFound as the information becomes available, after which the correct policy is applied.

Please keep in mind that the RPKI validator software you run in your network fetches cryptographic material from the outside world. To do this, it needs at least ports 873 and 443 open for rsync and HTTPS, respectively. In most cases, the processed data is fed to a router via RPKI-RTR over a clear channel, as it’s running in your local network. Currently, only Cisco IOS-XR provides a practical means to secure transports for RPKI-RTR, using either SSH or TLS.
It is recommended to run multiple validator instances as a failover measure. The router will use the union of RPKI data from all validators to which they are connected. This means that (temporary) differences in the validated cache produced by the validators, for example due to differing fetching intervals, does not pose a problem.

In the *Router Support* section we will look at which routers support route origin validation, and how to get started with each.
Several router vendors participated in the development of the RPKI standards in the IETF, ensuring the technology offered an end-to-end solution for route origin validation. The RPKI to Router protocol (RPKI-RTR) is standardised in RFC 6810 (v0) and RFC 8210 (v1). Is it specifically designed to deliver validated prefix origin data to routers. This, as well as origin validation functionality, is currently available in on various hardware platforms and software solutions.

### 8.1 Hardware Solutions

**Juniper** — [Documentation — Day One Book](#) Junos version 12.2 and newer

**Cisco** — [Documentation](#) IOS release 15.2 and newer, as well as Cisco IOS/XR since release 4.3.2.

**Nokia** — [Documentation](#) Release R12.0R4 and newer, running on the 7210 SAS, 7750 SR, 7950 XRS and the VSR.

### 8.2 Software Solutions

Various software solutions have support for origin validation:

- BIRD
- OpenBGPD
- FRRouting
- GoBGP
- VyOS

In some solutions, such as OpenBGPD, RPKI-RTR is not available but the same result can be achieved through a static configuration. The router will periodically fetch the validated cache and allow operators to set up route maps based on the result. Relying party software such as [Routinator](#) can export validated data in a format that OpenBGPD can parse.
RTRlib is a C library that implements the client side of the RPKI-RTR protocol, as well as route origin validation. RTRlib powers RPKI in BGP software routers such as FRR. In a nutshell, it maintains data from RPKI relying party software and allows to verify whether an autonomous system (AS) is the legitimate origin AS, based on the fetched valid ROA data. BGP-SRx by NIST is a prototype that can perform similar functions.
This page provides an overview of projects that support RPKI. It includes, statistics, measurements projects and presentations about operational experiences. Finally, there is an overview of all work in the Internet Engineering Task Force relevant to RPKI.

The Software Projects page an overview of all available tools for using RPKI.

### 9.1 RPKI Insights and Statistics

There are several initiatives that measure the adoption and data quality of RPKI:

- Global country stats, with AS and IP prefix analysis, by NLnet Labs
- Cirrus Certificate Transparency Log, by Cloudflare
- Global certificate and ROA statistics, by RIPE NCC
- RPKI Deployment Monitor, by NIST
- The RPKI Observatory, by nusenu
- RPKI connection test, by RIPE Labs

### 9.2 Operational Experiences

**Wikimedia RPKI Validation Implementation** Documentation by Arzhel Younsi describing RPKI validator and router configuration

**Dropping RPKI invalid routes in a service provider network** Lightning talk by Nimrod Levy - AT&T, NANOG 75, February 2019

**RPKI and BGP: our path to securing Internet Routing** Blog post by Jérôme Fleury & Louis Poinsignon - Cloudflare, September 2018

**RPKI For Managers** Presentation by Niels Raijer - Fusix Networks, NLNOG Day 2018, September 2018
9.3 Examples of BGP Hijacks

How Verizon and a BGP Optimizer Knocked Large Parts of the Internet Offline Today  Cloudflare Blog, 24 June 2019

BGP / DNS Hijacks Target Payment Systems  Oracle Internet Intelligence, 3 August 2018

Shutting down the BGP Hijack Factory  Oracle Dyn, 10 July 2018

Suspicious event hijacks Amazon traffic for 2 hours, steals cryptocurrency  Ars Technica, 24 April 2018

Popular Destinations rerouted to Russia  BGPmon, 12 December 2017

Insecure routing redirects YouTube to Pakistan  Ars Technica, 25 February 2008

9.4 IETF Documents

Most of the original work on RPKI standardisation for both origin and path validation was done in the Secure Inter-Domain Routing (sidr) working group. After the work was completed, the working group was concluded.

Since then, the SIDR Operations (sidrops) working group was formed. This working group develops guidelines for the operation of SIDR-aware networks, and provides operational guidance on how to deploy and operate SIDR technologies in existing and new networks.

All relevant drafts and standards can be found in the archives of these two working groups, with a few exceptions, such as draft-ietf-grow-rpki-as-cones.
This section provides an overview of all well known open source projects that support RPKI. It includes Relying Party software for validating RPKI data, Certificate Authority software to run RPKI on your own infrastructure and supporting tools that help deployment and integration.

### 10.1 Relying Party Software

**Fort Validator**  MIT-licensed Relying Party software by NIC.mx, written in C.

**OctoRPKI**  Cloudflare’s Relying Party software, written in the Go programming language.

**rcynic**  “Cynical rsync”, software to fetch and validate RPKI certificates by Dragon Research Labs, written in the Python programming language.

**RIPE NCC RPKI Validator**  Full-featured RPKI relying party software, written by the RIPE NCC in the Java programming language.

**Routinator**  RPKI relying party software written by NLnet Labs in the Rust programming language, designed to have a small footprint and great portability.

**rpki-client(8)**  rpki-client is written in C as part of the rpki-client(1) project, an RPKI validator for OpenBSD.

**RPSTIR**  Relying Party Security Technology for Internet Routing (RPSTIR) software, initially written by Raytheon BBN Technologies in the C programming language, now maintained by ZDNS.

### 10.2 Certificate Authority Software


**rpkid**  RPKI Certificate Authority software by Dragon Research Labs, written in the Python programming language.
10.3 Supporting Tools

**BGP-SRx** SRx is an open source reference implementation and research platform by the National Institute for Standards and Technology (NIST). It is intended for investigating emerging BGP security extensions and supporting protocols such as RPKI Origin Validation and BGPSec Path Validation.

**GoRTR** An open-source implementation of RPKI to Router protocol (RFC 6810) using the Go programming language. This project is maintained by Louis Poinsignon at Cloudflare.

**RTRlib** The RTRlib implements the client-side of the RPKI-RTR protocol (RFC 6810, RFC 8210) and BGP Prefix Origin Validation (RFC 6811). This also enables the maintenance of router keys, which are required to deploy BGPSec.

RTRlib was originally founded by researchers from the Computer Systems & Telematics group at Freie Universität Berlin and researchers from the INET research group at Hamburg University of Applied Sciences, under the supervision of Matthias Wählisch and Thomas Schmidt. It is now a community project.

**pmacct** pmacct is a small set of multi-purpose passive network monitoring tools. It can account, classify, aggregate, replicate and export forwarding-plane data, i.e., IPv4 and IPv6 traffic; collect and correlate control-plane data via BGP and BMP; collect and correlate RPKI data; collect infrastructure data via Streaming Telemetry.

The pmacct toolset can perform RPKI Origin Validation and present the outcome as a property in the flow aggregation process. Because it separates out the various types kinds of (invalid) BGP announcements, operators can a good grasp on how their connectivity to the rest of the Internet would look like after deploying a "invalid == reject" policy.

---

**Warning:** This part of the project is currently being built. Documentation will likely change significantly as the software evolves.
Krill is a free, open source Resource Public Key Infrastructure (RPKI) daemon, featuring a Certificate Authority and Publication Server, written by NLnet Labs in the Rust programming language.

This implementation will allow operators to run their own Certificate Authority (CA) as a child of a Regional Internet Registry or a different parent, such as a National Internet Registry (NIR) or Enterprise. The CA will allow operators to generate and publish their own cryptographic material, including all certificates and ROAs.

The software will support running the CA both upwards and downwards. Upwards means that operators can have multiple parents, such as ARIN, RIPE NCC, etc., simultaneously and transparently. Downwards means that the CA can issue to child organisations or customers who, in turn, run their own CA.

The CA is intended for:

- Operators who require easier RPKI management that is integrated with their own systems in a better way, instead of relying on the web-based user interface that the RIRs offer with the hosted systems
- Operators who are security conscious and require that they are the only ones in possession of the private key of a system they use
- Operators who want to be operationally independent from the parent RIR, such as NIRs or Enterprises

The Publication Server in Krill can also be run as an independent component. This can be used by organisations who want to offer publication of RPKI data as a service. This way, it will allow operators to do the publication of their certificates and ROAs themselves, or let a third party such as a Content Delivery Network do it.

If you want to know more about the project planning, please have a look at the high level roadmap on our website, or get a more detailed overview of the releases on GitHub.

If you have any questions, comments or ideas, you are welcome to discuss them on our RPKI mailing list, or feel free to create an issue on GitHub.

### 11.1 Installation

You can either build Krill from sources with Cargo or run it using Docker, both are quite easy.
11.1.1 Quick Start with Docker

The NLnet Labs provided Docker container for Krill allows you to use Krill just as you would with Cargo but without needing to build from sources first, you only need Docker.

To fetch and run Krill:

```bash
docker run --name krill -p 127.0.0.1:3000:3000 nlnetlabs/krill:v0.1.0
```

With a shell alias interacting with Krill via `krill_admin` is then as easy as:

```bash
$ alias ka='docker exec krill krill_admin \
  -s https://127.0.0.1:3000/ \
  -t <SOME_TOKEN>''

$ ka cas list
{
  "cas": []
}
```

To get the most out of Krill you will want to run Docker and Krill with additional arguments. See *Running Krill* and *Running Krill with Docker* for more information.

11.1.2 Installing with Cargo

There are three things you need for Krill: Rust, a C toolchain and OpenSSL. You can install the Krill on any Operating System where you can fulfil these requirements, but will will assume that you will run this on a UNIX-like OS.

**Rust**

While some system distributions include Rust as system packages, Krill relies on a relatively new version of Rust, currently 1.30 or newer. We therefore suggest to use the canonical Rust installation via a tool called `rustup`.

To install `rustup` and Rust, simply do:

```bash
curl https://sh.rustup.rs -sSf | sh
```

Alternatively, get the file, have a look and then run it manually. Follow the instructions to get rustup and cargo, the rust build tool, into your path.

You can update your Rust installation later by simply running:

```bash
rustup update
```

To get started you need Cargo’s bin directory (`$HOME/.cargo/bin`) in your PATH environment variable. To configure your current shell, run

```bash
source $HOME/.cargo/env
```

**C Toolchain**

Some of the libraries Krill depends on require a C toolchain to be present. Your system probably has some easy way to install the minimum set of packages to build from C sources. For example, `apt install build-essential` will install everything you need on Debian/Ubuntu.
If you are unsure, try to run `cc` on a command line and if there’s a complaint about missing input files, you are probably good to go.

**OpenSSL**

Your system will likely have a package manager that will allow you to install OpenSSL in a few easy steps. For Krill, you will need `libssl-dev`, sometimes called `openssl-dev`. On Debian like Linux distributions, this should be as simple as running:

```
sudo apt-get install -y libssl-dev openssl pkg-config
```

Note: we use Ubuntu xenial (16.04.5 LTS) in our Travis CI environment.

On macOS you can use Homebrew or MacPorts to get started.

**11.1.3 Building**

The easiest way to get Krill is to clone the repository and build it using cargo:

```
git clone git@github.com:NLnetLabs/krill.git
cd krill
```

Now you can build the krill binaries from the Rust source:

```
cargo build --release
```

This will build the following binaries:

```
target/release/krilld
target/release/krill_admin
```

You can copy these binaries to a location of your convenience or run them from this directory.

**11.2 Running Krill**

Krill has an embedded web server, and saves its status on disk as json files. Krill does not depend on any database, but will need to be configured and told where on the system its working directory is.

You can have a look at all possible configuration directives in the packaged version of the default configuration file. This file can be found, relative to where you cloned krill, under: `./daemon/defaults/krill.conf`.

You will notice that this file contains comments only. I.e. it documents default configuration settings. In order to override the Krill config you should create your own `krill.conf` file and use the `--config` directive when you start `krilld`:

```
krilld --config <path-to-config>
```

**11.2.1 Admin Token**

You will need to generate your own secret token (password) before you can run krill. If you do not supply a token, Krill will refuse to start and complain with the following message:
You MUST provide a value for the master API key, either by setting "auth_token" in the config file, or by setting the KRILL_AUTH_TOKEN environment variable.

There is no default token, because we want to avoid that people run with default passwords. So, make up a nice one, and either add it to your config file, or use the ENV variable if you prefer.

### 11.2.2 Proxy and HTTPS

Krill uses HTTPS and refuses to do plain HTTP. In theory Krill should be able to use a key pair and corresponding certificate signed by a web TA. However, this is untested.

By default Krill will generate a 2048 bit RSA key and self-signed certificate when it’s first started.

We recommend, at least for now, that you run Krill with this default, and use a proxy server such as nginx if you intend to make Krill available to the internet. Industry standard proxy servers such as nginx are much better suited to deal with the sometimes-not-so-well-meaning people on the internet, and implement best practices regarding HTTPS.

Also, setting up a widely accepted HTTPS certificate, e.g. through Letsencrypt, is well documented for these servers.

### 11.2.3 Minimal Configuration

Krill uses defaults that are sensible for development. Some of these may also be fine if you are testing Krill locally. However you should review the following.

#### Public IP and Port

We recommend that you keep the following settings, and use a proxy server if you want to expose Krill to other users:

```
# Specify the ip address and port number that the server will use.
#ip = "localhost"
#port = 3000
```

#### Data

By default Krill will expect a directory called data relative to its working directory:

```
# Specify the directory where the publication server will store its data.
# Note that clustering through a shared data directory is not supported.
# But, we plan to look into a proper clustering solution later.
#data_dir = ".:/data"
```

You should probably change this to use an absolute path instead.

Krill will create a number of subdirectories under this data_dir for various purposes:

<table>
<thead>
<tr>
<th>Directory</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>cas</td>
<td>The state of all CAs in this Krill instance.</td>
</tr>
<tr>
<td>ssl</td>
<td>The HTTPS key pair and certificate.</td>
</tr>
<tr>
<td>proxy</td>
<td>The state of remote publishers. (will be deprecated)</td>
</tr>
<tr>
<td>publishers</td>
<td>The state of all publishers in this Krill instance.</td>
</tr>
<tr>
<td>repo-server</td>
<td>The state of the repository server (client info, and published objects)</td>
</tr>
<tr>
<td>repo/rsync/</td>
<td>Published RPKI objects that should be made available through an rsync server.</td>
</tr>
<tr>
<td>repo/rrdp/</td>
<td>Published RPKI objects in RRDP (RFC8182) XML format.</td>
</tr>
</tbody>
</table>
Service and Certificate URIs

The `rsync_base` setting should match the URI of your rsync server, as this is put on any certificates, manifests and ROAs that Krill will create. The `service_uri` setting is used to determine the URI for the RRDP notification file used on certificates, but it’s also used to determine the public URI that will be included in responses to delegated remote child CAs that you may delegate resources to:

```
# Specify the base rsync repository for this server. Publishers will get
# a base URI that is based on the 'publisher_handle' in the XML file.
#
# Note, you should set up an rsync daemon to expose $data_dir/rsync to serve
# this data. The uri defined here should match the module name in your rsync
# configuration.
#rsync_base = "rsync://localhost/repo/"
#
# Specify the base public URI to this service. Other URIs will be derived
# from this:
# <BASE_URI>rrdp/notification.xml (pub point or rrdp)
# <BASE_URI>rrdp/<session>/<version>/snapshot.xml
# <BASE_URI>rrdp/<session>/<version>/delta.xml
# <BASE_URI>ta/ta.cer (on TAL for embedded TA)
# <BASE_URI>rfc6492 (for remote children)
#
# MUST end with a slash.
#service_uri = "http://localhost:3000/"
```

11.2.4 Embedded Trust Anchor

For testing purposes you may want to run Krill with an embedded test Trust Anchor (TA). Using a TA will allow you to create your own test Certificate Authority (CA) and with a locally signed certificate. This is useful when learning how to deploy and use Krill.

To use the embedded TA add the following line to your `krill.conf` file:

```
use_ta = true
```

The Trust Anchor Locator (TAL) for this TA can be retrieved from Krill at: `https://<yourhost>/ta/ta.tal`

You can use this TAL in a Relying Party (RP) tool, such as routinator, to validate the ROAs you create. But, note that no one else will have this TAL, so this is useful for testing only.

At this moment there is no way to disable the embedded TA once it’s created. We may add this later, but for now we recommend that you use this option only on instances that you are prepared to use for testing only.

11.3 Running Krill with Docker

This page explains the additional features and differences compared to running Krill with Cargo that you need to be aware of when running Krill with Docker.

Read *Running Krill* before reading this page.
11.3.1 Get Docker

If you do not already have Docker installed, follow the platform specific installation instructions via the links in the Docker official “Supported platforms” documentation.

11.3.2 Fetching and running Krill

The `docker run` command will automatically fetch the Krill image the first time you use it, and so there is no installation step in the traditional sense.

The `docker run` command can take many arguments and can be a bit overwhelming at first.

The command below runs Krill in the background and shows how to configure a few extra things like log level and volume mounts (more on this below).

```bash
$ docker run -d --name krill -p 127.0.0.1:3000:3000 \\
  -e KRILL_LOG_LEVEL=debug \\
  -e KRILL_FQDN=some.domain \\
  -e KRILL_AUTH_TOKEN=5tT8I7ygoDh9k8I \\
  -v krill_data:/var/krill/data/ \\
  -v /tmp/krill_rsync:/var/krill/data/repo/rsync/ \\
  nlnetlabs/krill:v0.1.0
```

11.3.3 Admin Token

By default Docker Krill secures itself with an automatically generated admin token. You will need to obtain this token from the Docker logs in order to manage Krill via the API or the `krill_admin` CLI tool.

```bash
$ docker logs krill 2>&1 | fgrep token
docker-krill: Securing Krill daemon with token <SOME_TOKEN>
```

You can pre-configure the token via the `auth_token` Krill config file setting, or if you don’t want to provide a config file you can also use the Docker environment variable `KRILL_AUTH_TOKEN` as shown above.

11.3.4 Running the Krill CLI

Local

Using a Bash alias with `<SOME_TOKEN>` you can easily interact with the locally running Krill daemon via its command-line interface (CLI):

```bash
$ alias ka='docker exec krill krill_admin \\n  -s https://127.0.0.1:3000/ \\n  -t <SOME_TOKEN>,'

$ ka cas list
{
  "cas": []
}
```
Remote

The Docker image can also be used to run `krill_admin` to manage remote Krill servers. Using a shell alias simplifies this considerably:

```
$ alias ka_remote='docker run --rm -v /tmp/ka:/tmp/ka nlnetlabs/krill:v0.1.0 krill_admin -s https://<some.domain>/ -t <SOME_TOKEN>'

$ ka_remote cas list
```

Note: The `-v` volume mount is optional, but without it you will not be able to pass files to `krill_admin` which some subcommands require, e.g.

```
$ ka cas roas -h child update -d /tmp/ka/delta.in
```

11.3.5 Proxy and HTTPS

As advised in *Running Krill* you should run Krill behind an industry standard proxy server such as nginx.

Service and Certificate URIs

The Krill `service_uri` and `rsync_base` config file settings can be configured via the Docker environment variable `KRILL_FQDN` as shown in the example above. Providing `KRILL_FQDN` will set both `service_uri` and `rsync_base`.

11.3.6 Data

Krill writes state and data files to a data directory which in Docker Krill is hidden inside the Docker container and is lost when the Docker container is destroyed.

Persistence

To protect the data you can write it to a persistent Docker volume which is preserved even if the Krill Docker container is destroyed. The following fragment from the example above shows how to configure this:

```
docker run -v krill_data:/var/krill/data/
```

Access

Some of the data files written by Krill to its data directory are intended to be shared with external clients via the rsync protocol. To make this possible with Docker Krill you can either:

- Mount the rsync data directory in the host and run rsyncd on the host, OR
- Share the rsync data with another Docker container which runs rsyncd

Mounting the data in a host directory:

```
docker run -v /tmp/krill_rsync:/var/krill/data/repo/rsync
```
Sharing via a named volume:

```
docker run -v krill_rsync:/var/krill/data/repo/rsync
```

### 11.3.7 Logging

Krill logs to a file by default. Docker Krill however logs by default to stderr so that you can see the output using the `docker logs` command.

At the default `warn` log level Krill doesn’t output anything unless there is something to warn about. Docker Krill however comes with some additional logging which appears with the prefix `docker-krill:`. On startup you will see something like the following in the logs:

```
docker-krill: Securing Krill daemon with token ba473bac-021c-4fc9-9946-6ec109befec3
docker-krill: Configuring /var/krill/data/krill.conf ..
docker-krill: Dumping /var/krill/data/krill.conf config file ...
docker-krill: End of dump
```

### 11.3.8 Docker Krill environment variable reference

The Krill Docker image supports the following environment variables which map to the following `krill.conf` settings:

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Equivalent Krill config setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>KRILL_AUTH_TOKEN</td>
<td>auth_token</td>
</tr>
<tr>
<td>KRILL_FQDN</td>
<td>service_uri and rsync_base</td>
</tr>
<tr>
<td>KRILL_LOG_LEVEL</td>
<td>log_level</td>
</tr>
<tr>
<td>KRILL_USE_TA</td>
<td>use_ta</td>
</tr>
</tbody>
</table>

### 11.3.9 Using a config file

Via a volume mount you can replace the Docker Krill config file with your own and take complete control:

```
docker run -v /tmp/krill.conf:/var/krill/data/krill.conf
```

This will instruct Docker to replace the default config file used by Docker Krill with the file `/tmp/krill.conf` on your host computer.

**Warning:** The CLI has grown organically and will most likely see an overhaul before Krill 1.0 is released. Functionality will remain mostly the same, but subcommands and options will change.

### 11.4 Using the Krill CLI

The Krill CLI is a wrapper around the Krill API which is based on Json over HTTPS.
11.4.1 Certificate Authority

The term Certificate Authority (CA) can sometimes be a bit overloaded, so it’s worth defining it better.

In the context of Krill we refer to a CA as unit that represents an organisational unit, e.g. your company. This CA will typically have a single parent Certificate Authority, like the RIR/NIR that you have registered IP addresses and/or AS numbers with. However, you may have multiple parents. It’s also possible to delegate resources down children of your own, e.g. business units, departments, members or clients.

Resources that you receive from each of your parents will each go on separate X509 certificates, and in fact you might even get resources from a single parent assigned to you on different certificates. These certificates are often referred to as “CA certificates”, which can be somewhat confusing with regards to the term CA. A “CA certificate” is simply a certificate that is allowed to sign delegated certificates in the RPKI. And an ‘organisational’ CA, as described above, will typically have one or many CA certificates.

So, in the context of Krill we always talk about ‘organisational’ CAs when we talk about CAs. In fact the main reason of being for Krill is that it let’s you think about your organisation at this higher level, while Krill will deal with the management of lower level CA certificates, and all the other moving parts that are used in the RPKI.

11.4.2 General use

To use the CLI you will need to specify the server, token and format you would like to use:

```bash
krill_admin -s https://<yourhost:port>/ -t <token> -f text|json|none
```

The default format is json, but for human operators this can be suboptimal. We may therefore change the default to ‘text’ in future. We also plan to introduce either some local config file (e.g. ~/.krill_admin.cfg) or ENV variables that allow an operator to set defaults for this in future.

For now though, you will need to specify this whenever you call the CLI, or perhaps create an alias.

11.4.3 Help

To access the help function of the CLI use the word ‘help’, e.g.:

```bash
$ ./target/debug/krill_admin --server https://localhost:3000/ --token secret --format --text help
```

Krill admin client 0.1.0

Usage:

```
krill_admin [OPTIONS] --server <URI> --token <token-string> [SUBCOMMAND]
```

Flags:

- `h, --help`: Prints help information
- `V, --version`: Prints version information

Options:

- `-f, --format <type>`: Specify the report format (none|json|text|xml). If unspecified the format will match the corresponding server api response type.
- `-s, --server <URI>`: Specify the full URI to the krill server.
- `-t, --token <token-string>`: Specify the value of an admin token.

Subcommands:

- `cas`: Manage CAs
- `health`: Perform a health check. Exits with exit code 0 if all is well, exit code 1 in case of any issues
11.4.4 Manage CA(s)

The `cas` subcommand is used to manage ‘organisational’ CAs in Krill:

- Add a CA to Krill
- Add a parent to a CA
- Add a child to a CA
- Manage Route Authorizations in a CA
- View the status of a CA
- Perform a key roll

**Add a CA**

When adding a CA you need to choose a “handle”, essentially just a name. The term “handle” comes from RFC 8183 and is used in the communication protocol between CAs and publication servers.

Furthermore you need to chose a “token”, or password, to manage this specific CA. For the moment though, this “token” is not used in practice. Krill is managed entirely using a single master “token”. However, this may well change in future, especially if operators tell us that they want to support multiple CAs in a single Krill instance, while delegating authority to different users. If this is a use case you have, please talk to us!

Example:

```
$ krill_admin --server https://localhost:3000/ --token secret --format text

cas add --handle child --token child
```

When a CA has been added, it is registered to publish locally in the Krill instance where it exists, but other than that it has no configuration yet. In order to do anything useful with a CA you will first have to add at least one parent to it, and then most likely some Route Authorizations and/or Child CAs.

**List CAs**

You can look at the existing CAs in krill using the following command:

```
$ krill_admin --server https://localhost:3000/ --token secret --format text cas list
ta
child
```

```
$ krill_admin --server https://localhost:3000/ --token secret --format json cas list
{
  "cas": [
    {
      "name": "ta"
    }
  ],

(continues on next page)
Show CA Details

You can use the following to show the details of the embedded TA, if you enabled it:

```
$ krill_admin --server https://localhost:3000/ --token secret --format text \
cas show --handle ta
```

Name: ta
Base uri: rsync://localhost/repo/ta/
RRDP uri: https://localhost:3000/rrdp/notification.xml
Parent: This CA is a TA
Resource Class: 0
State: active
Resources:

<table>
<thead>
<tr>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASNs: AS0-AS4294967295</td>
</tr>
<tr>
<td>IPv4: 0.0.0.0/0</td>
</tr>
<tr>
<td>IPv6: ::/0</td>
</tr>
</tbody>
</table>

Current objects:
CED18E1CDFA208EBA4A400DCBB3B6B7D609BBC8E.mft
CED18E1CDFA208EBA4A400DCBB3B6B7D609BBC8E.crl

Children:
<none>

Or for your new CA:

```
$ krill_admin --server https://localhost:3000/ --token secret --format text \
cas show --handle child
```

Name: child
Base uri: rsync://localhost/repo/child/
RRDP uri: https://localhost:3000/rrdp/notification.xml
Children:
<none>

Add a Child to the embedded TA

If you are using an embedded TA for testing then you will first need to add your new CA “child” to it. Krill supports two communication modes: 1) embedded, meaning the both the parent and child CA live in the same Krill 2) rfc6492, meaning that the official RFC protocol is used

Here we will document the second option. It’s slightly less efficient, but it’s the same as what you would need to delegate from your CA to remote CAs.

Step 1: Get the RFC8183 request XML from your child:
$ krill_admin --server https://localhost:3000/ --token secret --format text cas rfc8183_child_request --handle child > ./request.xml

Step 2: Now give the child request XML to the parent, and get its response:

$ krill_admin --server https://localhost:3000/ --token secret --format text cas children --handle ta add -4 10.0.0.0/8 rfc6492 --xml ./child-request.xml >./parent-response.xml

Step 3: Now add the TA as a parent to child

$ krill_admin --server https://localhost:3000/ --token secret --format text cas update --handle child add-parent --parent ta rfc6492 --xml ./parent-response.xml

Now you should see that your “child” is certified:

$ krill_admin --server https://localhost:3000/ --token secret --format text cas show --handle child

Name: child
Base uri: rsync://localhost/repo/child/
RRDP uri: https://localhost:3000/rrdp/notification.xml
Parent: RFC 6492 Parent
Resource Class: 0
State: active
Resources:
  ASNs:
  IPv4: 10.0.0.0/8
  IPv6:
Current objects:
  2E0AD62800126C5D1F6388CD89B540F24B28C0ED.mft
  2E0AD62800126C5D1F6388CD89B540F24B28C0ED.crl

Children:
  <none>

Add a real CA as your parent

Similar to above, except that you only need to generate the XML in step 1, hand it over to your parent CA through whatever function they provide, and then get the response.xml from them and add it your child as described in step 3.

ROAs

At this point you probably want to manage some ROAs!

Krill lets users configure Route Authorizations, i.e. the intent to authorise a Prefix you hold, up to a maximum length to be announced by an ASN. Krill will make sure that the actual ROA objects are created. Krill will also refuse to accept authorizations for prefixes you don’t hold.

Update ROAs

We will add Json support later, but for the moment Krill expects ROA updates in the following file format:
A: 192.168.1.0/24 => 64496
A: 192.168.0.0/16-20 => 64496  # Add prefix with max length
R: 192.168.3.0/24 => 64496  # Remove existing authorization

You can then add this to your CA:

```
$ krill_admin --server https://localhost:3000/ --token secret --format text cas \ 
roas --handle child update --delta ./data/authorise.txt
```

**List Route Authorizations**

You can list Route Authorizations as well:

```
$ krill_admin --server https://localhost:3000/ --token secret --format text cas \ 
roas --handle child list
10.0.0.0/20-24 => 64496
10.1.0.0/24 => 64496
```
Routinator is free, open source RPKI Relying Party software written by NLnet Labs in the Rust programming language.

The application is designed to be lightweight and have great portability. This means it can run on any Unix-like operating system, but also works on Microsoft Windows. Due to its lean design, it can run effortlessly on minimalist hardware such as a Raspberry Pi.

Routinator connects to the Trust Anchors of the five Regional Internet Registries (RIRs) — APNIC, AFRINIC, ARIN, LACNIC and RIPE NCC — downloads all of the certificates and ROAs in the various repositories, verifies the signatures and makes the result available for use in the BGP workflow. It can perform RPKI validation as a one-time operation and store the result on disk in formats such as CSV, JSON and RPSL, or run as a service that periodically fetches and verifies RPKI data. The data is then served via the built-in HTTP server, or fetched from RPKI-capable routers via the RPKI-RTR protocol.

If you run into a problem with Routinator or you have a feature request, please create an issue on Github. We are also happy to accept your pull requests. For general discussion and exchanging operational experiences we provide a mailing list. This is also the place where we will announce releases of the application and updates on the project.

You can follow the adventures of Routinator on Twitter and listen to its favourite songs on Spotify.

12.1 Installation

Getting started with Routinator is really easy either building from Cargo or running with Docker.

12.1.1 Quick Start

Assuming you have rsync and the C toolchain but not yet Rust 1.34 or newer, here’s how you get Routinator to run as an RTR server listening on 192.0.2.13 port 3323 and an HTTP server listening on port 9556:

```
curl https://sh.rustup.rs -sSf | sh
source ~/.cargo/env
```
cargo install routinator
routinator init
# Follow instructions provided
routinator server --rtr 192.0.2.13:3323 --http 192.0.2.13:9556

If you have an older version of Routinator, you can update via:

```
cargo install -f routinator
```

If you want to try the master branch from the repository instead of a release version, you can run:

```
cargo install --git https://github.com/NLnetLabs/routinator.git
```

### 12.1.2 Quick Start with Docker

Due to the impracticality of complying with the ARIN TAL distribution terms in an unsupervised Docker environment, before launching the container it is necessary to first review and agree to the ARIN Relying Party Agreement (RPA). If you agree to the terms, you can let the Routinator Docker image install the TALs into a mounted volume that is later reused for the server:

```
# Create a local directory for the RPKI cache
sudo mkdir -p /etc/routinator/tals
# Review the ARIN terms.
# Run a disposable container to install TALs.
sudo docker run --rm -v /etc/routinator/tals:/root/.rpki-cache/tals \
  nlnetlabs/routinator init -f --accept-arin-rpa
# Launch the final detached container named 'routinator' exposing RTR on 
# port 3323 and HTTP on port 9556
sudo docker run -d --name routinator -p 3323:3323 -p 9556:9556 \
  -v /etc/routinator/tals:/root/.rpki-cache/tals nlnetlabs/routinator
```

### 12.1.3 System Requirements

At this time, the size of the global RPKI data set is about 300MB. Cryptographic validation of it takes Routinator about 2 seconds on a quad-core i7.

When choosing a system to run Routinator on, make sure you have 512MB of available memory and 1GB of disk space. This will give you ample margin for the RPKI repositories to grow over time, as adoption increases.

### 12.1.4 Getting Started

There are three things you need to install and run Routinator: rsync, a C toolchain and Rust. You can install Routinator on any system where you can fulfil these requirements.

You need rsync because most RPKI repositories currently use it as its main means of distribution. Some of the cryptographic primitives used by Routinator require a C toolchain. Lastly, you need Rust because that’s the programming language that Routinator has been written in.
rsync

Currently, Routinator requires the rsync executable to be in your path. Due to the nature of rsync, it is unclear which particular version you need at the very least, but whatever is being shipped with current Linux and *BSD distributions and macOS should be fine. Alternatively, you can download rsync from its website.

On Windows, Routinator requires the rsync version that comes with Cygwin – make sure to select rsync during the installation phase.

C Toolchain

Some of the libraries Routinator depends on require a C toolchain to be present. Your system probably has some easy way to install the minimum set of packages to build from C sources. For example, apt install build-essential will install everything you need on Debian/Ubuntu.

If you are unsure, try to run cc on a command line and if there’s a complaint about missing input files, you are probably good to go.

Rust

The Rust compiler runs on, and compiles to, a great number of platforms, though not all of them are equally supported. The official Rust Platform Support page provides an overview of the various support levels.

While some system distributions include Rust as system packages, Routinator relies on a relatively new version of Rust, currently 1.34 or newer. We therefore suggest to use the canonical Rust installation via a tool called rustup.

To install rustup and Rust, simply do:

```
curl https://sh.rustup.rs -sSf | sh
```

Alternatively, visit the official Rust website for other installation methods.

You can update your Rust installation later by running:

```
rustup update
```

For some platforms, rustup cannot provide binary releases to install directly. The Rust Platform Support page lists several platforms where official binary releases are not available, but Rust is still guaranteed to build. For these platforms, automated tests are not run so it’s not guaranteed to produce a working build, but they often work to quite a good degree.

One such example that is especially relevant for the routing community is OpenBSD. On this platform, patches are required to get Rust running correctly, but these are well maintained and offer the latest version of Rust quite quickly.

Rust can be installed on OpenBSD by running:

```
pkg_add rust
```

Another example where the standard installation method does not work is CentOS 6, where you will end up with a long list of error messages about missing assembler instructions. This is because the assembler shipped with CentOS 6 is too old.

You can get the necessary version by installing the Developer Toolset 6 from the Software Collections repository. On a virgin system, you can install Rust using these steps:
12.1.5 Building

The easiest way to get Routinator is to leave it to cargo by saying:

```
cargo install routinator
```

If you want to try the master branch from the repository instead of a release version, you can run:

```
cargo install --git https://github.com/NLnetLabs/routinator.git
```

If you want to update an installed version, you run the same command but add the `-f` flag, a.k.a. force, to approve overwriting the installed version.

The command will build Routinator and install it in the same directory that cargo itself lives in, likely $HOME/.cargo/bin. This means Routinator will be in your path, too.

Building a Statically Linked Routinator

While Rust binaries are mostly statically linked, they depend on libc which, as least as glibc that is standard on Linux systems, is somewhat difficult to link statically. This is why Routinator binaries are actually dynamically linked on glibc systems and can only be transferred between systems with the same glibc versions.

However, Rust can build binaries based on the alternative implementation named musl that can easily be statically linked. Building such binaries is easy with rustup. You need to install musl and the correct musl target such as x86_64-unknown-linux-musl for x86_64 Linux systems. Then you can just build Routinator for that target.

On a Debian (and presumably Ubuntu) system, enter the following:

```
sudo apt-get install musl-tools
rustup target add x86_64-unknown-linux-musl
cargo build --target=x86_64-unknown-linux-musl --release
```

12.2 Initialisation

Before running Routinator for the first time, you must prepare its working environment. You do this using the `init` command. This will prepare both the directory for the local RPKI cache, as well as the Trust Anchor Locator (TAL) directory.

By default, both directories will be located under $HOME/.rpki-cache, but you can change their locations via the command line options `--repository-dir` and `--tal-dir`.

TALs provide hints for the trust anchor certificates to be used both to discover and validate all RPKI content. The five TALs — one for each Regional Internet Registry (RIR) — are bundled with Routinator and installed by the `init` command.
**Warning:** Using the TAL from ARIN, the RIR for the United States, Canada as well as many Caribbean and North Atlantic islands, requires you to read and accept their *Relying Party Agreement* before you can use it. Running the *init* command will provide you with instructions.

```
routinator init
Before we can install the ARIN TAL, you must have read
and agree to the ARIN Relying Party Agreement (RPA).
It is available at
https://www.arin.net/resources/manage/rpki/rpa.pdf
If you agree to the RPA, please run the command
again with the --accept-arin-rpa option.
```

Running the *init* command with the --accept-arin-rpa option will create the TAL directory and copy the five Trust Anchor Locator files into it.

```
routinator init --accept-arin-rpa
```

If you decide you cannot agree to the ARIN RPA terms, the --decline-arin-rpa option will install all TALs except the one for ARIN. If, at a later point, you wish to use the ARIN TAL anyway, you can add it to your current installation using the -f flag, to force the installation of all TALs.

### 12.2.1 Performing a Test Run

To see if Routinator has been initialised correctly and your firewall allows the required connections, it is recommended to perform an initial test run. You can do this by having Routinator print a validated ROA payload (VRP) list with the *vrps* sub-command, and using -v to increase the log level to INFO to see if Routinator establishes rsync connections as expected.

```
routinator -v vrps
```

Now, you can see how Routinator connects to the RPKI trust anchors, downloads the the contents of the repositories to your machine, validates it and produces a list of validated ROA payloads in the default CSV format to standard output. From a cold start, this process will take about two minutes.

```
routinator -v vrps
rsyncing from rsync://repository.lacnic.net/rpki/.
rsyncing from rsync://rpki.afrinic.net/repository/.
rsyncing from rsync://rpki.apnic.net/repository/.
rsyncing from rsync://rpki.ripe.net/ta/.
rsync://rpki.ripe.net/ta: The RIPE NCC Certification Repository is subject to Terms and Conditions
rsync://rpki.ripe.net/ta: See http://www.ripe.net/lir-services/ncc/legal/certification/repository-tc
rsync://rpki.ripe.net/ta:
rsyncing from rsync://rpki.ripe.net/repository/.
Found valid trust anchor rsync://rpki.afrinic.net/repository/AfriNIC.cer. Processing.
rsyncing from rsync://rpki.arin.net/repository/.
Found valid trust anchor rsync://rpki.arin.net/repository/arin-rpki-ta.cer. Processing.
```

(continues on next page)


12.3 Running Interactively

Routinator can perform RPKI validation as a one-time operation and print a Validated ROA Payload (VRP) list in various formats, or it can return the validity of a specific announcement. These functions are accessible on the command line via the following sub-commands:

- **vrps** Fetches RPKI data and produces a Validated ROA Payload (VRP) list in the specified format.
- **validate** Outputs the RPKI validity for a specific announcement by supplying Routinator with an ASN and a prefix.

12.3.1 Printing a List of VRPs

Routinator can produce a Validated ROA Payload (VRP) list in five different formats, which are either printed to standard output or saved to a file:

- **csv** The list is formatted as lines of comma-separated values of the prefix in slash notation, the maximum prefix length, the autonomous system number, and an abbreviation for the trust anchor the entry is derived from. The latter is the name of the TAL file without the extension .tal. This is the default format used if the **--format** or **-f** option is missing.
- **csvext** This is an extended version of the **csv** format, which was used by the RIPE NCC RPKI Validator 1.x. Each line contains these comma-separated values: the rsync URI of the ROA the line is taken from (or “N/A” if it isn’t from a ROA), the autonomous system number, the prefix in slash notation, the maximum prefix length, and lastly the not-before and not-after date of the validity of the ROA.
- **json** The list is placed into a JSON object with a single element **roas** which contains an array of objects with four elements each: The autonomous system number of the network authorised to originate a
prefix in \textit{asn}, the prefix in slash notation in \textit{prefix}, the maximum prefix length of the announced route in \textit{maxLength}, and the trust anchor from which the authorisation was derived in \textit{ta}. This format is identical to that produced by the RIPE NCC Validator except for different naming of the trust anchor. Routinator uses the name of the TAL file without the extension \textit{.tal} whereas the RIPE NCC Validator has a dedicated name for each.

\textbf{openbgpd} Choosing this format causes Routinator to produce a \textit{roa-set} configuration item for the OpenBGPD configuration.

\textbf{rpsl} This format produces a list of RPSL objects with the authorisation in the fields \textit{route}, \textit{origin}, and \textit{source}. In addition, the fields \textit{descr}, \textit{mnt-by}, \textit{created}, and \textit{last-modified}, are present with more or less meaningful values.

\textbf{summary} This format produces a summary of the content of the RPKI repository. For each trust anchor, it will print the number of verified ROAs and VRPs. Note that this format does not take filters into account. It will always provide numbers for the complete repository.

For example, to get the validated ROA payloads in CSV format, run:

\begin{verbatim}
routinator vrps --format csv
ASN,IP Prefix,Max Length,Trust Anchor
AS55803,103.14.64.0/23,23,apnic
AS267868,45.176.192.0/24,24,lacnic
AS41152,82.115.18.0/23,23,ripe
AS28920,185.103.228.0/22,22,ripe
AS11845,209.203.0.0/18,24,afrinic
AS63297,23.179.0.0/24,24,arin
...
\end{verbatim}

To generate a file with with the validated ROA payloads in JSON format, run:

\begin{verbatim}
routinator vrps --format json --output authorisedroutes.json
\end{verbatim}

\section*{Filtering}

In case you are looking for specific information in the output, Routinator allows filtering to see if a prefix or ASN is covered or matched by a VRP. You can do this using the \textit{--filter-asn} and \textit{--filter-prefix} flags.

When using \textit{--filter-asn}, you can use both AS64511 and 64511 as the notation. With \textit{--filter-prefix}, the result will include VRPs regardless of their ASN and MaxLength. Both filter flags can be combined and used multiple times in a single query and will be treated as a logical \textit{"or"}.

A validation run will be started before returning the result, making sure you get the latest information. If you would like a result from the current cache, you can use the \textit{--noupdate} or \textit{--n} flag.

Here are some examples filtering for an ASN and prefix in CSV and JSON format:

\begin{verbatim}
routinator vrps --format csv --filter-asn 196615
ASN,IP Prefix,Max Length,Trust Anchor
AS196615,2001:7fb:fd03::/48,48,ripe
AS196615,93.175.147.0/24,24,ripe
\end{verbatim}

\begin{verbatim}
routinator vrps --format json --filter-prefix 93.175.146.0/24
{
 "roas": [
  { "asn": "AS12654", "prefix": "93.175.146.0/24", "maxLength": 24, "ta": "ripe" }
 ]
}
\end{verbatim}
12.3.2 Validity Checker

You can check the RPKI origin validation status of a specific BGP announcement using the `validate` subcommand and by supplying the ASN and prefix. A validation run will be started before returning the result, making sure you get the latest information. If you would like a result from the current cache, you can use the `--noupdate` or `-n` flag.

```
routinator validate --asn 12654 --prefix 93.175.147.0/24
Invalid
```

A detailed analysis of the reasoning behind the validation outcome is printed in JSON format. In case of an Invalid state, whether this because the announcement is originated by an unauthorised AS, or if the prefix is more specific than the maximum prefix length allows. Lastly, a complete list of VRPs that caused the result is included.

```
routinator validate --json --asn 12654 --prefix 93.175.147.0/24
{
  "validated_route": {
    "route": {
      "origin_asn": "AS12654",
      "prefix": "93.175.147.0/24"
    },
    "validity": {
      "state": "Invalid",
      "reason": "as",
      "description": "At least one VRP Covers the Route Prefix, but no VRP ASN matches the route origin ASN",
      "VRPs": {
        "matched": [],
        "unmatched_as": [
          {
            "asn": "AS196615",
            "prefix": "93.175.147.0/24",
            "max_length": "24"
          }
        ],
        "unmatched_length": [
          {}
        ]
      }
    }
  }
}
```

If you run the HTTP service in daemon mode, this information is also available at the `/validity` endpoint.

12.4 Running as a Daemon

Routinator can run as a service that periodically fetches RPKI data, verifies it and makes the resulting data set available via the RPKI-RTR protocol and through the built-in HTTP server. You can start the Routinator service using the `server` sub-command.

12.4.1 The HTTP Service

The CSV, JSON, OpenBGPD and RPSL formats that Routinator can produce in interactive mode are available via HTTP if the application is running as a service. You can also check the RPKI origin validation status of a specific
The RPKI Documentation

BGP announcement at the /validity endpoint by supplying the ASN and prefix.

The HTTP server is not enabled by default for security reasons, nor does it have a default host or port. In order to start the HTTP server at 192.0.2.13 and 2001:0DB8::13 on port 8323, run this command:

```
routinator server --http 192.0.2.13:8323 --http [2001:0DB8::13]:8323
```

The application will stay attached to your terminal unless you provide the -d (for daemon) option. After fetching and validating the data set, the following paths are available:

- `/csv` Returns the current set of VRPs in csv output format
- `/json` Returns the current set of VRPs in json output format
- `/openbgpd` Returns the current set of VRPs in openbgpd output format
- `/rpsl` Returns the current set of VRPs in rpsl output format
- `/validity` Returns the RPKI origin validation status of a specific BGP announcement by supplying the ASN and prefix in the path, e.g. `/validity?asn=12654&prefix=93.175.147.0/24`

Please note that this server is intended to run on your internal network and doesn’t offer HTTPS natively. If this is a requirement, you can for example run Routinator behind an nginx reverse proxy.

Lastly, the HTTP server provides paths that allow you to monitor Routinator itself and the data it processes, so it may be desirable to have HTTP running alongside the RTR server. For more information, please refer to the Monitoring section.

12.4.2 The RTR Service

Routinator supports RPKI-RTR as specified in RFC 8210 as well as the older version described in RFC 6810.

When launched as an RTR server, routers with support for route origin validation (ROV) can connect to Routinator to fetch the processed data. This includes hardware routers such as Juniper, Cisco and Nokia, as well as software solutions like BIRD, GoBGP and others. The processed data is also available in a number of useful output formats, such as CSV, JSON, RPSL and a format specifically for OpenBGPD.

Like the HTTP server, the RTR server is not started by default, nor does it have a default host or port. Thus, in order to start the RTR server at 192.0.2.13 and 2001:0DB8::13 on port 3323, run this command:

```
routinator server --rtr 192.0.2.13:3323 --rtr [2001:0DB8::13]:3323
```

Please note that port 3323 is not the IANA-assigned default port for the protocol, which would be 323. But as this is a privileged port, you would need to be running Routinator as root when otherwise there is no reason to do that. The application will stay attached to your terminal unless you provide the -d (for daemon) option.

By default, the repository will be updated and re-validated every hour as per the recommendation in the RFC. You can change this via the --refresh option and specify the interval between re-validations in seconds. That is, if you rather have Routinator validate every 15 minutes, the above command becomes:

```
routinator server --rtr 192.0.2.13:3323 --rtr [2001:0DB8::13]:3323 --refresh=900
```

Communication between Routinator and the router using the RPKI-RTR protocol is done via plain TCP. Below, there is an explanation how to secure the transport using either SSH or TLS.

**Secure Transports**

These instructions were contributed by wk on Github.
RFC6810 defines a number of secure transports for RPKI-RTR that can be used to secure communication between a router and a RPKI relying party.

However, the RPKI Router Implementation Report documented in RFC7128 suggests these secure transports have not been widely implemented. Implementations, however, do exist, and a secure transport could be valuable in situations where the RPKI relying party is provided as a public service, or across a non-trusted network.

**SSH Transport**

SSH transport for RPKI-RTR can be configured with the help of netcat and OpenSSH.

Begin by installing the openssh-server and netcat packages.

Make sure Routinator is running as an RTR server on localhost:

```
routinator server --rtr 127.0.0.1:3323
```

Create a username and a password for the router to log into the host with, such as rpki.

Configure OpenSSH to expose an rpki-rtr subsystem that acts as a proxy into Routinator by editing the `/etc/ssh/sshd_config` file or equivalent to include the following line:

```
# Define an `rpki-rtr` subsystem which is actually `netcat` used to proxy STDIN/STDOUT to a running `routinator rtrd -a -l 127.0.0.1:3323`
Subsystem rpki-rtr /bin/nc 127.0.0.1 3323
```

Certain routers may use old KEX algos and Ciphers which are no longer enabled by default.

These examples are required in IOS-XR 5.3 but no longer enabled by default in OpenSSH 7.3.

Ciphers +3des-cbc
KexAlgorithms +diffie-hellman-group1-shal

Restart the OpenSSH server daemon.

An example router-side configuration for a device running IOS-XR:

```
router bgp 65534
  rpki server 192.168.0.100
  username rpki
  password rpki
  transport ssh port 22
```

**TLS Transport**

TLS transport for RPKI-RTR can be configured with the help of stunnel.

Begin by installing the stunnel package.

Make sure Routinator is running as an RTR server on localhost:

```
routinator server --rtr 127.0.0.1:3323
```

Acquire (via for example letsencrypt) or generate an SSL certificate. In the example below, an SSL certificate for the domain example.com generated by letsencrypt is used.

Create a stunnel configuration file by editing `/etc/stunnel/rpki.conf` or equivalent:
[rpki]
; Use a letsencrypt certificate for example.com
     cert = /etc/letsencrypt/live/example.com/fullchain.pem
     key = /etc/letsencrypt/live/example.com/privkey.pem
; Listen for TLS rpki-rtr on port 323 and proxy to port 3323 on localhost
     accept = 323
     connect = 127.0.0.1:3323

Restart stunnel to complete the process.

### 12.5 Configuration

Routinator has a number of default settings, such as the location where files are stored, the refresh interval, and the log level. You can view these settings by running:

```bash
routinator config
```

It will return the list of defaults in the same notation that is used by the optional configuration file, which will be largely similar to this:

```toml
exceptions = []
expire = 7200
history-size = 10
http-listen = []
log = "default"
log-level = "WARN"
refresh = 3600
repository-dir = "/Users/me/.rpki-cache/repository"
retry = 600
rsync-command = "rsync"
rsync-count = 4
rsync-timeout = 600
rtr-listen = []
strict = false
syslog-facility = "daemon"
systemd-listen = false
tal-dir = "/Users/me/.rpki-cache/tals"
validation-threads = 4
```

You can override these defaults, as well as configure a great number of additional options using either command line arguments or via the configuration file.

To get an overview of all available options, please refer to the manual page, which can be viewed by running `routinator man`. Alternatively, the manual page is also available on the NLnet Labs website.

#### 12.5.1 Using a Configuration File

Routinator can take its configuration from a file. You can specify such a config file via the `-c` option. If you don’t, Routinator will check if there is a file `$HOME/.routinator.conf` and if it exists, use it. If it doesn’t exist and there is no `-c` option, the default values are used.

For specifying configuration options, Routinator uses a TOML file. Its entries are named similarly to the command line options. A complete sample configuration file showing all the default values can be found in the repository at `etc/routinator.conf`.
For example, if you want Routinator to refresh every 15 minutes and run as an RTR server on 192.0.2.13 and 2001:0DB8::13 on port 3323, in addition to providing an HTTP server on port 9556, simply take the output from `routinator config` and change the `refresh`, `rtr-listen` and `http-listen` values in your favourite text editor:

```plaintext
exceptions = []
expire = 7200
history-size = 10
http-listen = ["192.0.2.13:9556", "[2001:0DB8::13]:9556"]
log = "default"
log-level = "WARN"
refresh = 900
repository-dir = "/Users/me/.rpki-cache/repository"
retry = 600
rsync-command = "rsync"
rsync-count = 4
rsync-timeout = 600
rtr-listen = ["192.0.2.13:3323", "[2001:0DB8::13]:3323"]
strict = false
syslog-facility = "daemon"
systemd-listen = false
tal-dir = "/Users/me/.rpki-cache/tals"
validation-threads = 4
```

After saving this file as `.routinator.conf` in your home directory, you can start Routinator with:

```
routinator server
```

### 12.5.2 Applying Local Exceptions

In some cases, you may want to override the global RPKI data set with your own local exceptions. For example, when a legitimate route announcement is inadvertently flagged as `invalid` due to a misconfigured ROA, you may want to temporarily accept it to give the operators an opportunity to resolve the issue.

You can do this by specifying route origins that should be filtered out of the output, as well as origins that should be added, in a file using JSON notation according to the SLURM standard specified in RFC 8416.

A full example file is provided below. This, along with an empty one is available in the repository at `/test/slurm`.

```json
{
    "slurmVersion": 1,
    "validationOutputFilters": {
        "prefixFilters": [
            {
                "prefix": "192.0.2.0/24",
                "comment": "All VRPs encompassed by prefix"
            },
            {
                "asn": 64496,
                "comment": "All VRPs matching ASN"
            },
            {
                "prefix": "198.51.100.0/24",
                "asn": 64497,
                "comment": "All VRPs encompassed by prefix, matching ASN"
            }
        ]
    }
}
```

(continues on next page)
Use the \texttt{-x} option to refer to your file with local exceptions. Routinator will re-read that file on every validation run, so you can simply update the file whenever your exceptions change.

\section*{12.6 Monitoring}

The HTTP server in Routinator provides endpoints for monitoring the application. A data format specifically for Prometheus is available, as well as dedicated port 9556.

This means it may be a good idea to run the HTTP server alongside the RTR server. To launch Routinator in server mode on 192.0.2.13 with RTR running on port 3323 and HTTP on 9556, use the following command:

\begin{verbatim}
routinator server --rtr 192.0.2.13:3323 --http 192.0.2.13:9556
\end{verbatim}
On the `/metrics` path, Routinator will expose the number of valid ROAs seen for each trust anchor, as well as the total number of validated ROA payloads (VRPs) for each.

In addition, several counters are available that indicate when the last update was started, when it finished and what the duration was. This will allow you to trigger alerts, for example when the update duration is taking longer than your refresh interval.

The current serial number for RPKI-RTR is also exposed. This number is used to notify connected routers that new data is available. The number Routinator has should match the serial on your connected router. You can verify this on your router with using the following command:

- **Juniper** `show validation session detail`
- **Cisco** `show ip bgp rpki server`
- **Nokia** `show router origin-validation rpki-session detail`

Lastly, the endpoint provides several gauges related to rsync, such as the exit code and the duration of fetching the data in each RPKI repository.

This is an example of the output of the `/metrics` endpoint:

```
# HELP routinator_valid_roas number of valid ROAs seen
# TYPE routinator_valid_roas gauge
routinator_valid_roas{tal="afrinic"} 338
routinator_valid_roas{tal="lacnic"} 2435
routinator_valid_roas{tal="apnic"} 3187
routinator_valid_roas{tal="ripe"} 10779
routinator_valid_roas{tal="arin"} 4964

# HELP routinator_vrps_total total number of VRPs seen
# TYPE routinator_vrps_total gauge
routinator_vrps_total{tal="afrinic"} 459
routinator_vrps_total{tal="lacnic"} 7042
routinator_vrps_total{tal="apnic"} 21941
routinator_vrps_total{tal="ripe"} 56909
routinator_vrps_total{tal="arin"} 6621

# HELP routinator_last_update_start seconds since last update started
# TYPE routinator_last_update_start gauge
routinator_last_update_start 568

# HELP routinator_last_update_duration duration in seconds of last update
# TYPE routinator_last_update_duration gauge
routinator_last_update_duration 47

# HELP routinator_last_update_done seconds since last update finished
# TYPE routinator_last_update_done gauge
routinator_last_update_done 520

# HELP routinator_serial current RTR serial number
# TYPE routinator_serial gauge
routinator_serial 42

# HELP routinator_rsync_status exit status of rsync command
# TYPE routinator_rsync_status gauge
routinator_rsync_status{uri="/rsync://rpki-repository.nic.ad.jp/ap/"} 0
routinator_rsync_status{uri="/rsync://rpki.apnic.net/repository/"} 0
routinator_rsync_status{uri="/rsync://rpki.ca.twnic.tw/rpki/"} 0
routinator_rsync_status{uri="/rsync://ca.rg.net/rpki/"} 0
```

(continues on next page)
The RPKI Documentation

The HTTP service has two additional endpoints on the following paths:

/status  Returns the information from the /metrics endpoint in a more concise format

/version  Returns the version of the Routinator instance
This is the user handbook of the RTRlib. It provides guidance on how to use the library for development and gives an overview of some command line tools that are based on RTRlib. Further information can be found on the RTRlib website\(^1\) and its source code repository on Github\(^2\).

13.1 About

13.1.1 In a Nutshell

RTRlib is a C library that implements the client side of the RPKI-RTR protocol as well as route origin validation. Basically, it maintains data from an RPKI cache server (e.g., Routinator) and allows to verify whether an autonomous system (AS) is the legitimate origin AS, based on the fetched valid ROA data. It is prepared for BGPsec path validation.

RTRlib powers RPKI in BGP software routers such as FRR and is the base for monitoring tools. A Python binding is available. The basis RTRlib package includes the library and lightweight command line tools.

13.1.2 Why do I need the RTRlib?

RTRlib gives easy and highly efficient access to cryptographically valid RPKI data without relying on a specific cache server or RPKI validator implementation. To prevent single point of failures, it handles failover between multiple cache servers.

Not only developers of routing software but also network operators benefit from RTRlib. Developers can integrate the RTRlib into their BGP daemon to extend their implementation towards RPKI. Network operators may use the RTRlib to develop monitoring tools (e.g., to evaluate the performance of caches or to validate BGP data).

\(^1\) Project website - https://rtrlib.realmv6.org
\(^2\) Source code on Github - https://github.com/rtrlib/rtrlib
13.1.3 License

This software is free, open source and licensed under MIT.

13.1.4 Supported RFCs

The current version implements RFC 6811 and RFC 8210.

13.1.5 Community

If you run into a problem with RTRlib or you have a feature request, please create an issue on Github. We are also happy to accept your pull requests. For general discussion and exchanging operational experiences we provide a mailing list. More details about RTRlib are available on the project website.

13.2 Installation

Most Linux distributions as well as Apple macOS support RTRlib. The RTRlib software package includes the library and basic ready-to-use command line tools that show some of the RTRlib features.

13.2.1 Apple macOS

For macOS we provide a Homebrew tap to easily install the RTRlib. First, setup Homebrew¹ and then install the RTRlib package:

```bash
brew tap rtrlib/pils
brew install rtrlib
```

13.2.2 Archlinux

For Archlinux we maintain two PKGBUILDs in the Archlinux User Repository, rtrlib² and rtrlib-git³. rtrlib includes the latest official RTRlib release. rtrlib-git includes the current git master.

You can either use your favourite aur helper or execute the following commands:

```bash
sudo pacman --needed base-devel

# for the latest release
wget https://aur.archlinux.org/cgit/aur.git/snapshot/rtrlib.tar.gz
tar xf rtrlib
cd rtrlib

# for the git version
wget https://aur.archlinux.org/cgit/aur.git/snapshot/rtrlib-git.tar.gz
tar xf rtrlib-git
cd rtrlib-git
```

¹ Homebrew – http://brew.sh
² https://aur.archlinux.org/packages/rtrlib/
³ https://aur.archlinux.org/packages/rtrlib-git/
13.2.3 Debian

RTRlib is part of the official Debian package repository since Buster and can be installed using apt. The following packages are available:

- **librtr0**: includes the basis library.
- **librtr0-dev**: includes header files etc. for developers.
- **rtr-tools**: includes basic command line tools based on RTRlib.
- **librtr0-dbgsym**: includes debugging symbols.
- **librtr-doc**: includes offline documentation.

To install the minimal set of packages required for development, execute the following command:

```
apt install librtr0 librtr-dev
```

If you just want to use the RTRlib command line tools, run

```
apt install librtr0 rtr-tools
```

13.2.4 Gentoo

The FRR routing project maintains a gentoo overlay that contains an ebuild for the RTRlib. First, setup layman, then install rtrlib with the following commands:

```
# If this doe not work try layman -f
layman -a frr-gentoo
emerge rtrlib
```

13.2.5 From Source

The source code repository of RTRlib includes everything that you need to implement or run applications based on the RTRlib, and to use the RTRlib command line tools.

The RTRlib source code consists of the following subdirectories:

- **cmake/**: CMake modules
- **doxygen/**: Example code and graphics used in the Doxygen documentation
- **rtrlib/**: Header and source code files of the RTRlib
- **tests/**: Function tests and unit tests
- **tools/**: Contains rtrclient and rpki-rov

---

4 Buster is currently in testing and scheduled for release Mid 2019.
5 https://github.com/FRRouting/gentoo-overlay
6 https://wiki.gentoo.org/wiki/Layman
The RPKI Documentation

Getting Started

To build and install the RTRlib from source, you need the following common software:

- cmake version >= 2.6 to build the system.
- libssh version >= 0.5.0 to establish SSH transport connections (optional but highly recommended).

Additional optional requirements are:

- cmocka to run RTRlib unit tests
- doxygen to build the RTRlib API documentation

Building

The easiest way to get the source code is to download either the latest RTRlib release from [https://github.com/rtrlib/rtrlib/releases/latest](https://github.com/rtrlib/rtrlib/releases/latest) or the current master from [https://github.com/rtrlib/rtrlib/archive/master.zip](https://github.com/rtrlib/rtrlib/archive/master.zip), and then unpack:

```
unzip rtrlib-master.zip
cd rtrlib-master
# or alternatively, clone the current git master
git clone https://github.com/rtrlib/rtrlib/
cd rtrlib
```

Then, build the library and command line tools using cmake. We recommend an out-of-source build:

```
# inside the main RTRlib source code directory
mkdir build && cd build
cmake -D CMAKE_BUILD_TYPE=Release ../
make
sudo make install
```

To enable debug symbols and messages, change the cmake command to:

```
cmake -D CMAKE_BUILD_TYPE=Debug ../
```

If the build command fails with any error, please consult the RTRlib README\(^7\) and Wiki\(^8\), you may also join our mailing list\(^9\) or open an issue on Github\(^10\).

Additional cmake Options and Targets

If you did not install libssh in the default directories, you can run cmake with the following parameters:

```
-D LIBSSH_LIBRARY=<path-to-libssh.so>
-D LIBSSH_INCLUDE=<include-directory>
```

To configure explicitly a directory where to place the RTRlib during installation, you can pass the following argument to cmake:

```
-D CMAKE_INSTALL_PREFIX=<path>
```

---

\(^7\) README – [https://github.com/rtrlib/rtrlib/blob/master/README](https://github.com/rtrlib/rtrlib/blob/master/README)
\(^8\) Wiki – [https://github.com/rtrlib/rtrlib/wiki](https://github.com/rtrlib/rtrlib/wiki)
\(^9\) Mailing list – [https://groups.google.com/forum/#!forum/rtrlib](https://groups.google.com/forum/#!forum/rtrlib)
\(^10\) Issue tracker – [https://github.com/rtrlib/rtrlib/issues](https://github.com/rtrlib/rtrlib/issues)
For developers, we provide a pre-build API documentation online\(^\text{11}\) which documents the API of the latest release. Alternatively, and if Doxygen is available on your system, you can build the documentation locally as follows:

```
make doc
```

To execute the build-in tests provided by the RTRlib package, run:

```
make test
```

## 13.3 RTRlib Command Line Tools

The RTRlib software package includes two lightweight command line tools to showcase some of the RTRlib features. `rtr-client` connects to an RPKI cache server, fetches and maintains the valid ROA payloads, and prints the received data. `rpki-rov` allows to verify whether an autonomous system is the legitimate origin AS of an IP prefix, based on RPKI data.

If you want to use these command line tools, you need an RPKI-RTR connection to an RPKI cache server (e.g., Routinator). For those who do not have access to a cache server, we provide a public cache with `hostname rpki-validator.realmv6.org` and `port 8282`.

### 13.3.1 RTRlib RTR Client

`rtrclient` is part of the default RTRlib software package. This command line tool connects to an RPKI cache server and prints the received valid ROA payloads to standard out.

To establish a connection to RPKI cache servers, the client can use `TCP` or `SSH` transport sockets. To run the program you have to specify the transport protocol as well as the hostname and port of the RPKI cache server; additionally you can set several options. To get a complete reference over all options for the command simply run `rtrclient` in a shell.

Listing 13.1 shows how to connect the `rtrclient` to a cache server as well as 10 lines of the resulting output. It shows IPv4 and IPv6 prefixes secured by ROAs, the allowed prefix lengths, and the legitimate origin AS numbers. Each line represents either a ROA that was added (+) or removed (–) from the selected RPKI cache server. The RTRlib client will receive and print such updates until the program is terminated, i.e., by `ctrl + c`.

### Listing 13.1: Output of the rtrclient tool.

```
<table>
<thead>
<tr>
<th>Prefix</th>
<th>Prefix Length</th>
<th>ASN</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ 89.185.224.0</td>
<td>19 - 19</td>
<td>24971</td>
</tr>
<tr>
<td>+ 180.234.81.0</td>
<td>24 - 24</td>
<td>45951</td>
</tr>
<tr>
<td>+ 37.32.128.0</td>
<td>17 - 17</td>
<td>197121</td>
</tr>
<tr>
<td>+ 161.234.0.0</td>
<td>16 - 24</td>
<td>6306</td>
</tr>
<tr>
<td>+ 85.187.243.0</td>
<td>24 - 24</td>
<td>29694</td>
</tr>
<tr>
<td>+ 2a02:5d8::</td>
<td>32 - 32</td>
<td>8596</td>
</tr>
<tr>
<td>+ 2a03:2260::</td>
<td>30 - 30</td>
<td>201701</td>
</tr>
<tr>
<td>+ 2001:13c7:6f08::</td>
<td>48 - 48</td>
<td>27814</td>
</tr>
<tr>
<td>+ 2a07:7cc3::</td>
<td>32 - 32</td>
<td>61232</td>
</tr>
<tr>
<td>+ 2a05:b480:fc00::</td>
<td>48 - 48</td>
<td>39126</td>
</tr>
</tbody>
</table>
```

\(^{11}\) API reference – https://rtrlib.realmv6.org/doxygen/latest
13.3.2 RTRlib ROV Validator

rpki-rov is also part of the RTRlib software package. This simple command line interface allows to verify whether an autonomous system is allowed to announce a specific IP prefix, based on data received from an RPKI cache server.

To run the program, you must provide two parameters, hostname and port of a known RPKI cache server. Then, you can interactively validate IP prefixes by typing prefix, prefix length, and origin ASN separated by spaces. Press ENTER to run the validation. The result will be shown instantly below the input.

Note: rpki-rov can validate IPv4 and IPv6 prefixes by default.

Listing 13.2 shows the validation results of all RPKI-enabled RIPE RIS beacons. The output consists of three columns, which are separated by pipes (|):

<input query> | <ROAs> | <validation result>.

The validation results are 0 for valid, 1 for not found, and 2 for invalid.

In case of a valid and invalid prefix-AS pair, the output shows the matching ROAs for the given prefix and AS number. If multiple ROAs for a prefix exist, they are listed in a row separated by commas (,).

Listing 13.2: Output of rpki-rov showing validation results of multiple prefixes.