Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Fabian Holler

Konzeption und Entwicklung einer Client-seitigen RPKI-RTR Library zur Validierung der Präfix-Zugehörigkeit von autonomen Systemen in BGP-Routen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Fabian Holler

Konzeption und Entwicklung einer Client-seitigen RPKI-RTR Library zur Validierung der Präfix-Zugehörigkeit von autonomen Systemen in BGP-Routen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Thomas Schmidt
Zweitgutachter : Prof. Dr. rer. nat. habil. Dirk Westhoff

Abgegeben am 9. November 2011

Fabian Holler

Thema der Bachelorarbeit

Konzeption und Entwicklung einer Client-seitigen RPKI-RTR Library zur Validierung der Präfix-Zugehörigkeit von autonomen Systemen in BGP-Routen

Stichworte

Sicherheit, BGP, Präfix-Hijacking, MOAS, RPKI-RTR, SIDR

Kurzzusammenfassung

Die Routing-Infrastruktur des Internets ist anfällig für eine Reihe von Angriffen, welche die Stabilität des Internets gefährden. Um die Robustheit des Internet-Routings zu verbessern, entwickelt die SIDR Arbeitsgruppe der IETF eine RPKI zur Validierung der Präfix-Zugehörigkeit von autonomen Systemen. Auf lokalen Cache-Servern werden Zertifikate für IP-Präfixe validiert und Routern über das RPKI-RTR Protokoll zur Verfügung gestellt. Mithilfe der Validierungsdatensätze des Cache-Servers kann ein Router valide Routen-Annoncierung von gefälschten unterscheiden.

In dieser Arbeit wird eine Bibliothek zur Nutzung des Client-seitigen RPKI-RTR Protokolls entworfen und implementiert. Mithilfe der Bibliothek soll es ermöglicht werden, vorhandene BGP-Daemons um RPKI-Unterstützung zu erweitern sowie Test- und Analyseprogramme für RPKI-Cache-Server zu entwickeln. Diese Arbeit gibt eine Übersicht über die vorhandenen BGP-Sicherheitsprobleme und erläutert mögliche Lösungen. Des Weiteren wird die Funktionsweise der RPKI und des RPKI-RTR-Protokolls erläutert. Es erfolgt eine Dikussion des Architekturentwurfs und der Implementierung der Bibliothek. Abschließend wird die Performanz der Bibliothek evaluiert und die Validierungsergebnisse aktueller BGP-Annoncierungen analysiert.

Fabian Holler

Title of the paper

Design and implementation of a client-sided RPKI-RTR library for origin validation of BGP routes

Keywords

Security, BGP, Prefix-Hijacking, MOAS, RPKI-RTR, SIDR

Abstract

One of the most fragile and vulnerable parts of the Internet is its core routing-infrastructure. To improve the robustness of the internet routing, the SIDR working group of the IETF recently developed a RPKI that certifies network prefixes as belonging to certain autonomous systems. A BGP-Router retrieves a list of certified origin ASs for network routes with the RPKI-RTR protocol from a local certificate validation server. The validation records enable a router to distinguish authorized route annoncements from forged ones.

In this bachelor thesis, a client-sided RPKI-RTR library is developed. The library shall simplify the enhancement of existent BGP server implementations with RPKI support and the development of RPKI cache server analysis and test tools. This thesis gives an overview of BGP security issues and possible countermeasures in particular it describes the functionality of the RPKI and the RPKI-RTR Protocol. Furthermore the devolpment and implementation of a client-sided RTR C library is discussed. Finally the performance of the library is evaluated and the validation results of current BGP-Announcements are analyzed.

Danksagung

Sebastian Meiling für das Korrekturlesen und zur Verfügung Stehen für Diskussionen über offene Fragen.

Thomas Schmidt für die Unterstützung bei der Erstellung der Bachelorarbeit sowie das Bereitstellen des Arbeitsplatzes in der inet Arbeitsgruppe.

Matthias Wählisch für die Idee des Themas der Arbeit sowie die Betreuung während der Ausarbeitung.

Inhaltsverzeichnis

Tabellenverzeichnis **8**

Abbildungsverzeichnis **9**

1. Einleitung **10**
1.1. Motivation . . . 10
1.2. Allgemeine Grundlagen . . . 11
1.3. Problemstellung . . . 12
1.4. Aufbau der Arbeit . . . 13

2. Grundlagen **14**
2.1. Vergabe von IP-Adressen . . . 14
2.2. IP-Routing . . . 14
2.3. Autonome Systeme . . . 16
2.4. Border Gateway Protokoll . . . 17
2.5. Public Key Infrastructures . . . 20

3. Analyse von BGP-Sicherheitsproblemen und möglichen Lösungen **22**
3.1. BGP-Sicherheitsschwachstellen . . . 22
3.1.1. Präfix-Hijacking . . . 22
3.1.2. Modifizierung des AS_PATH Attributs . . . 23
3.1.3. Link-Flapping . . . 25
3.1.4. TCP-basierte Angriffe . . . 25
3.2. Lösungsansätze . . . 26
3.2.1. Secure BGP . . . 26
3.2.2. Resource Public Key Infrastructure . . . 27
3.2.3. BGPsec . . . 31
3.3. Zusammenfassung . . . 33

4. RPKI-RTR Protokoll **34**
4.1. Transportprotokolle . . . 34
4.2. Protokollablauf . . . 36
4.3. Validierung von BGP-Präfixen . . . 38

5. Entwurf der Bibliothek **39**
5.1. Anforderungen . . . 39
5.2. Architektur . . . 40
5.3. Transport-Socket . . . 41
5.4. RTR-Socket . . . 43
5.5. Prefix Validation Table . . . 45
5.6. RTR Connection Manager . . . 47

6. Implementierung **50**
6.1. Wahl der Programmiersprache . . . 50
6.2. Implementierung der Transport-Sockets . . . 51
6.2.1. TCP-Transport-Socket . . . 51
6.2.2. SSH-Transport-Socket . . . 54
6.2.3. Funktionalitätstests . . . 56
6.3. Implementierung des RTR-Sockets . . . 56
6.3.1. Realisierung des RTR-Protokolls als Zustandsautomat . . . 56
6.3.2. Verarbeitung der RTR-Nachrichten . . . 58
6.3.3. Funktionalitätstests . . . 60
6.4. Implementierung der Prefix Validation Table . . . 61
6.4.1. Auswahl einer Datenstruktur zur Verwaltung der Validierungsdatensätze . . . 61
6.4.2. Realisierung der Prefix Validation Table . . . 65
6.4.3. Funktionalitätstests . . . 68
6.5. Implementierung des RTR Connection Managers . . . 69

7. Evaluation **73**

8. Zusammenfassung und Ausblick **78**

Literaturverzeichnis **80**

A. Anhang **85**
A.1. Quellcodeausschnitte der Bibliothek . . . 85
A.2. Evaluationsprogramme . . . 88
A.3. Unit-Tests . . . 95
A.4. Konfigurationsdateien . . . 113

Tabellenverzeichnis

2.1. Zuständigkeitsbereiche der RIRs [12] 15
2.2. Beispiel einer Routing-Tabelle 16

6.1. Speicherbedarf eines Validierungsdatensatzes 64

Abbildungsverzeichnis

2.1. Vergabe von IP-Adressen . . . 15
2.2. Austausch von Pfadinformationen zwischen autonomen Systemen . . . 20

3.1. Präfix-Hijacking . . . 24
3.2. Präfix-Hijacking per Präfix-Deaggregation . . . 24
3.3. Beispiel der RPKI-Zertifikatsvergabe . . . 29
3.4. RPKI-Struktur . . . 31
3.5. BGPsec . . . 32

4.1. RPKI-RTR Datensynchronisation . . . 37

5.1. Komponenten der RPKI-RTR Bibliothek . . . 41

6.1. RPKI-RTR Zustandsautomat . . . 56
6.2. Sequenzdiagramm der rtr_recv_pdu(..) Funktion . . . 59
6.3. Longest Prefix First Search Tree . . . 62

7.1. Testaufbau . . . 74
7.2. Erzeugte CPU-Last durch Validierungsoperationen (1336 Datensätze) . . . 75
7.3. Validierungsergebnisse . . . 76
7.4. Erzeugte CPU-Last durch Validierungsoperationen (2.093.971 Datensätze) 77

1. Einleitung

1.1. Motivation

Das Internet ist heutzutage von großer Relevanz für unsere Gesellschaft. Ein Großteil der westlichen Bevölkerung nutzt täglich das Internet zur Informationsbeschaffung, Abwicklung von Geschäften, Unterhaltung oder zur Kommunikation. Bei Protesten und in staatlichen Krisenzeiten dient es den Bürgern als Medium zur Organisation und zur Informationsübermittlung. In der Wirtschaft sind ganze Geschäftszweige entstanden, welche auf dem Internet basieren [36]. Durch die Einbindung des Internets in unser tägliches Leben werden aber auch die Folgen von Internetausfällen schwerwiegender. Angestellte können ihre Berufe nur eingeschränkt ausüben, Firmen keine betriebsnotwendigen Daten übertragen und täglich genutzte Informationsquellen und Kommunikationsmedien stehen nicht zur Verfügung. Aufgrund von fehlenden Schutzmaßnahmen für das Internet-Routing gegenüber Fehlkonfigurationen und Angriffen sind größere Internetausfälle aber keineswegs unwahrscheinlich.

Das Internet ist unterteilt in logische Organisationsstrukturen, welche *autonome Systeme* (AS) genannt werden. Ein AS umfasst mindestens ein IP-Netz und kommuniziert dieselben Pfade für IP-Präfixe an andere ASe. Zum Austausch von Routing-Informationen zwischen ASen wird das *Border Gateway Protokoll* (BGP) eingesetzt. Dieses ist aufgrund von fehlenden Validierungsmechanismen für die übermittelten Daten anfällig für Fehler, welche durch die Verbreitung von falschen Routing-Informationen entstehen. Akzeptieren Router falsche Pfad-Informationen, kann dies die Unerreichbarkeit von Teilen des Internets, Verzögerungen bei der Datenzustellung oder Stabilitätsprobleme zur Folge haben. Dritte, welche gezielt falsche Routing-Informationen verbreiten, haben die Möglichkeit sensitive Information mitzulesen und die übertragenen Daten zu manipulieren. Eine große Gefahr liegt in der unrechtmäßigen Übernahme eines IP-Präfixes (*Präfix-Hijacking*). Hierbei annonciert ein BGP-Router ein Pfad für ein Netzwerkpräfix, für das er nicht zuständig ist. Die Pakete, welche über diesen Pfad übertragen werden, können von Unberechtigten eingesehen und manipuliert werden oder erreichen nicht ihr eigentliches Ziel. In der Vergangenheit gab es bereits mehrere Routing-Probleme, welche aufgrund von Präfix-Hijacks entstanden sind [61]. Zum Beispiel führte 2008 die falsche Bekanntgabe eines Präfixes, durch Pakistan Telecom, zu einer kurzzeitigen Unerreichbarkeit von YouTube.com [22]. 2010 wur-

den aufgrund falsch verbreiteter Routing-Pfade für eine Viertelstunde 15% des gesamten Internet-Verkehrs durch den Provider China Telecom geleitet. Für diesen bestand damit die Möglichkeit sensitive Informationen abzuhören und zu übertragende Daten zu modifizieren. abzuhören [23].

1.2. Allgemeine Grundlagen

Um die Sicherheit des Internet-Routings zu verbessern, wurde 2007 die *Secure Inter-Domain Routing* (SIDR) Arbeitsgruppe [19] der *Internet Engineering Task Force* (IETF) [21] gegründet. Diese hat einen Entwurf für eine *Resource Public Key Infrastructure* (RPKI) [49] erarbeitet, welche es ermöglicht Präfix-Hijacks zu verhindern. Eigentümer von IP-Präfixen können Zertifikate erstellen, welche dem BGP-Router eines autonomen Systems autorisieren, eine Route für bestimmte Präfixe zu annoncieren. Die Zertifikate werden auf öffentlichen Servern bereitgestellt, von lokalen Cache-Servern heruntergeladen und validiert. Die Cache-Server stellen den Routern über das RPKI-RTR-Protokoll [33] eine Liste von autonomen Systemen, welche ein bestimmtes IP-Präfix annoncieren dürfen, zur Verfügung. Die Idee, BGP-Informationen kryptographisch mithilfe einer PKI abzusichern, war bereits Basis mehrerer Entwürfe zur Absicherung von BGP (z. B. *Secure BGP* (S-BGP) [50]). Bis heute konnte sich allerdings keine Sicherheitserweiterung u. a. aufgrund der folgenden Aspekte durchsetzen:

- Die Validierung und Speicherung der Zertifikate auf den Routern hat eine erhöhte CPU-Last und einen erhöhten Arbeitsspeicherbedarf zur Folge. Die Router-Betreiber müssten bereit sein, in neue Hardware zu investieren und Kosten für die Installation der Sicherheitserweiterung zu tragen.
- Der Rechenaufwand für die Sicherheitserweiterung darf die Möglichkeiten von aktuellen Hardware-Routern nicht übersteigen. Hardware-Router verfügen oft nur über 512 MB RAM und können nicht auf vergleichbare Arbeitsspeichermengen wie Desktop- oder Serverrechner aufgerüstet werden.
- Eine gleichzeitige Installation einer Sicherheitserweiterung auf allen Routern im Internet ist nicht möglich. BGP-Protokollerweiterungen oder separate Sicherheitserweiterungen müssen deshalb inkrementell installiert werden können.
- Kryptographische Zertifikate, die zur Validierung von Routing-Informationen genutzt werden verfügen über ein Ablaufdatum. Falls das Zertifikat vom Eigentümer bis zum Ablaufdatum nicht erneuert wurde besteht die Gefahr, dass korrekte Routing-Informationen als falsch validiert werden. Infolgedessen können Netzbereiche unerreichbar werden.

Beim RPKI-Verfahren sind diese Aspekte beachtet worden. Durch die Validierung der Zertifikate auf separaten Cache-Servern entsteht nur eine minimale Zusatzlast auf den Routern. Es ist eine inkrementelle Installation im Internet möglich. Sicherheitsbewusste BGP-Administratoren können RPKI in ihrem AS einsetzen ohne auf die Installation von RPKI in anderen ASen angewiesen zu sein. Zudem bekommen abgelaufene Zertifikate einen neutralen Validierungsstatus zugewiesen, statt als ungültig klassifiziert zu werden.

Installationsbemühungen für RPKI im Internet haben bereits Anfang 2011 begonnen. Alle Regional Internet Registries (RIRs), ausgenommen der ARIN, stellen seit dem 01.01.2011 Zertifikats-Repositories zur Verfügung [27]. Bei der RIPE wurden bis zum 21.10.2011 bereits Zertifikate für 1008 IPv4-Präfixe hinterlegt [26]. Zur Betreibung von lokalen Cache Servern stehen die RPKI-RTR-Server Implementierungen rtr-origin [6] und RPKI-Validator [16] zur Verfügung. Die beiden größten Router-Herstellen Cisco und Juniper arbeiten bereits an RPKI-Implementierungen für ihre Router [40]. Abgesehen von Hardware-Router werden in kleineren Netzwerken oft freie BGP-Implementierungen, wie z. B. Quagga [15], oder BIRD [20] eingesetzt. Für diese gibt es derzeit noch keine RPKI-RTR Implementierungen oder laufende Entwicklungen.

1.3. Problemstellung

Ziel dieser Bachelor-Arbeit, ist die Entwicklung einer frei verfügbaren RPKI-RTR-Client-Bibliothek, welche es ermöglicht, RPKI-Datensätze von einem lokalen Cache Server abzurufen und BGP-Routen gegen die Datenbasis zu validieren. Mithilfe dieser Bibliothek sollen vorhandene BGP-Router-Implementierungen um RPKI-Unterstützung erweitert sowie Testprogramme für RPKI-Server entwickelt werden können. Die Implementierung des Clients als eine frei verfügbare Bibliothek, in Abgrenzung zu einer spezifischen Implementierung für nur einen Routing-Daemon, hat mehrere Vorteile:

- Es ermöglicht die Einbindung des RPKI Verfahrens in bestehende BGP-Daemons mit geringem Aufwand. Statt einer vollständigen Neuimplementierung des Verfahrens ist nur eine Integration der Bibliothek nötig.
- Probleme, die z. B. aufgrund von Fehlern in der RTR-Spezifikation entstanden sind, müssen nur einmal behoben werden, statt in jeder spezifischen BGP-Router-Implementierung.
- Das Vorhandensein einer frei nutzbaren Bibliothek unterstützt eine schnelle Verbreitung des RPKI-Verfahrens im Internet, da der Aufwand für die Erweiterung von BGP-Software minimiert wird.

- Die Bibliothek kann nicht nur für eine Nutzung in BGP-Daemons verwendet werden, sondern auch z. B. zum Erstellen von Testprogrammen für RTR-Server. Mit diesen könnte z. B. die Stabilität und Spezifikationskonformität von RTR-Servern überprüft werden.

Um die Interoperabilität mit RPKI-Servern sicherzustellen, muss die RPKI-RTR Spezifikationen [33] beachtet werden. Die Implementierung von späteren Änderungen an der RPKI-RTR-Spezifikation soll durch die Anwendungsarchitektur ermöglicht und unterstützt werden. Um die Hardware des Routers nicht unnötig zu belasten, soll die Bibliothek möglichst effizient implementiert sein.

1.4. Aufbau der Arbeit

In zweiten Abschnitt der Arbeit werden die benötigten Grundlagen zum Verständnis des Themas erläutert. Anschließend findet eine Analyse von BGP-Sicherheitsproblemen statt, in der die Gefahren erläutert sowie S-BGP, RPKI und BGPsec als mögliche Lösungen diskutiert werden. Abschnitt 4 vermittelt die Funktionsweise des RPKI-RTR Protokolls, welches die Bibliothek implementieren soll. Im nachfolgenden Abschnitt werden die Anforderungen an die Bibliothek dargelegt und der Entwurf der Bibliothek beschrieben. Nachfolgend wird, in Abschnitt 6, die Implementierung der Bibliothek vorgestellt. Anschließend findet eine Evaluation der Arbeit statt und es wird ein abschließendes Fazit gegeben.

2. Grundlagen

Im folgenden Abschnitt werden die grundlegenden Kenntnisse, welche zum Verständnis der Arbeit notwendig sind, erläutert. In den ersten Kapiteln wird die gängige Praxis bei der Vergabe von IP-Adressen im Internet sowie die Funktionsweise des IP-Routings vermittelt. Anschließend erfolgt eine Erläuterung der Funktionsweise des Border Gateway Protokolls und von Public Key Infrastructures.

2.1. Vergabe von IP-Adressen

Mithilfe einer Resource Public Key Infrastructure (RPKI) wird die Annoncierung von IP-Präfixen abgesichert, welche von der *Internet Assigned Numbers Authority* (IANA) [11] vergeben werden. Die Vergabe von IP-Adressen ist hierarchisch organisiert, an oberster Stelle der Hierarchie steht die IANA. Blöcke von IP-Adressen vergibt die IANA an die *Regional Internet Registries* (RIRs), welche für die Zuteilung von Adressbereichen in größeren geographischen Gebieten zuständig sind (siehe Tabelle 2.1) [42]. Diese wiederum delegieren die ihnen zugeteilten Adressbereiche an die *Local Internet Registrys* (LIRs). Dies sind Internetprovider, Universitäten oder große Unternehmen, welche die IP-Bereiche an Endnutzer vergeben.

Eine exemplarische Vergabe von IP-Blöcken ist in Abbildung 2.1 dargestellt. Die IANA vergibt den Adressbereich `141.0.0.0/8` an die RIPE, welche wiederum das Subnetz `141.22.0.0/16` an das Deutsche Forschungsnetz (DFN) delegiert. Das DFN vergibt den IP-Bereich `141.22.0.0/16` an die HAW-Hamburg.

2.2. IP-Routing

Damit Rechner aus verschiedenen Netzbereichen miteinander kommunizieren können, müssen IP-Pakete von Routern in das jeweilige Zielnetzwerk transportiert werden. Um den nächsten zuständigen Router für ein Zielnetzwerk zu ermitteln, verfügt jeder Rechner über eine Routing-Tabelle (siehe Tabelle 2.2). Diese enthält, für erreichbare Zielnetzwerke, die

Regional Internet Registry	Zuständigkeitsbereiche
African Network Information Center (AfriNIC)	Afrika
Asian-Pacific Network Information Centre (APNIC)	Asien-Pazifik
American Registry for Internet Numbers (ARIN)	Nord-Amerika
Latin America and Caribbean Network Information Centre (LACNIC)	Lateinamerika, Karibik Inseln
Réseaux IP Européens Network Coordination Centre (RIPE)	Europa, Mittlerer Osten, Zentral-Asien

Tabelle 2.1.: Zuständigkeitsbereiche der RIRs [12]

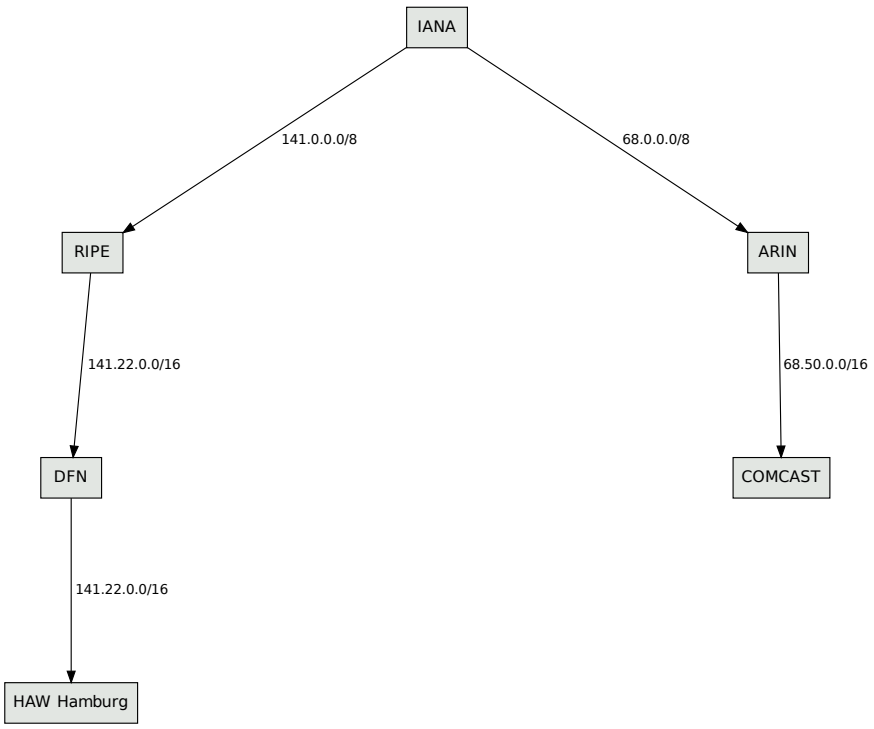

Abbildung 2.1.: Vergabe von IP-Adressen

Zielnetzwerk	Router	Schnittstelle
123.24.5.0/24	192.168.0.1	eth0
123.24.0.0/16	192.168.1.254	eth1

Tabelle 2.2.: Beispiel einer Routing-Tabelle

IP-Adressen von Routern, welche die Pakete in das jeweilige Netzwerk weiterleiten können. Ein Rechner, der ein Paket in ein anderes IP-Netzwerk verschicken soll, liest aus seiner Routing-Tabelle die IP-Adresse eines zuständigen Routers aus und sendet das Paket an diesen. Falls der Router über eine Netzwerkschnittstelle verfügt, welche sich im Zielnetzwerk befindet, wird das Paket über die Schnittstelle an den Zielrechner gesendet. Anderenfalls entnimmt der Router aus seiner Routing-Tabelle den nächsten zuständigen Router und leitet das Paket an diesen weiter. Existieren mehrere Routen, über die ein Paket in das Zielnetzwerk weitergeleitet werden kann, wird die Route mit dem längsten Ziel-Präfix ausgewählt (*Longest Prefix Match*).

Beispielsweise soll ein Paket an die Zieladresse `123.24.5.1` zugestellt werden. Der Router verfügt über Pfade für die Ziele `123.24.0.0/16` und `123.24.5.0/24` in seiner Routing-Tabelle (siehe Tabelle 2.2). Anhand der Longest Prefix Match Regel würde die Route `123.24.5.0/24` ausgewählt werden, da die Präfix-Länge 24 länger als 16 ist. Zum dynamischen Erstellen und Aktualisieren der Einträge in der Routing-Tabelle tauschen Router die erreichbaren Zielnetzwerke über Routing-Protokolle miteinander aus. Router innerhalb einer Organisation kommunizieren mithilfe von *Interior Gateway Protokollen* (IGP) miteinander. Für den Austausch von Pfadinformationen mit anderen Organisationen werden die internen Routen aggregiert und über *Exterior Gateway Protokolle* (EGP) ausgetauscht.

2.3. Autonome Systeme

Eine Organisationseinheit, welche IP-Routen für eine Menge von IP-Blöcken einheitlich verwaltet, wird als *autonomes System* (AS) bezeichnet. Ein AS stellt sich nach außen als eine einzige Entität da und kommuniziert für die Erreichung seiner Adressbereiche Routen an benachbarte ASe. Für die Identifizierung bekommen autonome Systeme eine 8- oder 32-Bit lange *autonomous System Number* (ASN) zugewiesen[1]. Für die Vergabe der ASNs sind ebenfalls die IANA und die RIRs zuständig. Anders als bei der Vergabe von IP-Adressen werden ASNs von den RIRs direkt an die Endkunden vergeben. Die ASN wird zur Identifizierung von autonomen Systemen in EGPs, wie z. B. dem *Border Gateway Protokoll* genutzt.

[1] Die ASN war bis 2007 8-Bit lang wurde dann im RFC 4893 [58], um eine vorhersehbare ASN-Knappheit zu vermeiden, auf 32-Bit vergrößert.

2.4. Border Gateway Protokoll

Das Border Gateway Protokoll (BGP) [55] ist das de facto Standard Routing-Protokoll im Internet. Es kann sowohl als EGP (eBGP) zum Austausch von Routing-Informationen zwischen autonomenen Systemen als auch als IGP (iBGP) zum Austausch von Routing-Informationen innerhalb eines ASes eingesetzt werden. Zum Transfer der BGP-Nachrichten baut ein BGP-Sprecher eine Verbindung zu seinen BGP-Nachbarn[2] über TCP (Port 179) auf. Die empfangenen Routing-Informationen werden von den BGP-Routern bewertet und falls sie den lokalen Richtlinien entsprechen an die jeweiligen Nachbarn weitergeleitet. Zwischen den BGP-Routern entsteht somit ein vermaschtes Netzwerk, indem sich Pfadinformationen im ganzen Internet ausbreiten. Für die Kommunikation zwischen BGP-Sprechern sind folgende vier Nachrichtentypen definiert:

OPEN-Nachrichten werden bei Aufbau einer BGP-Verbindung gesendet, um Informationen über die BGP-Sprecher auszutauschen und Verbindungsparameter auszuhandeln.

UPDATE-Nachrichten enthalten Informationen zu Netzwerkpfaden, welche die BGP-Sprecher untereinander austauschen.

KEEPALIVE-Nachrichten müssen regelmäßig von beiden Verbindungspartnern übertragen werden, um die Verbindung aufrechtzuerhalten.

NOTIFICATION-Nachrichten werden verschickt, um den Kommunikationspartner über das Auftreten eines Fehlers zu informieren. Nach Absenden einer *NOTIFICATION*-Nachricht, wird die BGP-Verbindung beendet.

Nach einem erfolgreichen TCP-Verbindungsaufbau tauschen beide BGP-Sprecher zum Abstimmen der Verbindungsparameter *OPEN*-Nachrichten aus. Hierbei wird die ASN, die IP-Adresse des BGP-Routers, ein Timeout für die Verbindung[3] sowie Informationen über genutzte Protokollerweiterungen übertragen. Wenn die Verbindungspartner den Parametern in der *OPEN*-Nachricht zustimmen, senden beide zur Bestätigung eine *KEEPALIVE*-Nachricht. Falls ein Parameter nicht konform mit der Konfiguration des anderen BGP-Routers ist, sendet dieser eine *NOTIFICATION*-Nachricht mit einem entsprechenden Fehler-Code und beendet die Verbindung. Nach der Zustimmung zu den Verbindungsparametern tauschen die BGP-Router mithilfe von *UPDATE*-Nachrichten ihre Routing-Informationen aus. *UPDATE*-Nachrichten können eine Vielzahl von IP-Präfixen enthalten, für die eine neue Route annonciert wird, sowie mehrere Präfixe, für welche die vorher bekannt gegebenen Routen zurückgezogen werden. Für die annoncierten IP-Präfixe enthält

[2] Eine Gruppe von statisch hinterlegten BGP-Routern, mit denen Routing-Informationen ausgetauscht werden.

[3] Nach der Initialisierung muss mindestens eine *KEEPALIVE*-, *UPDATE*- oder *NOTIFICATION*-Nachricht vor Ablauf des Timeouts übertragen werden ansonsten wird die Verbindung beendet.

eine *UPDATE*-Nachricht mehrere Attribute, welche für alle in der Nachricht annoncierten Präfixe gelten. Folgende Attribute müssen in jeder *UPDATE*-Nachricht zwingend enthalten sein:

ORIGIN Beschreibt die Herkunft der Route, mögliche Werte sind:

- **IGP** Die Routeninformationen stammen aus einem Netzwerk innerhalb des ASes.
- **EGP** Die Routeninformationen wurden von einem anderen externen BGP-Router empfangen.
- **INCOMPLETE** Die Route stammt aus anderen Quellen. Beispielsweise kann es sich um eine statisch konfigurierte Route handeln, die über BGP annonciert wird.

AS_PATH Das AS_PATH Attribut enthält die ASNs der autonomen Systeme, die die Routeninformation durchlaufen hat. Jeder BGP-Router, welcher *UPDATE*-Nachrichten versendet, fügt seine ASN zum AS_PATH Attribut hinzu. Der traversierte Pfad kann als unsortierte (AS_SET[4]) oder sortierte (AS_SEQUENCE) Liste im Attribut gespeichert werden. Wenn es sich um eine AS_SEQUENCE handelt, hängen die Router ihre ASN an den Anfang der Liste an. Der erste Eintrag im AS_PATH, ist in diesem Fall die ASN des autonomen Systems, in welchem die Rechner mit dem annoncierten IP-Präfix stationiert sind. Das AS_PATH Attribut dient zur Verhinderung von Schleifen beim Weiterleiten von Pfad-Informationen. Zusätzlich wird es im *Tie-Breaking-Verfahren* genutzt, wenn für ein IP-Präfix eine Route unter mehreren ausgewählt werden muss. Routen mit weniger Einträgen im AS_PATH Attribut werden beim Tie-Breaking-Prozess bevorzugt.

NEXT_HOP Das NEXT_HOP Attribut enthält die IP-Adresse des Routers, an welche die IP-Pakete für die annoncierten Präfixe zum Routing geschickt werden.

Zusätzlich zu den beschriebenen Attributen kann eine *BGP-UPDATE*-Nachricht optionale und benutzerdefinierte Attribute enthalten. Benutzerdefinierte Attribute können von Protokollerweiterungen genutzt werden um zusätzliche Informationen in einer *BGP-UPDATE*-Nachricht zu verschicken. Nach Empfang einer neuen Routen-Annoncierung speichert ein BGP-Router die Routeninformation in einer Datenstruktur, welche *Adjacent-Routing-Information-Base-Incoming* (Adj-RIB-In) genannt wird. Immer wenn Daten in der Adj-RIB-In durch Empfang einer *BGP-UPDATE*-Nachricht geändert werden, wird ein Entscheidungsprozesses durchlaufen. Am Ende des Prozesses wurden Routen ausgewählt, die an die BGP-Nachbarn weitergeleitet wurden und in die Routing-Tabelle übertragen werden können. Der Entscheidungsprozess ist in folgende Phasen unterteilt:

[4] Von der Verwendung von AS_SETs wird in einem IETF-Draft [47] abgeraten.

Phase 1 In der ersten Phase des Entscheidungsprozess wird der Präferenzwert der Routen in der Adj-RIB-In berechnet. Dieser wird im weiteren Prozess genutzt, um eine Route unter mehreren für ein IP-Präfix auszuwählen. Wenn die Route von einem iBGP-Nachbarn stammt, kann diese einen berechneten Präferenzwert (`LOCAL_PREF` Attribut) enthalten, welcher vom Empfänger übernommen werden kann. Die Höhe des Präferenzwerts ist abhängig von lokalen individuellen Bewertungsregeln.

Phase 2 In der zweiten Phase werden die besten Routen aus der Adj-RIB-In ausgewählt und in die *Local-Routing-Information-Base* (Loc-RIB) übertragen. Aus der Loc-RIB wählt ein Router Pfade aus, die in die Routing-Tabelle übertragen werden. Routen werden aus dem weiteren Entscheidungsprozess ausgeschlossen wenn die IP-Adresse im `NEXTHOP`-Feld nicht erreichbar ist oder das `AS_PATH`-Attribut bereits die eigene ASN enthält. Aus den übrigen Routen, wird die Route mit dem höchsten Präferenzwert für ein Ziel-Präfix ausgewählt und in Loc-RIB kopiert. Falls mehrere Routen mit demselben Präferenzwert für ein IP-Präfix existieren, wird eine Route mittels des Tie-Breaking-Prozesses ausgewählt und in die Loc-RIB übertragen. Wenn bereits ein Pfad für das Präfix in der Loc-RIB existiert, wird der bereits bestehende Pfad ersetzt.

Phase 3 In der dritten Phase werden Routen aus der Loc-RIB ausgewählt, die an die Routing-Nachbarn annonciert werden. Jede Route die an die Nachbarn kommuniziert wird, wird in der *Ajdacent-Routing-Information-Base-Outgoing* (Adj-RIB-Out) gespeichert. Damit eine Route in Adj-RIB-Out übertragen werden kann, muss sie den Richtlininen der *Policy Information Base* (PIB) entsprechen. Routen in der Ajd-RIB-Out können vor der Übertragung an die Nachbarn aggregiert werden. Wenn eine Route, die bereits in der Adj-RIB-Out enthalten ist bei einem späteren Entscheidungsprozess nicht mehr ausgewählt wird, wird die Route mit einer *BGP-UPDATE* bei den Nachbarn widerrufen.

Ein exemplarischer Austausch von BGP-Routing-Informationen ist in Abbildung 2.2 dargestellt. Der Router des AS1 annonciert eine Route für das IP-Präfix `1.2.3.0/24` an seinen Nachbarn AS2. AS2 empfängt die Route, trägt seine ASN in das `AS_PATH` Attribut der *BGP-UPDATE*-Nachricht ein und leitet sie wiederum zu seinem Nachbarn weiter. AS3 fügt ebenso seine ASN an den Anfang des `AS_PATH` Attributes ein und sendet die Routing-Information zu seinem Nachbar AS5.

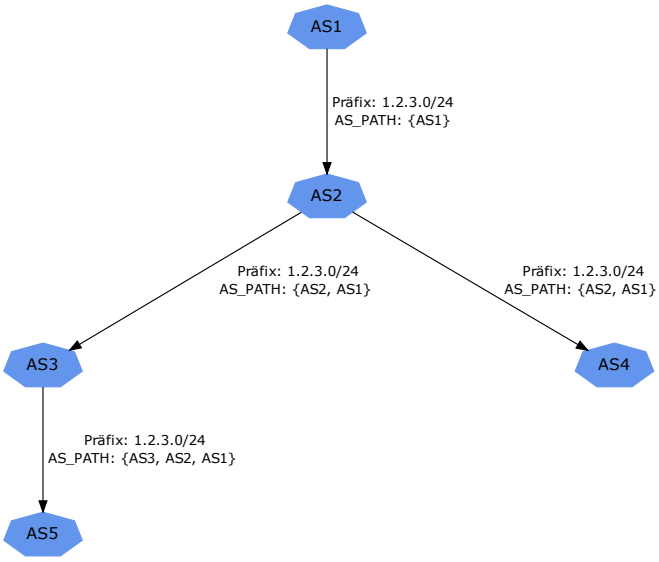

Abbildung 2.2.: Austausch von Pfadinformationen zwischen autonomen Systemen

2.5. Public Key Infrastructures

Zur Verwaltung der Zertifizierungen von IP-Präfixen nutzt das RPKI-Verfahren eine *Public Key Infrastructure* (PKI) [30]. Eine PKI dient zum Zertifizieren, Ausstellen, Verwalten und Prüfen von digitalen Zertifikaten. Als kryptographisches Verfahren wird die asymmetrische Verschlüsselung genutzt. Hierbei besteht ein Schlüsselpaar aus einem öffentlichen und einem privaten Schlüssel. Der öffentliche Schlüssel sollte in der Regel[5] frei zugänglich sein und dient zum Verschlüsseln sowie zum Überprüfen von Signaturen. Der private Schlüssel wird vom Eigentümer unter Verschluss gehalten und kann genutzt werden zum Entschlüsseln und Signieren von Objekten. Um die Identität des Eigentümers eines Schlüssels zu authentifizieren, wird das Schlüsselpaar von einer dritten, vertrauenswürdigen Stelle signiert. In einer PKI wird der Eigentümer eines Schlüsselpaares von der *Registration Authority* (RA) überprüft. Wenn die Überprüfung positiv war, stellt die *Certificate Authority* (CA) der PKI eine Signatur für das Schlüsselpaar aus. Um die Authentizität eines Schlüsselpaares zu prüfen, kann die Signatur der CA validiert werden, ist diese valide gehört der Schlüssel dem beschriebenen Eigentümer. Voraussetzung für dieses Verfahren ist, dass die RA die Identität korrekt überprüft hat und der private Schlüssel der CA nur dieser bekannt ist. Um die Gültigkeitsdauer einer Signatur zu begrenzen, können Signaturen von dem Aussteller mit einem Verfallsdatum versehen werden. Falls Signaturen vor Ablauf des

[5] Abhängig vom Einsatzzweck der asymmetrischen Verschlüsselung

Verfallsdatums zurückgezogen werden sollen, weil z. B. der private Schlüssel eines Paares gestohlen wurde, können Signaturen in die öffentliche *Certificate Revocation List* (CRL) der PKI aufgenommen werden. Wenn die Identifikationsnummer einer Signatur in der CRL enthalten ist, wird sie bei der Validierung als ungültig erachtet und gegebenenfalls bei der Beurteilung des Validierungsstatus ignoriert.

Eine Zertifizierungshierarchie, in der eine einzige PKI zum Einsatz kommt, wird als Einstufenhierarchie bezeichnet. Möglich sind auch mehrstufige Hierarchien, in welcher das Schlüsselpaar einer CA von der übergeordneten CA signiert wird. Bei der Überprüfung einer Signatur in einer mehrstufigen Hierarchie wird die komplette Signaturenkette bis zur obersten CA (Wurzel-CA) durchlaufen und validiert.

3. Analyse von BGP-Sicherheitsproblemen und möglichen Lösungen

Das Border-Gateway-Protokoll weist eine Reihe von Sicherheitslücken auf, welche die Stabilität des Internets und die Vertraulichkeit der versendeten Daten gefährden. In diesem Abschnitt werden Schwachstellen des BGP erläutert und anschließend mögliche Lösungen vorgestellt, die die Sicherheit des BGP verbessern.

3.1. BGP-Sicherheitsschwachstellen

3.1.1. Präfix-Hijacking

Präfix-Hijacking bezeichnet das Annoncieren einer Route, in welcher ein IP-Präfix mit falscher AS-Zugehörigkeit propagiert wird. Hierdurch können sowohl registrierte IP-Adressbereiche angegriffen, als auch unregistrierte Blöcke etabliert werden. Die Annoncierung von unregistrierten IP-Bereichen wird oft von SPAM-Versendern genutzt, indem ein SMTP-Server im annoncierten Bereich betrieben wird, welcher SPAM- oder Phising-E-Mails versendet. Die annoncierten IP-Bereiche werden zum Versand von unerwünschten E-Mails nur so lange genutzt, bis die verwendeten IP-Adressen zu SPAM-Blacklisten hinzugefügt worden sind. Danach werden andere unregistrierte IP-Bereiche annonciert und der E-Mail-Versand von diesen fortgesetzt [37]. Der Hijack eines registrierten IP-Bereiches ermöglicht verschiedene Angriffssszenarien, welche nachfolgend kategorisiert sind [52]:

Blackholing Der annoncierte Netzwerkpfad enthält ein AS, welches alle weiterzuleitenden IP-Pakete für ein Präfix verwirft. Für alle Rechner, die sich in einem AS befinden, welches die gehijackte Route nutzt, werden Ziele im annoncierten IP-Präfix unerreichbar.

Redirection Der Datenverkehr für Ziele im annoncierten IP-Bereich wird zu einem AS geleitet, welches sich als der eigentliche Empfänger ausgibt, aber unter der Kontrolle

des Angreifers steht. Login-Daten, Kreditkartennummern und andere sensitive Informationen, die an das fingierte Ziel übermittelt werden, können vom Angreifer mitgelesen werden (Phishing). Wenn der Datenverkehr für ein oder mehrere IP-Präfixe über eine bestimmte Leitung oder AS umgeleitet wird, die mit dem Datenaufkommen überfordert ist kann dies zu Verbindungsproblemen bis zum Zusammenbruch der Verbindung zum jeweiligen Netzbereich führen (Denial of Service (DoS)).

Subversion Bei einem Subversion-Angriff leitet der Angreifer den Datenverkehr durch ein von ihm kontrolliertes Netzwerk an das legitime Ziel weiter. Der Datenverkehr kann vom Angreifer, auf dem Weg zum Ziel, modifiziert oder mitgelesen werden.

Ein exemplarischer Präfix-Hijack ist in Abbildung 3.1 dargestellt. Die BGP-Router von AS5 und AS1 annoncieren jeweils einen Pfad für den Netzbereich `12.34.5.0/24`. Der legitime Eigentümer des Präfixes ist AS5. AS1 ist ein Angreifer, der eine illegitime Annoncierung (Präfix-Hijack) durchführt. Da beide *BGP-UPDATE*-Nachrichten eine Route für dasselbe Präfix annoncieren, entscheiden die Empfänger anhand der Länge des `AS_PATH` Attributes, welche der beiden Routen sie in ihre Routing-Tabelle aufnehmen. Die annoncierte Route von AS1 wird von AS3 mit einem Eintrag im `AS_PATH` Attribut empfangen. Die valide Routen-Annoncierungen wird über AS4 an AS5 weitergeleitet. im `AS_PATH` Attribut sind drei Einträge gespeichert. Da AS1 eine Route mit einem kürzere `AS_PATH` propagiert, würde AS2 die fingierte Route wählen und diese evtl. an seine Nachbarn annoncieren.

Durch solch einen Angriff werden nur ASe beeinflusst, welche sich in näherer Umgebung zum Angreifer als zum legitimen BGP-Router befinden. Eine größere Anzahl von BGP-Routern kann mit einer falschen Route „infiziert“ werden, indem der Angreifer einen Pfad für ein spezielleres IP-Präfix (*Präfix Deaggregation*) annonciert als der legitime Router. Im Beispiel 3.2 annonciert AS1 eine valide Route für das Netzwerk `12.34.0.0/16`. Der Angreifer AS2 annonciert eine Route für das längere Präfix `12.34.5.0/24`. Ein BGP-Router der beide Annoncierungen empfängt wird beide Routen in seine Routing-Tabelle eintragen. IP-Pakete die für das Ziel-Netzwerk `12.34.5.0/24` bestimmt sind würden immer über die Route vom Angreifer verschickt werden, da die Route mithilfe der Longest-Prefix-Match Regel ausgewählt wird.

3.1.2. Modifizierung des AS_PATH Attributs

Das `AS_PATH`-Attribut einer *BGP-UPDATE*-Nachricht beeinflusst, wie der Pfad von einem BGP-Router bewertet wird und somit, ob ein Pfad unter mehreren für ein IP-Präfix präferiert oder nachgestellt wird. Durch die künstliche Verlängerung oder Verkürzung des `AS_PATH` Attributs sind vielfältige Angriffe möglich. Wenn ein Router eine nicht optimale Route wählt, kann dies zur Folge haben, dass die Datenübertragung verzögert wird oder Routen genutzt werden, die höhere Kosten für den Betreiber zufolge haben. Beispielsweise

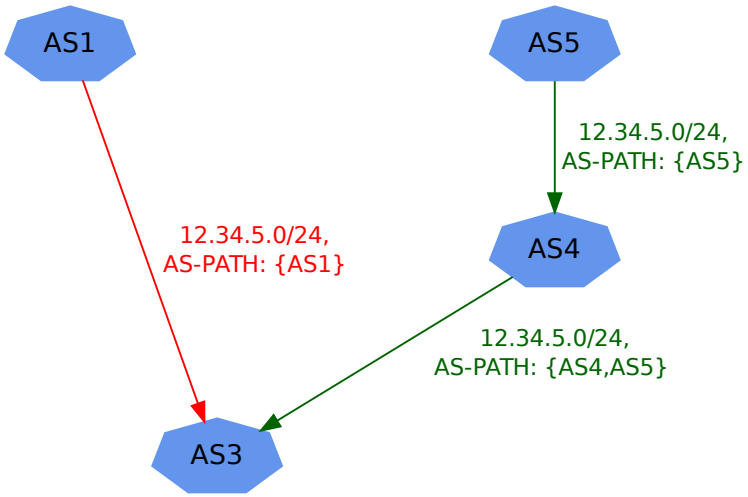

Abbildung 3.1.: Präfix-Hijacking

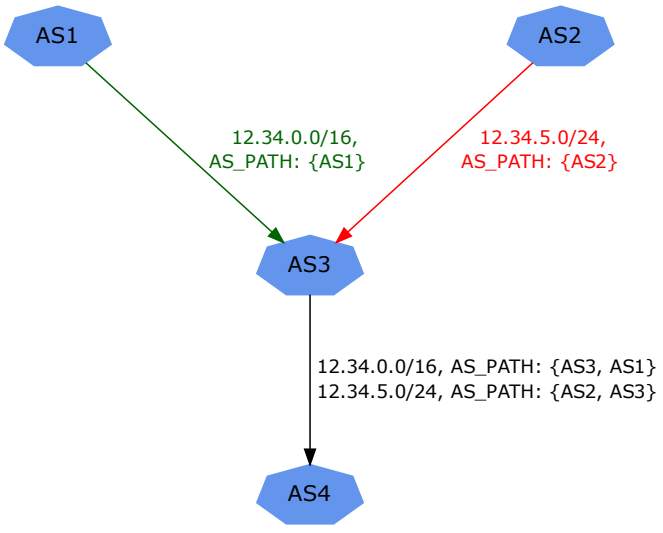

Abbildung 3.2.: Präfix-Hijacking per Präfix-Deaggregation

könnte das AS1 eine Datenleitung zum AS2 betreiben. Die Leitung ist als Notfallleitung vorgesehen und soll genutzt werden, falls keine anderen Verbindungen zu AS2 bestehen. Durch die Nutzung der Notfallleitung entstehen höhere Kosten für das AS1 als für den primären Routingpfad. Um die Notfallroute für andere ASe unattraktiv zu machen, verlängert AS1 das `AS_PATH` Attribut künstlich, indem es seine ASN zweimal zum `AS_PATH` hinzufügt (`AS_PATH`-Padding). Ein kompromittierter Router, der die Routen-Annoncierungen von AS1 empfängt, könnte die mehrfachen Einträge vom AS1 im `AS_PATH` entfernen und die modifizierte Annoncierung an seine Nachbarn weiterleiten. Dies hat zu Folge, dass die Notfallroute von den Routern, welche die modifizierte Routing-Informationen empfangen haben, der primären Routen vorgezogen und somit erhöhte Kosten für AS1 entstehen. Dasselbe Verhalten kann erzeugt werden wenn bestimmte Routen-Annoncierungen oder -Widerrufungen von einem anderen AS unterdrückt und nicht an seine Nachbarn weitergeleitet werden.

3.1.3. Link-Flapping

Link-Flapping bezeichnet das wiederholte Annoncieren und Widerrufen einer Route innerhalb einer kurzen Zeitspanne für dasselbe IP-Präfix. Der schnelle Statuswechsel führt zu einer instabilen Verbindung zu dem betroffenen IP-Präfix, erzeugt einen erhöhten BGP-Datenverkehr und erzeugt Rechenlast auf den betroffenen BGP-Nachbarn. Um dies zu verhindern, ignorieren BGP-Router die Statusänderung für die betroffene Route und entfernen sie aus ihrer Routing-Tabelle. Falls möglich wählen die Router einen alternativen Pfad, um Daten in das IP-Netz zuzustellen. Wenn kein anderer Pfad verfügbar ist, werden Ziele in dem betroffenen Netzbereich unerreichbar. Dieser Mechanismus zur Verbesserung der Robustheit von BGP-Router kann von einem Angreifer missbraucht werden, um den Datenverkehr gezielt über andere Routen umzulenken oder die Konnektivität zu bestimmten Netzbereichen zu unterbinden.

3.1.4. TCP-basierte Angriffe

Eine andere Schwachstelle im BGP stellt die Verwendung des TCPs für die Übertragung der BGP-Nachrichten zwischen Routern dar [35]. Durch TCP-basierte Angriffe kann die Verbindung zwischen zwei BGP-Routern unterbrochen oder verzögert werden. Ein Angreifer kann übertragene Nachrichten modifizieren oder gezielt bestimmte Nachrichten unterdrücken. Bei einer *SYN-Flooding*-Attacke [39] nutzt ein Angreifer aus, dass bereits während dem TCP-Verbindungsaufbaus (Drei-Wege-Handshake) Ressourcen für die zukünftige Datenübertragung von dem Kommunikationspartner alloziert werden. Wenn ein Angreifer einen TCP-Sprecher mit Verbindungsanfragen überflutet, belegt dieser daraufhin alle Ressourcen, die für TCP-Verbindungen reserviert sind und kann keine weiteren TCP-Verbindungen

mehr aufbauen. BGP-Router können keine Verbindungen mehr zu ihren Nachbarn aufbauen und somit auch keine Routeninformationen austauschen. Eine bereits bestehende TCP-Verbindung kann durch einen Angreifer terminiert werden, indem er gefälschte TCP-RST Pakete an die Kommunikationspartner verschickt [60]. Ein betroffener BGP-Router löscht bei Abbruch der TCP-Verbindung die empfangenen Routen, die er vom Verbindungspartner empfangen hat. Eine weitere Gefahr besteht durch TCP *Man-in-the-Middle* (MITM) Angriffe. Hierbei schaltet sich ein Angreifer zwischen die TCP-Verbindung zweier Rechner und hat im Zuge dessen die Möglichkeit übertragene Nachrichten zu modifizieren, bestimmte Nachrichten zu unterdrücken oder neue Nachrichten einzuschleusen. Für BGP-Router kann das zur Folge haben, dass gezielt *BGP-Update*-Nachrichten, welche bestimmte Präfixe betreffen gelöscht werden, Routen-Attribute modifiziert oder falsche *BGP-UPDATE*-Nachrichten erzeugt werden. Wenn der Angreifer gezielt die TCP- oder BGP-Keepalive Nachrichten verwirft, die zwischen beiden Rechnern regelmäßig ausgetauscht werden, kann ein Abbruch der Verbindung hervorgerufen werden.

Gegen die in diesem Kapitel erwähnten TCP-Angriffe sind in aktuellen Unix-Varianten bereits Sicherheitsmechanismen implementiert, die die Angriffe verhindern oder erschweren. Zudem existieren die von BGP-Routern oft genutzten TCP-Erweiterungen TCP-MD5 [41] und TCP-AO [56], welche TCP-Verbindungen vor Angriffen absichern.

3.2. Lösungsansätze

3.2.1. Secure BGP

Secure BGP (S-BGP) ist eine von BBN Technologies [1] entwickelte BGP-Protokollerweiterung. Die Entwicklung von S-BGP begann bereits 1997 [45]. Als eine der ersten Konzepte zur Verbesserung der Sicherheit des BGP, dient S-BGP heute oftmals als Grundlage für andere Sicherheitserweiterungen. In einer S-BGP Infrastruktur wird die Vergabe von IP-Präfixen und ASNs durch eine PKI abgesichert, welche an oberster Stelle von der IANA betrieben wird. Mithilfe von kryptographischen Signaturen wird eine Signaturkette von der IANA, über die Präfix-Eigentümer bis zu einzelnen Routern gebildet. Ein S-BGP-Router, welcher eine Pfad-Annoncierung empfängt, kann mithilfe der Zertifikate der PKI überprüfen, ob der BGP-Router berechtigt ist eine Route für das jeweilige IP-Präfix zu annoncieren (*Address Attestations* (AA)). Zusätzlich verschickt ein S-BGP-Router mit jeder *BGP-UPDATE*-Nachricht eine Autorisierung (*Route Attestations* (RA)), welche dem Empfänger erlaubt die Routen an seine Nachbarn zu annoncieren. Die enthaltenen RAs werden bei Empfang nicht aus den BGP-Nachrichten entfernt und es entsteht eine kaskadierte Liste von Signaturen, welche den Pfad der *BGP-UPDATE*-Nachrichten abbildet. Jeder S-BGP-Router kann überprüfen, ob der Pfad im `AS_PATH`-Attribut mit

der Reihenfolge der RAs übereinstimmt. Zur Absicherung des Transport-Protokolls gegen Angriffe nutzt S-BGP als Vermittlungsschicht-Protokoll IPsec. IPsec gewährleistet die Vertraulichkeit und Integrität der übertragenen Daten mittels Kryptographie und schützt vor den, im vorherigen Kapitel, vorgestellten TCP-Attacken.

S-BGP bietet eine größtmögliche Absicherung des BGP, konnte sich bis heute allerdings, aufgrund der nachfolgenden Punkte, nicht durchsetzen:

- Die Validierung der S-BGP Zertifikate verursacht CPU-Last und für die Haltung der Zertifikate wird zusätzlicher Speicher auf den BGP-Routern benötigt. Um S-BGP nutzen zu können, müssten viele Router aufgerüstet oder durch neue Modelle ersetzt werden, wodurch eine finanzielle Investition der Betreiber notwendig wird.
- Der erhöhte Zeitaufwand zur Validierung der BGP-Updates hat zur Folge, dass Routenaktualisierungen etwa das Doppelte der Zeit benötigen, um im Internet verteilt zu werden [34].
- Für die Speicherung der Zertifikate pro BGP-Verbindung würden etwa 30-35 MB zusätzlicher Arbeitsspeicher benötigt werden [44]. Bei kleinen autonomen Systemen, welche nur 2-3 Verbindungen zu anderen BGP-Routern aufrechterhalten, stellt dies kein Problem da. Allerdings halten die größten ASe Verbindungen zu ca. 3000 BGP-Routern aufrecht [25], was einen zusätzlichen Speicherbedarf auf den Routern von bis zu circa 103 GB bedeuten würde. Derzeitig erwerbbare Hardware-Router werden ohne Massenspeicher ausgeliefert und unterstützen daher solche Mengen an Arbeitsspeicher nicht.

3.2.2. Resource Public Key Infrastructure

Die *Secure Inter-Domain Routing* (SIDR) Arbeitsgruppe der IETF wurde 2009 gegründet, um Verfahren zur Verbesserung der BGP-Sicherheit auszuarbeiten. In der Zielbeschreibung der SIDR Arbeitsgruppe werden folgende Schwachstellen genannt, für welche Lösungen ausgearbeitet werden[19]:

- Die Annoncierung von Pfaden für IP-Präfixen, welche nicht vom annoncierenden autonomen System registriert wurden (Präfix-Hijacking).
- Die Abweichung des beschriebenen Pfades im `AS_PATH` Attribut vom tatsächlichen Pfad, den die *BGP-UPDATE*-Nachricht traversiert hat.

Mithilfe der in diesem Kapitel vorgestellten *Ressource Public Key Infrastructure* (RPKI) sollen Präfix-Hijacks verhindert werden. Hierzu wird wie bei S-BGP die hierarchische Vergabestruktur der IP-Adressen in eine PKI abgebildet. Die IANA bildet die Wurzel-CA und

autorisiert untergeordnete Subjekte mithilfe von kryptographischen Zertifikaten die Benutzung von IP-Präfixen oder AS-Nummern. Um die Ressourcenvergabe in der PKI nachzubilden, sind folgende Objekte und Zertifikate definiert worden [43]:

Ressourcen-Zertifikate (RC) enthalten eine Liste von IP-Präfixen oder AS-Nummern sowie die Identifikationsnummer (ID) eines öffentlichen Schlüssel. Dem Eigentümer des öffentlichen Schlüssels wird mit dem Zertifikat die Nutzung der IP-Präfixe und AS-Nummern gewährt. Ein Ressourcen-Zertifikat ist ein CA-Zertifikat und ermöglicht dem Besitzer die Delegierung, der im Zertifikat enthaltenen Ressourcen an Dritte.

Route-Origin-Authorization-Objekte (ROA) enthalten eine Menge von IP-Präfixen mit minimaler und optionaler maximaler Längenangabe sowie genau eine AS-Nummer. Das autonome System, mit der im ROA Objekt genannten ASN, ist berechtigt Routen für die enthaltenen IP-Präfixe zu annoncieren. Um das ROA Objekt kryptographisch abzusichern wird es mit einem End-Entity Zertifikat signiert. Der öffentliche Schlüssel des End-Entity-Schlüsselpaares wird bei der Signierung an das ROA-Objekt angehangen.

End-Entity-Zertifikate (EE) werden mit einem Ressourcen-Zertifikat signiert und enthalten eine Liste von IP-Präfixen oder ASNs sowie ein Ablaufdatum. EE Zertifikate dienen ausschließlich der Signierung von anderen Objekten. Bei der Verwendung in einer RPKI, sollte der private Schlüssel des EEs nur einmalig zur Signierung eines Objektes benutzt werden und danach gelöscht werden. Ein mit einem EE-Zertifikat signiertes Objekt wird ungültig sobald das Ablaufdatum des EE-Zertifikats erreicht ist oder das EE-Zertifikat in die Certificate-Revocation-List der CA aufgenommen wurde.

Manifest Ein Manifest dient dem Sicherherstellen der Integrität eines Zertifikat-Repositories. Es enthält den Dateinamen und Hashwert aller signierten Objekte im Repository und wird mit einem EE-Zertifikat signiert.

Die IANA verfügt über ein selbst ausgestelltes Ressourcenzertifikat für den vollständigen IP-Adress- und ASN-Bereich. Für IP-Präfixe und ASNs, welche an die RIRs delegiert werden, stellt die IANA Ressourcenzertifikate aus, die sie mit ihrem eigenem Ressourcenzertifikat signiert. Die RIRs können wiederum ASNs und IP-Präfixe an untergeordnete Stellen vergeben, indem sie ein signiertes RC-Zertifikat ausstellen. Um die Annoncierung eines IP-Präfixes mithilfe der RPKI abzusichern, erstellt der Präfix Eigentümer zuerst ein EE-Objekt, welches die IP-Präfixe enthält, die annonciert werden sollen. Das EE-Zertifikat signiert der Eigentümer mit einem RC-Zertifikat. Anschließend wird ein ROA-Objekt ausgestellt, dass die ASN des BGP-Routers enthält sowie die Präfixe, die annonciert werden sollen. Mithilfe des zuvor erstellten EE-Zertifikat wird das ROA-Objekt signiert.

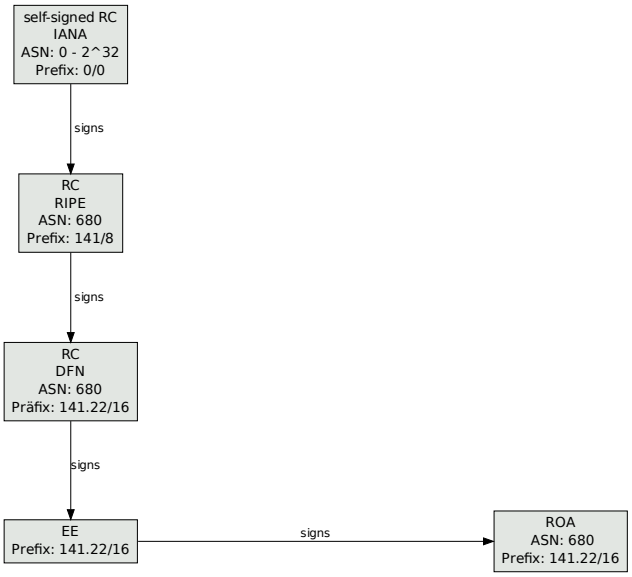

Abbildung 3.3.: Beispiel der RPKI-Zertifikatsvergabe

Ein BGP-Router ist autorisiert, eine Route für ein Präfix zu annoncieren, wenn ein ROA Zertifikat für die Route existiert, dessen Zertifikatskette erfolgreich bis zum Ressourcenzertifikat der IANA validiert werden kann. Eine exemplarische Zertifikatskette ist in Abbildung 3.3 dargestellt. Die IANA genehmigt der RIPE die Nutzung der ASN 680 und des IP-Netzes 141.0.0.0/8, indem sie ein signiertes Ressourcen-Zertifikat ausstellt. Die RIPE delegiert die ASN 680 sowie den Teiladressbereich 141.22.0.0/16 an das DFN, indem sie diesem wiederum ein Ressourcenzertifikat ausstellt. Für die Signierung eines ROA Objekts erstellt das DFN ein End-Entity Zertifikat, welches das IP-Präfix 141.22.0.0/16 enthält und mit dem DFN Ressourcen-Zertifikat unterschrieben wird. Mit dem EE-Zertifikat wird ein ROA-Objekt signiert, welches die Annoncierung einer Route für das IP-Präfix 141.22.0.0/16 dem AS 680 genehmigt.

Damit andere BGP-Router, Annoncierungen mithilfe der RPKI validieren können, werden die Zertifikate auf öffentlich zugreifbaren Repository-Servern hinterlegt. Die Repository-Server werden von den IP-Adress-Registraturen (RIRs, NIRs, LIRs) betrieben, jeder Repository-Server enthält mindestens alle EE, RC, CRLs und Manifests die von der Registratur unterschrieben worden sind. Die Repositories der LIRs enthalten zusätzlich alle ROA Objekte für die von der Registratur vergebenen IP-Bereiche. Um zentrale Anlaufstellen anzubieten, auf denen alle Zertifikate gesammelt zur Verfügung stehen wird empfohlen, dass alle Repositories zusätzlich die Zertifikate aller untergeordneten Stellen enthalten. Zum Abruf der Zertifikate von den Repositories wird das RSYNC-Protokoll [17] genutzt,

welches eine inkrementelle Datenübertragung von Aktualisierungen des Datenbestandes ermöglicht.

Um die BGP-Router möglichst wenig zu belasten, werden die Zertifikate nicht von den Routern selber gespeichert und validiert, sondern von externen Cache-Servern (RTR-Servern) (siehe Abbildung 3.4). Jedes autonome System verfügt über ein oder mehrere Cache-Server, welche mit einer Auswahl an Zertifikatsrepositories konfiguriert sind. Per RSYNC laden die Cache-Server die Zertifikate von den Repositories und synchronisieren ihren Datenbestand fortlaufend. Die heruntergeladenen Zertifikate werden auf den Cache-Servern validiert und es wird eine Liste mit validen ASN und IP-Präfix Kombinationen erstellt. Zur Validierung von BGP-Routen gegen den RPKI-Datenbestand bauen die Router eine Verbindung mithilfe des RPKI-RTR-Protokolls zu den Cache-Servern auf und laden die validierten Datensätze herunter. Die Liste der IP-Präfixe und AS-Nummern speichert der Router in einer lokalen Datenstruktur. Bei Empfang einer Pfad-Annoncierung durchsucht der BGP-Router die Datenstruktur nach passenden Einträgen, welche für den annoncierten IP-Bereich gelten. Falls die Datenstruktur einen Eintrag für das Herkunfts-AS und das IP-Präfix des Pfades enthält, kann der BGP-Router die Route bedenkenlos in seine Routing-Tabelle aufnehmen. Wenn ein Eintrag für das Präfix mit einem anderen Herkunfts-AS oder nicht übereinstimmendem maximalen Längenattribut existiert, handelt es sich um ein nicht authorisierte Annoncierung und die Route sollte vom weiteren Entscheidungsprozess ausgeschlossen werden. Falls dass Präfix der Route von keinem Validierungsdatensatz abgedeckt wird ist es unklar, ob es sich um eine valide Annoncierung handelt. In diesem Fall kann der Präferenzwert der Route angepasst werden, damit andere valide Routen für das Präfix präferiert werden.

Mithilfe des vorgestellten RPKI Konzepts ist eine Validierung von Routen-Annoncierungen möglich. RPKI erfüllt dieselbe der Aufgabe wie die Adress Attestations in S-BGP. Durch die Auslagerung und Validierung der Zertifikate auf einem separaten Cache-Server entsteht allerdings nur eine geringe Zusatzlast auf den BGP-Routern. RPKI kann im Internet inkrementell installiert werden. Die Effektivität wird nicht beeinflusst durch die Anzahl der Router, die die Erweiterung nutzen. AS-Betreiber, welche die Kosten für die RPKI-Installation nicht aufbringen möchten, weil sie eventuell die Bedrohung von BGP-Angriffen als nicht signifikant ansehen, können auf RPKI verzichten. Andere AS-Betreiber, welche Sicherheit einen hohen Wert zuschreiben, können RPKI in ihren BGP-Routern nutzen. Die Effektivität von RPKI ist allerdings abhängig von den verfügbaren ROA-Objekten in den Repository-Servern. Nur wenn möglichst alle IP-Präfix-Eigentümer ROAs für Routen-Annoncierungen erstellen und öffentlich verfügbar machen können Routen, welche durch die RPKI nicht validiert werden können von den BGP-Routern ignoriert werden. Anderenfalls kann nur der Präferenz-Wert von Routen herabgestuft werden, was nach wie vor zur Nutzung von gefälschten Routen führen kann. Am 24.10.2011, verzeichnete die RIPE 1008 IPv4-Präfixe [26] welche durch RPKI validiert werden konnten. Routing-Tabellen von BGP-Routern, enthielten zu dem Zeitpunkt 336.072 Routen [24] für IPv4-Präfixe. Daraus-

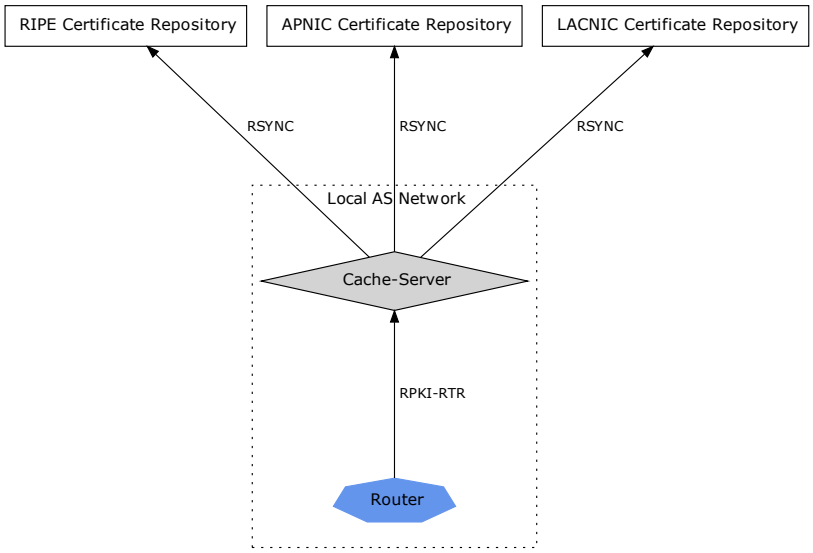

Abbildung 3.4.: RPKI-Struktur

folgt das mithilfe RPKI zurzeit nur ca. 0,3% der IPv4-Routen validiert werden können[1]. Aufgrund der Ankündigung von Router-Herstellern, welche RPKI in ihren Routern implementieren wollen [40] sowie den RIRs die bereits jetzt schon Repositorityserver betreiben, kann damit gerechnet werden dass das Interesse an RPKI in Zukunft steigt und für mehr Routen ROAs angelegt werden.

3.2.3. BGPsec

BGPsec [48] ist die zweite Sicherheitserweiterung die von der SIDR Arbeitsgruppe entworfen wird. Mithilfe BGPsec wird garantiert, dass jedes AS die Annoncierung der *BGP-Update*-Nachricht zum nachfolgenden AS autorisiert hat, sowie dass die Einträge im `AS_PATH` Attribut den tatsächlich traversierten autonomen Systemen entspricht. Hierzu wird das BGP um das optionale Attribut `BGP_SEC_Path_Signatures` erweitert und jeder BGPsec-Router bekommt ein End-Entity-Zertifikat ausgestellt, welches seine ASN enthält. Wenn ein Router eine neue Route annonciert oder eine *BGP-Update*-Nachricht an sei-

[1] Bei der Berechnung wurde angenommen, dass pro Route ein Validierungsdatensatz existiert. Es könnten viel weniger Datensätze nötig sein, da mit einem Datensatz mehrere Routen positiv validiert werden können.

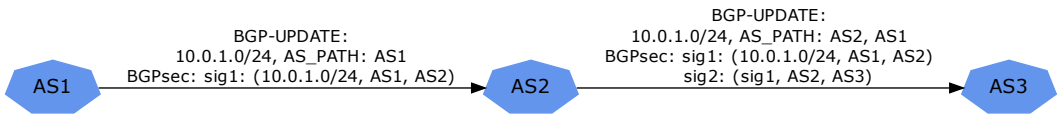

Abbildung 3.5.: BGPsec

ne BGP-Nachbarn weiterleitet, bildet er mithilfe seines EE-Zertifikats eine Signatur über folgende Attribute:

- die enthaltenen IP-Präfixe,
- der Herkunfts-ASN der Route,
- der AS-Nummer des BGP-Routers an den die Nachricht bestimmt ist,
- das Verfallsdatum der BGPsec Nachricht,
- sowie über eine ID welche den verwendeten Signierungsalgorithmus identifiziert.

Die erstellte Signatur wird zusammen mit dem Verfallsdatum an das `BGPSEC_Path_Signatures` Attribut der BGP-Nachricht angehängt. Jeder Router der die *BGP-UPDATE* Nachricht an seine Nachbarn weiterleitet, erweitert die Nachricht um seine BGPsec Signatur. Es entsteht eine Liste von Signaturen, mit der jeder BGP-Sprecher überprüfen kann, ob die Einträge im `AS_PATH`-Attribut den Daten in der Signaturkette entsprechen (siehe Abbildung 3.5). Das Ablaufdatum in den BGPsec-Nachrichten definiert eine maximale Zeitspanne, in der die Nachricht gültig ist. Wenn das Ablaufdatum erreicht ist, wird die annoncierte Route aus der Routing-Tabelle des Routers entfernt. Hiermit wird ein gewisser Schutz vor Angriffen erreicht, bei der ein Router eine vorher annoncierte Route zurückziehen will, aber die entsprechende *BGP-UPDATE* Nachricht von einem Router nicht an seine Nachbarn weitergeleitet wird.

BGPsec gewährleistet, dass Modifikationen des `AS_PATH`-Attributs von den BGP-Nachbarn entdeckt werden und entsprechend behandelt werden können. Zusätzlich wird verhindert dass ein Angreifer eine Route, eines fremden Präfixes mit korrektem Herkunfts-AS annonciert und den empfangenen Datenverkehr abhört, modifiziert oder verwirft. Die Validierung und Speicherung der Zertifikate bei BGPsec müssten eine ähnliche Belastung für die Router zur Folge haben wie S-BGP. Aufgrund des Alters des Konzepts sind die Hardwareanforderungen von BGPsec noch nicht bekannt. Die Anforderungen im “Security Requirements for BGP Path Validation„ Draft [31] schreiben aber vor, dass die Installationskosten und Hardwareanforderungen in einem vertretbaren Rahmen liegen sollen. Falls sich herausstellen

sollte, dass die verursachte Hardwarelast von BGPsec zu groß ist, könnte dass BGP Secure Routing Extension (BGP-SRx) Framework [3] als Abhilfe eingesetzt werden. BGP-SRx minimiert die Hardwarelast, die durch RPKI-RTR und BGPsec auf dem Router entstehen. Hierfür werden die BGPsec-Zertifikate und RTR-Validierungsdatensätze auf einen speraten Server ausgelagert. Wenn Nachrichten vom BGP-Router signiert oder validiert werden müssen, wird die Operation an den SRx-Server delegiert, von diesem ausgeführt und dass Ergebniss wird zurück an den Router übertragen.

3.3. Zusammenfassung

BGP verfügt ist anfällig für eine Reihe von Sicherheitslücken, welche die Stabilität und Funktionsfähigkeit des Internets gefährden. Mit S-BGP steht seit längerer Zeit eine Sicherheitserweiterung zur Verfügung, welche die vorgestellten Probleme löst, sich aber bis heute nicht durchsetzen konnte. Die Entwürfe der SIDR Arbeitsgruppe, versprechen eine Absicherung gegen die vorgestellten Sicherheitsprobleme mit minimaler Hardwarelast sowie der Möglichkeit der inkrementellen Installation im Internet. Die Sicherheitserweiterungen der SIDR-Arbeitsgruppe umgehen die Probleme, die bei S-BGP auftraten. Von den RIRs werden bereits heute Zertifikatsrepositories angeboten mit denen RPKI genutzt werden kann. Da auch Hardware-Router-Hersteller eine Implementierung von RPKI angekündigt haben ist es wahrscheinlich, dass die Sicherheitserweiterung zukünftig von einer großen Anzahl von BGP-Routern genutzt wird.

4. RPKI-RTR Protokoll

Das RPKI-RTR Protokoll spezifiziert den Datenaustausch von Validierungsdatensätzen zwischen einem Router (RPKI-Client) und einem RPKI-Cache (RTR-Server). Als Grundlage für diesen Abschnitt dient die Version 18 des IETF-Drafts „The RPKI/Router Protokoll“ [33]. Im ersten Kapitel werden die möglichen Transportprotokolle beschrieben, über welches das RPKI-RTR Protokoll gesprochen werden kann. Anschliessend folgt eine Erläuterung des Protokollablaufs und der Behandlung von Fehlerzuständen. Im letzten Kapitel wird der Ablauf von BGP-Präfix Validierungen beschrieben.

4.1. Transportprotokolle

Das RPKI-RTR Protokoll ist in eine Menge von Nachrichten (*Protocol Data Units* (PDUs)) unterteilt, welche zwischen einem RTR-Client und -Server ausgetauscht werden. Der Draft nennt mehrere Transportprotokolle, welche zur Übertragung der PDUs zwischen Server und Client genutzt werden können. Jede RTR-Implementierung muss als Mindestanforderung TCP [54] zum Übertragen der PDUs unterstützen. TCP wird von jedem gängigen netzwerkfähigem Betriebssystem unterstützt, allerdings bietet es weder Authentifizierung der Kommunikationspartner noch Schutz der Integrität der übertragenen Daten. Um das komplette RPKI-System von der Ausstellung der Zertifikate bis zur Übertragung der Validierungsdatensätze zum Router abzusichern, ist ein sicheres Transportprotokoll für RPKI-RTR nötig. Anderenfalls besteht die Gefahr, dass ein Angreifer Validierungsdatensätze während der Übertragung zwischen RTR-Server und -Client modifiziert, löscht oder Datensätze in den RTR-Client einschleust, für die kein ROA-Objekt existiert. Dies hätte zur Folge, dass der BGP-Router Pfade falsch bewertet und evtl. Routen für gehijackte Präfixe verwendet. Da die Betriebssysteme von Hardware-Routern nicht einheitlich und nur mit minimalem Funktionsumfang ausgestattet sind, ist es schwierig ein einziges sicheres Transportprotokoll vorzuschreiben, welches auf allen Routern implementiert werden kann. Im IETF-Draft werden folgende sichere Transportprotokolle genannt, welche von einer RPKI-RTR Implementierung für die Datenübertragung genutzt werden können[1].

[1] Während der Entwicklung des Drafts, hat die Auswahl eines sicheren Transportprotokolls für kontroverse Diskussionen gesorgt, weitere Details sind unter [7] nachlesbar.

SSH [63] bietet Authentifizierung der Kommunikationspartner und verschlüsselte Datenübertragung. SSH ist weit verbreitet und das de facto Standard-Protokoll für Fernwartung bei Unix-basierenden Betriebssystemen. Es existieren mehrere frei verfügbare SSH-Bibliotheken, welche für eine Implementierung genutzt werden können. Anwendungen, die ein eigenes Protokoll sprechen, können als SSH-Subsystem von einem SSH-Server ausgeführt werden. Alle Ausgaben des SSH-Subsystem Programmes werden, werden über die SSH-Verbindung zum Kommunikationspartner übertragen. Vom SSH-Server empfangene Daten, werden über den Standarteingabe-Kanal an das Programm weitergegeben. Da Hardwarerouter sehr minimal gehalten sind verfügen sie oft über keine SSH-Bibliothek, welche zur Implementierung genutzt werden kann. SSH operiert auf der Anwendungsschicht des OSI-Modells, für die Nutzung wird separate Anwendungssoftware benötigt und es entsteht im Vergleich zu Protokollen auf einer niedrigeren OSI-Schicht zusätzlicher Overhead.

TCP-MD5 [41] bietet Authentifizierung der Kommunikationspartner und schützt die Integrität der übertragenen Daten. es ist allerdings anfällig für TCP-Replay-Attacken [57]. Das TCP-MD5 RFC [41] ist als obsolet gekennzeichnet und wurde durch das TCP-AO [56] ersetzt.

TCP-AO[56] wird als sicheres Transportprotokoll für RPKI-RTR empfohlen. Es ist der Nachfolger des TCP-MD5 Protokolls und bietet Schutz der Integrität der übertragenen Daten, inkl. des TCP-Headers, Authentifizierung der Kommunikationspartner und schützt vor Packet-Injection Angriffen. Da, TCP-AO erst 2010 in einem RFC spezifiziert worden ist verfügen heutzutage nur wenige Betriebssysteme über eine TCP-AO Implementierung. Wenn TCP-AO von einer größeren Anzahl von Betriebssystemen unterstützt wird, wird TCP-AO höchstwahrscheinlich dass Standard-Transportprotokoll für RPKI-RTR werden [33].

TCP-TLS [38] bietet Authentifizierung der Kommunikationspartner, verschlüsselte Datenübertragung und schützt die Integrität der übertragenen Daten. Bietet allerdings keinen Schutz vor Verfälschungen des TCP-Headers und ist anfällig für DoS-Attacken durch Packet-Injection.

IPsec [46] arbeitet auf Schicht 3 des OSI-Modells. Es authentifiziert die Kommunikationspartner, bietet verschlüsselten Datenaustausch und schützt die Integrität des kompletten IPsec-Paketes und dessen Payload. Es wird von vielen Betriebssystemen unterstützt ist allerdings komplex und aufwendig zu konfigurieren und kann nur mit Sicherheitseinschränkungen in NAT-Umgebungen benutzt werden [29].

4.2. Protokollablauf

Der RTR-Client wird mit mehreren RTR-Servern konfiguriert, welche mit einem Präferenzwert priorisiert sind. Aus der Konfiguration werden die RTR-Server mit dem niedrigsten Präferenzwert ermittelt und die Transportverbindung zu den Servern aufgebaut. Nachdem der Client eine Verbindung zum RTR-Cache-Server aufgebaut hat, sendet der Router eine *Reset Query* PDU an den Cache-Server, um eine neue Sitzung zu starten. Der Cache-Server antwortet mit einer *Cache Response* PDU, welche eine Nonce[2]) für die Identifizierung der Sitzung enthält. Im Anschluss überträgt der Cache-Server die Valdierungsdatensätze. Diese werden als *IPv4* oder *IPv6 Prefix* PDUs übertragen und enthalten ein IPv4 bzw. IPv6 Präfix, die minimale und maximale Präfixlänge und eine ASN, welche das Präfix als BGP-Router annoncieren darf. Die übertragenen Datensätze werden vom Client in einer lokalen Datenstruktur gespeichert, die *Prefix Validation Table* (pfx_table) genannt wird [51]. Nachdem alle Validierungsdatensätze übertragen worden sind, sendet der Server eine *End of Data* PDU. Diese teilt dem Client mit dass die Datensynchronisation beendet ist und enthält eine Seriennummer, die den übertragenen Datenbestand identifiziert.

Nach erfolgter Synchronisierung startet der Client einen internen Timer. Wenn dieser abgelaufen ist, sendet der Client einen *Serial Query*, welche die zuletzt empfangene Seriennummer enthält, um beim Server abzufragen, ob aktualisierte Datensätze vorliegen. Der Server antwortet mit einer *Cache Response* PDU, welche die zur Sitzungsidentifizierung verwendete Nonce enthält. Falls die enthaltene Nonce nicht mit der zuvor zugewiesenen Nonce übereinstimmt wird eine Fehlermeldung an den Server gesendet und die Verbindung beendet. Wenn der Server über Änderungen des Datenbestandes verfügt, werden diese an den Client übertragen und die Datenübertragung mit einer *End of Data* PDU abgeschlossen. Falls der Client bereits über den aktuellen Datenbestand verfügt, werden keine *IPv4* und *IPv6 Prefix* PDUs übertragen, sondern die Synchronisation mit einer *End of Data* PDU beendet. Die *End of Data* PDU enthält in diesem Fall die gleiche Seriennummer, wie die *Serial Query* PDU mit der die Aktualisierung eingeleitet wurde. kennzeichnet. Optional kann der Server wenn sich sein Datenbestand aktualisiert dem Client eine *Serial Notify* PDU zusenden, um ihm zu benachrichtigen, dass aktualisierte Daten vorliegen. Dieser kann daraufhin ein *Serial Query* an den Server senden, um die Datensynchronisation einzuleiten. Der Synchronisationsvorgang ist in Abbildung 4.1 dargestellt.

Für die Identifizierung des aktuellen Datenbestandes, über die der RTR-Client verfügt, wird nach jedem Datenabgleich eine Seriennummer übertragen. Dies ermöglicht dass bei Änderungen des Datenbestandes des Cache-Servers der Client seinen Datenbestand mit der Seriennummer identifiziert und die Server nur die Differenz zwischen den Datenbeständen überträgt. Eine komplette Neuübertragung des Datenbestandes ist nicht notwendig. Die

[2]Generierte Zufallszahl, welche möglichst nur einmalig verwendet wird

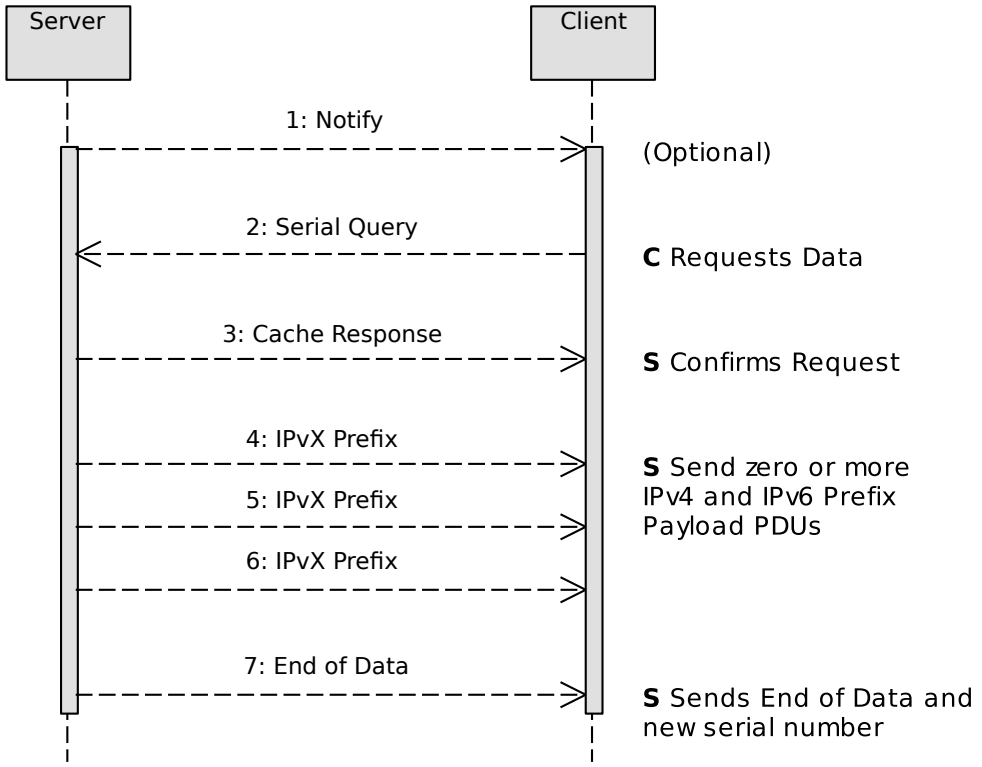

Abbildung 4.1.: RPKI-RTR Datensynchronisation

Seriennummern des RTR-Servers sind mit den Nonce-Nummern für die Sitzungsidentifizierung assoziert, um sicherzustellen dass z. B. bei einer neu gestarteten Transportverbindung die übertragene Seriennummer auch von dem selben Server vergeben worden ist. Falls ein Cache-Server keine Aktualisierungen zu einer Seriennummer zur Verfügung stellen kann, sendet der Cache-Server als Antwort auf eine *Serial Query* PDU eine *Cache Reset* PDU an den Client. Wenn möglich sollte der Client die Verbindung zu anderen RTR-Servern herstellen. Anderenfalls kann eine *Reset Query* PDU an den Server gesendet werden, um die Sitzung neu zu starten und den Datenbestand vollständig neu zu übertragen. Der Client kann für eine vollständige Neuübertragung aller Datensätze bei Sitzungsneustart einen *Reset Query* senden oder, was vom IETF-Draft empfohlen wird, wenn möglich die Verbindung zu anderen RTR-Servern herzustellen. Dieser Fehlerfall kann auftreten, wenn der RTR-Client zu lange kein Abgleich des Datenbestandes mit dem Server mehr durchgeführt hat, die übertragene Seriennummer nicht mit der Nonce assoziert ist oder der Server zwischenzeitlich Informationen zum Datenbestand der Seriennummer gelöscht hat. Falls der Cache-Server zur Zeit über keine Validierungsdatensätze verfügt, beispielsweise weil er neu gestartet worden ist und die Synchronisation mit den Repository-Servern noch nicht abgeschlossen wurden ist, antwortet der Cache-Server auf *Reset* und *Serial Query* PDUs

mit einer *Error* PDU. Diese enthält ein Feld, welches den Typ des Fehlers identifiziert, optional eine Kopie der empfangenen PDU, die den Fehler ausgelöst hat sowie einen optionalen Text, der den Fehler beschreibt. *Error* PDUs werden vom Client und Server versandt, sobald Fehler auftreten, die den weiteren Protokollablauf beeinflussen. Beim Auftreten von kritischen Fehlern stellt der RTR-Client, wenn er mit mehreren Cache-Servern konfiguriert wurde, eine Verbindung zu den Cache-Servern mit dem nächsthöheren Präferenzwert her. Alte Validierungsdatensätze werden gelöscht wenn eine erfolgreiche Verbindung zu den neuen Cache-Servern hergestellt worden ist oder die Datensätze seit einer konfigurierbaren Zeitspanne nicht mehr aktualisiert worden sind.

4.3. Validierung von BGP-Präfixen

Der Ablauf der Validierung von BGP-Routen, mithilfe der Validierungsdatensätze von einem RTR-Server ist im IETF-Draft „BGP Prefix Origin Validation“ [51] beschrieben. Eine BGP-Route kann mit einem der folgenden Status ausgezeichnet werden:

Not Found In der Prefix Validation Table ist kein Datensatz enthalten, welcher mit dem gesuchten IP-Präfix übereinstimmt oder dieses abdeckt.

Valid Es existiert ein Datensatz, welcher das gesuchte IP-Präfix abdeckt sowie eine Herkunfts-ASN enthält, die der Herkunfts-ASN der Route entspricht.

Invalid Es wurden Datensätze gefunden, welche das gesuchte IP-Präfix bis zur minimalen Längenangabe abdecken, aber das maximale Längenattribut ist kleiner als die Präfixlänge der Route oder das Herkunfts-ASN entspricht nicht der Route überein.

Falls Validierungsoperationen auf dem RTR-Client ausgeführt werden bevor eine Datensynchronisation mit dem RTR-Server abgeschlossen wurde werden alle Routen mit dem Status *Not Found* validiert. Für diese Fälle müssen von der RTR-Implementierung Mechanismen bereitgestellt werden, welche es ermöglichen eine erneute Validierung durchzuführen sobald eine Synchronisation erfolgreich durchgeführt wurde

5. Entwurf der Bibliothek

In dem folgendem Abschnitt wird der Architekturentwurf der Bibliothek erläutert. Im ersten Kapitel werden die Anforderungen beschrieben, welche die Architektur sowie die Implementierung erfüllen sollen. Anschließend wird der Entwurf der Gesamtarchitektur diskutiert. In den nachfolgenden Kapiteln werden die Anforderungen an die einzelnen Komponenten dargestellt und die Schnittstellen der Komponenten definiert.

5.1. Anforderungen

Die Bibliothek soll auf eine großen Anzahl von Unix-Betriebssystemen portierbar sein. Um die Portierbarkeit möglichst einfach zu gestalten, soll die Bibliothek Betriebssyytemunabhänge Schnittstellen nutzen (z. B. POSIX [53]). Da die Abhängigkeit von Software, welche wiederum nur auf bestimmten Hard- und Softwarekombinationen betrieben werden kann, die Portierbarkeit erschwert, sollen möglichst wenige fremde Bibliotheken und Anwendungen zur Installation und zum Betrieb benötigt werden. Die RPKI-RTR Spezifikation [33] befindet sich noch im Draft Status. Dass die Spezifikation zukünftig noch geändert wird ist nicht unwahrscheinlich. Damit Modifizierungen und Erweiterungen der Bibliothek einfach zu realisieren sind, soll eine Architektur entworfen und implementiert werden die Änderungen unterstützt. Fehler, welche bei der Ausführung von Operationen auftreten, sollten möglichst im Programmablauf behandelt werden. Falls dies nicht möglich ist, soll der Benutzer über das Auftreten von Fehlern informiert werden. Für Benutzer der Bibliothek soll die Verwendung möglichst einfach gestaltet sein. Wenn sinnvoll soll die Bibliothek Komfortfunktionen anbieten, welche nicht zwingend notwendig sind aber die Verwendung vereinfachen. Damit die Interoperabilität mit RTR-Servern gewährleistet ist, soll die Spezifikation des RPKI-RTR-Protokolls [33] streng befolgt und vollständig implementiert werden. Die Hardwareanforderung der Bibliothek sollten möglichst gering gehalten werden, sodass die Bibliothek auch auf älteren Rechnern lauffähig ist.

5.2. Architektur

Die Bibliothek wird in einzelne Komponenten aufgeteilt, welche hierarchisch aufeinander aufbauen und über klar definierte Schnittstelle miteinander kommunizieren. Durch die Aufteilung der Bibliothek in Komponenten und Schnittstellen wird die unabhängige Entwicklung von Programmteilen unterstützt und die Austauschbarkeit von einzelnen Komponenten erleichtert. Für die Eigenschaften der Komponenten gilt, dass diese eine möglichst hohe Kohäsion und eine niedrige Kopplung besitzen sollen. Eine hohe Kohäsion bedeutet, dass eine einzelne Komponente nur für eine klar definierte Aufgabe zuständig ist. Dies hat zur Folge, dass die Verständlichkeit der Komponenten erhöht und die Wartung erleichtert wird. Durch eine niedrige Kopplung zwischen den Komponenten werden die Abhängigkeiten untereinander minimiert und einzelne Komponenten können verändert werden, ohne dass dies Änderungen in anderen Komponenten erfordert.

Der hierarchische Aufbau der Bibliothek ist in Abbildung 5.1 dargestellt. Der Transport-Socket (`tr_socket`) ist die unterste Komponente in der Architektur und ist zuständig für den Transfer von Daten über einen Kommunikationskanal zwischen dem RTR-Server und -Client. Die überliegende RTR-Socket (`rtr_socket`) Komponente, ist zuständig für den RTR-Protokollablauf zwischen einem Client und einem Server. Für den Versand und Empfang der PDUs nutzt der `rtr_socket` einen Transport-Socket Datenkanal. Aus den empfangenen *IPv4* und *IPv6 Prefix* PDUs extrahiert der `rtr_socket` die Validierungsdatensätze und speichert sie in einer Datenstruktur, welche in der Prefix Validation Table (`pfx_table`) Komponente implementiert ist. Abgesehen von für die Verwaltung der Datensätze benötigte Operationen, wie z. B. Hinzufügen und Löschen eines Datensatzes, stellt die `pfx_table` Komponente Operationen zur Validierung von BGP-Routen bereit.

Die oberste Komponente der Architektur bildet der RTR Connection Manager (`rtr_mgr`). Dieser gruppiert eine Menge von RTR-Sockets in einer Konfiguration und koordiniert den Verbindungsauf- und -abbau der einzelnen Sockets, um Failover-Funktionalität bereitzustellen. Falls verbundene RTR-Sockets den Auftritt eines Fehlers melden, wird dieser vom `rtr_mgr` behandelt, indem Verbindungen zu den Ersatzservern einer andere Konfiguration aufgebaut werden.

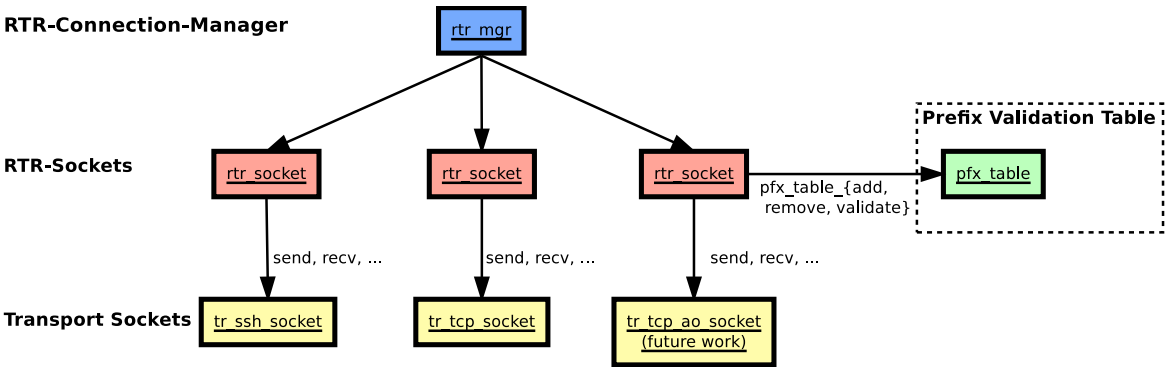

Abbildung 5.1.: Komponenten der RPKI-RTR Bibliothek

5.3. Transport-Socket

Die Transport-Socket Komponente stellt Kommunikationsprimitiven zur Verfügung, die zum Übertragen von Daten zwischen zwei Endpunkten benutzt werden können. In der RPKI-RTR Spezifikation [33] werden mehrere Netzwerkprotokolle genannt, die zum Übertragen der PDUs vorgesehen sind. Als Minimalanforderung muss eine Implementierung TCP für den Austausch der PDUs unterstützen. Mit TCP steht ein Protokoll zur Verfügung, welches auf jedem Unix-basierten System über standardisierte POSIX-Schnittstellen [53] implementiert werden kann und ohne zusätzliche Bibliotheken lauffähig ist. Aufgrund der unverschlüsselten Datenübertragung bei einer TCP-Verbindung kann der TCP-Transport-Socket beim Auftreten von Protokollfehlern zur Lokalisierung der Ursache behilflich sein. Mithilfe eines Paket-Sniffers kann der Datenverkehr zwischen einem RTR-Server und -Client protokolliert und durch eine nachfolgende Analyse des Mitschnitts die Problemursache eingegrenzt werden.

Um Benutzern der Bibliothek auch ein Transportmedium anbieten zu können, welches einen authentifizierten und verschlüsselten Datentransfers zwischen RTR-Server und -Client ermöglicht, soll das SSH-Protokoll als Transport-Socket implementiert werden. SSH wurde als Transportprotokoll ausgewählt, da es das einzige sichere Protokoll ist, welches von einem frei verfügbaren RTR-Server (*rtr-origin* [6]) unterstützt wird. Die verschiedenen Netzwerkprotokolle, welche von der Transport-Socket Komponente unterstützt werden, sollen möglichst einheitlich und unabhängig vom verwendeten Protokoll bedienbar sein. Spezielle Eigenschaften des verwendeten Protokolls sollen abstrahiert und in allgemeinen Funktionen gekapselt werden. Dies ermöglicht eine einfache und benutzerfreundliche Bedienung, welche kein spezielles Wissen zur Verwendung des Protokolls erfordert.

Der protokollunabhängige Datenkanal wird in der Komponente durch eine `tr_socket` Datenstruktur (siehe 5.1) abgebildet. Diese kapselt alle für die Datenübertragung benötigten

Daten und enthält Funktionszeiger, welche zur Ausführung von im RTR-Kontext benötigten Operationen aufgerufen werden können.

```
1  typedef void (*tr_close_fp)(void* socket);
2  typedef int (*tr_open_fp)(void* socket);
3  typedef void (*tr_free_fp)(struct tr_socket* tr_sock);
4  typedef int (*tr_recv_fp)(const void* socket, void* pdu, const
     size_t len, const time_t timeout);
5  typedef int (*tr_send_fp)(const void* socket, const void* pdu,
     const size_t len, const time_t timeout);

7  typedef struct tr_socket{
8      void* socket;
9      tr_open_fp open_fp;
10     tr_close_fp close_fp;
11     tr_free_fp free_fp;
12     tr_send_fp send_fp;
13     tr_recv_fp recv_fp;
14 } tr_socket;
```

Listing 5.1: tr_socket

Die für den Betrieb des Protokolls notwendigen Daten können von der Implementierung im `socket` Element gespeichert werden. Alle Funktionszeiger nehmen als erstes Argument einen Zeiger vom Datentyp `void*` entgegen, welcher ein `tr_socket.socket` Element referenziert auf dem die Operation durchgeführt wird. Bevor Daten über den Transport-Socket übertragen werden können, muss durch einen Aufruf des Funktionszeigers `tr_socket.open_fp` der Kommunikationskanal zur Gegenstelle aufgebaut werden. Ein Kommunikationskanal kann geschlossen werden, indem der Funktionszeiger `tr_socket.close_fp` aufgerufen wird. Der `tr_socket.free_fp` Funktionszeiger referenziert eine Funktion, welche jeglichen Speicherplatz freigibt, welcher bei der Benutzung des Transport Socket alloziert wurde. Von dem `tr_socket.recv_fp` Element wird eine Funktion referenziert, die Daten vom Socket empfängt. Nach einem erfolgreichen Funktionsaufruf sind die empfangenen Daten im `pdu` Datenbereich gespeichert. Die Länge des Speicherbereiches wird der Funktion im `len` Parameter übergeben. Der `timeout` Parameter beschreibt wie lange die Empfangsfunktion in Sekunden blockiert bevor `len` Bytes empfangen worden sind. Falls der Timeout abgelaufen ist bevor `len` Byte Daten empfangen wurden, soll die Funktion mit einem Fehlerwert zurückkehren.

Die Sendefunktion, die der `tr_socket.send_fp` Zeiger referenziert, verhält sich analog zu der erläuterten Empfangsfunktion statt Daten zu empfangen, sollen `len` Bytes aus dem Speicherbereich `pdu` über den Socket versendet werden. Um Operationen auf einem `tr_socket` zu erleichtern, werden Hilfsfunktionen implementiert, welche einen `tr_socket`

übergeben bekommen und je nach Funktionsnamen (z. B. `tr_send(..)`) den entsprechenden Funktionszeiger aus der Struktur aufrufen. Da die vom Betriebssystem bereitgestellten Empfangs- und Sendefunktionen ([53]) erfolgreich zurückkehren können bevor die übergebene Menge von Daten empfangen bzw. gesendet wurden sollen die Funktionen `tr_recv_all(..)` und `tr_send_all(..)` implementiert werden. Diese sollen garantieren, dass beim Senden bzw. Empfangen die Funktion blockiert bis entweder `len` Bytes Daten übertragen wurden, der übergebene `timeout` überschritten oder ein Fehler aufgetreten ist.

5.4. RTR-Socket

Die **`rtr_socket`** Komponente implementiert das RTR-Protokoll [33] für die Übertragung der Validierungsdatensätze zwischen einem RTR-Server und -Client. Der Protokollablauf des RTR-Sockets soll selbständig im Hintergrund ablaufen und keine Benutzerinteraktion erfordern. Beim Auftreten von Verbindungs- oder Protokollfehlern soll die Verbindung zum Server nach Ablauf eines Timers neu gestartet werden. Eine Instanz eines RTR-Sockets wird mit der in Listing 5.2 dargestellten **`rtr_socket`** Datenstruktur abgebildet. Die Elemente der Datenstruktur dienen der Konfiguration der Sockets oder sind intern verwendete Daten, welche zum Implementieren des RTR-Protokolls benötigt werden.

```
1  typedef void (*rtr_connection_state_fp)(const struct
    rtr_socket* rtr_socket, const rtr_socket_state state, void*
    connection_state_fp_param);
2  typedef struct rtr_socket{
3      tr_socket* tr_socket;
4      unsigned int polling_period;
5      unsigned int cache_timeout;
6      time_t last_update;
7      rtr_socket_state state;
8      uint32_t nonce;
9      bool request_nonce;
10     uint32_t serial_number;
11     struct pfx_table* pfx_table;
12     pthread_t thread_id;
13     rtr_connection_state_fp connection_state_fp;
14     void* connection_state_fp_param;
15 } rtr_socket;
```

Listing 5.2: rtr_socket

Die einzelnen Elemente der **`rtr_socket`** Struktur dienen folgendem Zweck:

tr_socket referenziert einen initialisierten Transport-Socket, über welchen das RTR-Protokoll gesprochen werden soll.

polling_period beschreibt nach wievielen Sekunden nach der letzten Synchronisierung der Client eine *Serial Query* PDU an den Server sendet.

cache_timeout beschreibt nach wievielen Sekunden, ohne erfolgreiches Update, die empfangenen Validierungsdatensätze aus der `pfx_table` gelöscht werden.

last_update enthält den Zeitpunkt des letzten durchgeführten Datensatzabgleichs mit dem Server.

state speichert den derzeitigen Protokoll-Zustand des RTR-Sockets.

nonce speichert die aktuelle Sitzungs-Nonce, welche durch den RTR-Server zur Identifizierung der Client-/Server-Sitzung zugewiesen worden ist.

request_nonce speichert einen Booleschen Wert, welcher angibt ob eine neue Nonce vom RTR-Server beantragt werden muss oder die `rtr_socket.nonce` eine gültige Nonce enthält, welche für Anfragen verwendet werden kann.

serial_number enthält die Seriennummer, welche nach der letzten Datensynchronisation übermittelt worden ist.

pfx_table verweist auf eine `pfx_table` Struktur, in welcher die empfangenen Validierungsdatensätze verwaltet werden.

thread_id enthält die ID des Threads indem der RTR-Socket ausgeführt wird.

connection_state_fp verweist auf eine Funktion, welche bei einer Zustandsänderung des RTR-Sockets aufgerufen wird und den aktuellen Socketzustand übergeben bekommt.

fp_param speichert eine Adresse eines Datenbereiches, die der `connection_state_fp` Funktion beim Aufruf übergeben wird.

Eine `rtr_socket` Struktur wird mithilfe der `rtr_init(..)` Funktion initialisiert (siehe Listing 5.3). Als Parameter werden der Funktion die konfigurierbaren Elemente der `rtr_socket` Struktur übergeben. Alle anderen Elemente der Struktur werden mit ihren Standart-Werten initialisiert.

```
1 void rtr_init(rtr_socket* rtr_socket, tr_socket* tr, struct
   pfx_table* pfx_table, const unsigned int polling_period,
   const unsigned int cache_timeout, rtr_connection_state_fp fp,
   void* fp_data);
2 int rtr_start(rtr_socket* rtr_socket);
3 void rtr_stop(rtr_socket* rtr_socket);
```

Listing 5.3: rtr_socket init Funktion

Die Funktion `rtr_start(..)` baut die Transport-Verbindung zum RTR-Server auf und startet den RTR-Protokollablauf. Zum Trennen der Verbindung mit dem RTR-Server und Anhalten des Protokolls wird die Funktion `rtr_stop(..)` aufgerufen.

5.5. Prefix Validation Table

Die Prefix Validation Table (**`pfx_table`**) verwaltet alle vom RTR-Server empfangenen Validierungsdatensätze. Die Komponente stellt Operationen zum Hinzufügen und Löschen von Datensätzen sowie zum Validieren von BGP-Routen bereit. Um Seiteneffekte bei der parallelen Nutzung von mehreren RTR-Sockets einer **`pfx_table`** zu vermeiden, müssen die Funktionen thread-sicher implementiert werden. BGP-Router können über tausend Präfix Annoncierungen pro Sekunden empfangen, wie die „The BGP Instability Report "Statistiken [28] zu entnehmen ist, hat das AS 131072 am 7. November 2011 Routen-Annoncierungen für 1981 Präfixe innerhalb einer Sekunde empfangen. In den BGP-Routing-Tabellen werden zur Zeit mehr als 300.000 Pfade gespeichert [24]. Wenn zukünftig für jede Route ein Validierungsdatensatz existiert, muss die Bibliothek aus über 300.000 Einträgen 1981 mal pro Sekunde den Validierungsstatus der Route ermitteln. Damit dies ohne intensive CPU-Belastung möglich ist, muss eine performante Datenstruktur ausgewählt und implementiert werden, auf der die Validierungsoperationen mit $O(\log(N^1))$ Komplexität ausführbar sind. Der benötigte Speicherplatz zur Haltung der Datensätze soll linear skalieren.

Die Definition einer Schnittstelle für die Komponente ermöglicht es die Bibliothek zukünftig um weitere **`pfx_table`** Implementierungen zu ergänzen, die sich durch Performanz und Speichernutzung unterscheiden. Abhängig von der Betriebsumgebung kann dann die optimale **`pfx_table`** Implementierung gewählt werden. Die Schnittstelle der Komponente ist in Listing 5.4 dargestellt. Abgebildet wird die Prefix Validation Table durch die **`pfx_table`** Struktur. Die Komponente deklariert nur den Namen und Typ der Datenstruktur, allerdings keine Datenelemente. Die konkrete Deklaration wird der jeweiligen Implementierung überlassen und bietet somit größtmögliche Freiheit bei der Realisierung. Um eine **`pfx_table`** Struktur zu initialisieren, wird die Funktion `pfx_table_init(..)` bereitgestellt. Als Parameter nimmt diese einen Zeiger zur initialisierenden **`pfx_table`** Struktur und einen **`rtr_update_fp`** Funktionszeiger entgegen. Die Funktion, auf welche der **`pfx_update_fp`** Funktionszeiger verweist, wird für jeden **`pfx_record`** aufgerufen, welcher aus der **`pfx_table`** gelöscht oder dieser hinzugefügt wird. Der Callback kann genutzt werden, um z. B. zu überprüfen, welche Datensätze vom RTR-Server übertragen werden. Nach der Nutzung einer **`pfx_table`** Struktur kann jeglicher allozierter Speicher mit einem Aufruf der `pfx_table_free(..)` Funktion freigegeben werden.

[1] N entspricht der Anzahl der Datensätze.

```
1  struct pfx_table;
2  typedef enum {
3      BGP_PFXV_STATE_VALID,
4      BGP_PFXV_STATE_NOT_FOUND,
5      BGP_PFXV_STATE_INVALID
6  } pfxv_state;

8  typedef struct pfx_record{
9    uint32_t asn;
10   ip_addr prefix;
11   uint8_t min_len;
12   uint8_t max_len;
13   uintptr_t socket_id;
14 } pfx_record;

16 typedef void (*pfx_update_fp)(struct pfx_table* pfx_table,
    const pfx_record record, const bool added);

18 void pfx_table_init(struct pfx_table* pfx_table, pfx_update_fp
    update_fp);
19 void pfx_table_free(struct pfx_table* pfx_table);

21 int pfx_table_add(struct pfx_table* pfx_table, const
    pfx_record* pfx_record);
22 int pfx_table_remove(struct pfx_table* pfx_table, const
    pfx_record* pfx_record);
23 int pfx_table_src_remove(struct pfx_table* pfx_table, const
    uintptr_t socket_id);

25 int pfx_table_validate(struct pfx_table* pfx_table, const
    uint32_t asn, const ip_addr* prefix, const uint8_t mask_len,
    pfxv_state* result);
```

Listing 5.4: pfx_table

Einzelne Validierungsdatensätze werden durch die `pfx_record` Datenstruktur abgebildet. Im Element `socket_id` wird die Speicheradresse des RTR-Sockets, von welchem der Datensatz stammt, hinterlegt. Mithilfe dieser Assoziierung kann ein Datensatz dem RTR-Socket von dem er empfangen worden ist zugeordnet werden. Bei Beendigung des RTR-Sockets oder beim Auftreten von Fehlern ist es mithilfe der `socket_id` möglich, alle Datensätze zu löschen, welche von einem spezifischen RTR-Socket stammen. Die Benutzung der Speicheradresse eignet sich hierfür, weil diese eine eindeutige Identifizierung ermöglicht. Zum Hinzufügen und Entfernen von Datensätzen dienen die Funktio-

nen `pfx_table_add(..)` und `pfx_table_remove(..)`. Als Parameter nehmen die Funktionen ein Zeiger auf die **`pfx_table`** Struktur entgegen, auf der die Operationen ausgeführt wird sowie einen Zeiger auf einen **`pfx_record`**, der hinzugefügt bzw. entfernt werden soll. Falls ein Datensatz der hinzugefügt werden soll bereits in der **`pfx_table`** existiert soll die `pfx_table_add(..)` Funktion den Wert `PFX_DUPLICATE_RECORD` zurückgeben. Falls ein zu entfernender Datensatz nicht in der **`pfx_table`** gefunden werden kann, wird von der `pfx_table_remove(..)` Funktion `PFX_RECORD_NOT_FOUND` zurückgegeben. Die `pfx_table_remove_from_origin(..)` Funktion löscht alle Datensätze aus der **`pfx_table`**, die mit der übergebenen `rtr_socket_id` identifiziert werden können. Ob ein AS berechtigt ist eine Route für ein bestimmtes IP-Präfix zu annoncieren, kann mit der Funktion `pfx_table_validate(..)` überprüft werden. Diese nimmt eine ASN, ein Präfix, die Länge des Präfixes sowie einen Zeiger auf eine Variable vom Typ **`pfxv_state`** entgegen. Nach erfolgreicher Ausführung enthält die **`pfxv_state`** Variable den Validierungsstatus des übergebenen **`pfx_record`** Datensatzes.

5.6. RTR Connection Manager

Der RTR Connection Manager (**`rtr_mgr`**) realisiert den Verbindungsaufbau zu Gruppen von RTR-Servern sowie das automatisierte *Failover* beim Auftritt von Fehlern. Eine RTR-Servergruppe besteht aus einer Menge von RTR-Sockets sowie dem Präferenzwert der Gruppe. Wenn der RTR Connection Manager mit mehreren Servergruppen initialisiert wurde, soll beim Start eine Verbindung zu den RTR-Servern der Gruppe mit dem niedrigsten Präferenzwert hergestellt werden. Falls ein kritischer Fehler in einer der RTR-Verbindungen auftritt, wird die Gruppe mit dem nächsthöheren Präferenzwert gefunden und eine Verbindung zu den RTR-Sockets der Gruppe hergestellt. Sobald die Verbindung zur neuen Gruppe erfolgreich aufgebaut worden ist, werden die RTR-Sockets der vorherigen Gruppe heruntergefahren. Die Validierungsdatensätze einer RTR-Server-Gruppe sollen bei Fehlerauftritt möglichst bis zum Überschreiten des **`rtr_socket`**`.cache_timeout` Timers in der **`pfx_table`** verbleiben, um zu gewährleisten, dass IP-Präfixe als gültig oder ungültig validiert werden können.

Eine einzelne RTR-Servergruppe wird vom RTR-Manager als **`rtr_mgr_group`** abgebildet (siehe Listing 5.5). Diese enthält ein Array von RTR-Sockets, die Anzahl der Elemente des Arrays, den Verbindungsstatus sowie den Präferenzwert der Gruppe. Der RTR-Socket muss zuvor ein initialisierter Transport-Socket zugewiesen worden sein. Eine Menge von **`rtr_mgr_group`** Datenstrukturen wird in einer **`rtr_mgr_config`** Struktur gespeichert.

```
typedef struct {
    rtr_socket** sockets;
    unsigned int sockets_len;
    uint8_t preference;
    rtr_mgr_status status;
} rtr_mgr_group;

typedef struct rtr_mgr_config {
    unsigned int len;
    rtr_mgr_group* groups;
    pthread_mutex_t mutex;
} rtr_mgr_config;

typedef struct{
    rtr_socket** sockets;
    unsigned int sockets_len;
    uint8_t preference;
    rtr_mgr_status status;
} rtr_mgr_config;

int rtr_mgr_init(rtr_mgr_config* config, const unsigned int
 polling_period, const unsigned int cache_timeout,
 pfx_update_fp update_fp);
void rtr_mgr_free(rtr_mgr_config* config);

int rtr_mgr_start(rtr_mgr_config* config);
void rtr_mgr_stop(rtr_mgr_config* config);

bool rtr_mgr_group_in_sync(rtr_mgr_config* config);
int rtr_mgr_validate(rtr_mgr_config* config, const uint32_t
 asn, const ip_addr* prefix, const uint8_t mask_len,
 pfxv_state* result);
```

Listing 5.5: rtr_mgr

Zum Starten einer Instanz des RTR Connection Managers wird die Funktion `rtr_mgr_init(..)` aufgerufen. Beim Aufruf nimmt die Funktion die zu initialisierende **`rtr_mgr_config`** Struktur entgegen sowie Konfigurationsparameter, mit welchen die RTR-Sockets und eine **`pfx_table`** initialisiert werden sollen. Den RTR-Sockets wird bei der Initialisierung ein Zeiger auf eine `connection_state_fp` Funktion des RTR Connection Managers sowie ein Zeiger auf die **`rtr_mgr_config`** übergeben. Nach der Initialisierung sollen allen RTR-Sockets in der Konfiguration eine gemeinsame initialisierte **`pfx_table`** Struktur aufweisen sowie alle untergeordneten Komponenten bereit zur Verwendung sein. Die Funk-

tion `rtr_mgr_start(..)` soll die RTR-Socket-Gruppe mit dem niedrigsten Präferenzwert extrahieren und RTR-Sockets der Gruppe starten. Beim Auftreten von Fehlerzuständen in einzelnen RTR-Sockets, benachrichtigen diese den RTR Connection Manager über ihren `connection_state_fp` Callback und übergeben die verwendete **`rtr_mgr_config`**. Je nach Fehlerzustand soll der Connection-Manager die nächste notwendige Operation evaluieren und z. B. die Verbindung zu einer anderen Socket-Gruppe herstellen. Wenn mehrere RTR-Sockets einer **`rtr_mgr_config`** aktiv sind, kann es passieren dass mehrere RTR-Socket Threads parallel die Callback Funktion aufrufen. Um *Race Conditions* zu verhindern verfügt die **`rtr_mgr_config`** über ein Mutex Element (`mutex`), mit dem der gegenseitige Ausschluss in der Callback Funktion gewährleistet wird. Die Funktion `rtr_mgr_group_in_sync(..)` soll dem Aufrufer einen Wahrheitswert zurückliefern, welcher dem Wert „true“ entspricht, wenn alle RTR-Sockets einer **`rtr_mgr_config`** Struktur eine Datensynchronisierung durchgeführt haben und diese Datensätze sich noch in der **`pfx_table`** befinden. Dies ermöglicht einem Router zu evaluieren, ob eine Präfix Validierung später erneut durchgeführt werden muss, weil zur Zeit der Abfrage keine RTR-Socket-Gruppe vollständig synchronisiert war. Die Funktion `rtr_mgr_validate(..)` entspricht der Funktion `pfx_table_validate(..)` und wurde definiert, um die Bedienung zu erleichtern. Statt den Zeiger auf die **`pfx_table`** aus der tief verschachtelten **`rtr_mgr_config`** auszulesen und `pfx_table_validate(..)` zu übergeben, kann `rtr_mgr_validate(..)` direkt ein Zeiger auf die verwendete **`rtr_mgr_config`** Struktur übergeben werden.

6. Implementierung

Der folgende Abschnitt erläutert die Implementierung der Bibliothek, wobei Besonderheiten bei der Realisierung hervorgehoben werden. Im ersten Kapitel werden die Kriterien für die Wahl der Programmiersprache dargelegt. Anschließend wird die Implementierung des TCP und SSH-Protokolls als Transport-Sockets erläutert. In Kapitel 3 wird die Umsetzung des RTR-Protokolls als Zustandsautomat in der `rtr_socket` Komponente beschrieben. Nachfolgend wird die Wahl einer konkreten Datenstruktur für die `pfx_table` diskutiert und die Umsetzung der Komponente in Programmcode geschildert. Im letzten Kapitel erfolgt die Beschreibung der Realisierung der Failover-Funktionalität, die vom RTR Connection Manager implementiert wird.

6.1. Wahl der Programmiersprache

Für die Implementierung standen die Programmiersprachen C und C++ zur Auswahl, andere Sprachen wie z. B. Java oder Python wurden von vornherein ausgeschlossen da:

- Freie BGP-Daemons, wie z. B. BIRD oder Quagga, in welche die Bilbiothek integrierbar sein soll, in C implementiert sind. Die Wahl einer Sprache, die nicht direkt mit den C-Objekt-Dateien der BGP-Implementierungen verlinkt werden kann, erschwert die Integration.
- Die Bibliothek über möglichst wenig Abhängigkeiten verfügen soll, damit diese auf einer großen Anzahl von Unix-Varianten lauffähig ist und mit minimalem Aufwand installiert werden kann. Unix-Betriebssysteme werden in der Regel mit einem C-Compiler und C-Standard-Bibliothek ausgeliefert sodass keine weitere Software installiert werden müsste. Die Nutzung anderer Programmiersprache würde die Installation von zusätzlichen Entwicklungs- und Laufzeitumgebungen vorraussetzen.

In C geschriebene Funktionen haben den Vorteil dass sie ohne Modifizierung aus anderen C- und C++-Programme aufgerufen werden können. Um C++-Funktionen in C-Programmen aufzurufen, müssen hingegen spezielle Schnittstelle implementiert werden, welche Exceptions abfangen und objektorientierte Aufrufe kapseln.

Die Bibliothek soll unter einer Open-Source Lizenz veröffentlicht werden. Infolgedessen darf Programmcode in andere Anwendungen übernommen werden. C hat bei der Verwendung in Open-Source Projekten den Vorteil, dass Programmcode direkt in andere C- sowie C++-Programme kopiert und genutzt werden kann. C++-Quellcode hingegen kann nur in C++-Programme übernommen werden.

Aufgrund der genannten Vorteile wurde die Programmiersprache C für die Realisierung ausgewählt. Damit die Portierbarkeit auf andere Betriebssysteme mit wenig Aufwand möglich ist, werden die POSIX [53] Schnittstellen verwendet.

6.2. Implementierung der Transport-Sockets

In der RTR-Bibliothek wurde TCP und das SSH-Protokoll als Transport-Sockets implementiert. Die Realisierung der Transport-Sockets wird in dem nachfolgendem Abschnitt erläutert.

6.2.1. TCP-Transport-Socket

Um eine einfache Initialisierung einer `tr_socket` Struktur für die Nutzung mit dem TCP zu ermöglichen wurde die `tr_tcp_init(..)` Funktion implementiert. Bei der Initialisierung wird dem TCP-Transport-Socket eine Konfigurationsstruktur übergeben, welche den Hostnamen und die Portnummer des Rechners enthält mit dem die TCP-Verbindung aufgebaut werden soll. Nach der Initialisierung des Transport-Sockets kann durch einen Aufruf der `tr_tcp_open(..)` Funktion (siehe Listing 6.1) ein Betriebssystem-Socket angelegt und die TCP-Verbindung zur Gegenstelle aufgebaut werden. Um sowohl TCP-Verbindungen über IPv4 als auch über IPv6 zu unterstützen wurde die `getaddrinfo(..)` Funktion zum konvertieren von Hostnamen und Portnummer in Protokoll- und Platformspezifische Datenstrukturen genutzt. Hierfür werden der `getaddrinfo(..)` Funktion u. a ein Hostnamen (oder eine IP-Adresse) und einen Servicenamen (oder eine Portnummer) übergeben, welche die Funktion je nach verfügbaren Netzwerkschnittstellen in eine numerische IPv4- oder IPv6-Adresse auflöst. Als Ergebnis liefert die Funktion Datenstrukturen zurück, die den `socket(..)` und `connect(..)` Aufrufen übergeben werden können.

```
1  int tr_tcp_open(void* tr_socket){
2      int rtval = TR_ERROR;
3      tr_tcp_socket* tcp_socket = tr_socket;
4
5      struct addrinfo hints;
6      struct addrinfo* res;
7      bzero(&hints, sizeof(hints));
8      hints.ai_family = AF_UNSPEC;
9      hints.ai_socktype = SOCK_STREAM;
10     hints.ai_flags = AI_ADDRCONFIG;
11     if(getaddrinfo(tcp_socket->config->host,
       tcp_socket->config->port, &hints, &res) != 0){
12         TCP_DBG("getaddrinfo error, %s", tcp_socket,
           gai_strerror(errno));
13         goto end;
14     }
15
16     if ((tcp_socket->socket = socket(res->ai_family,
       res->ai_socktype, res->ai_protocol)) == -1){
17         TCP_DBG("Socket creation failed, %s", tcp_socket,
           strerror(errno));
18         goto end;
19     }
20
21     if (connect(tcp_socket->socket, res->ai_addr,
       res->ai_addrlen) == -1){
22         TCP_DBG("Couldn't establish TCP connection, %s",
           tcp_socket, strerror(errno));
23         goto end;
24     }
25     TCP_DBG1("Connection established", tcp_socket);
26     rtval = TR_SUCCESS;
27
28 end:
29     freeaddrinfo(res);
30     if(rtval == -1)
31         tr_tcp_close(tr_socket);
32     return rtval;
33 }
```

Listing 6.1: tr_tcp_open(..) Funktion

Die Implementierung der Timeouts der `tr_recv(..)` (siehe Listing 6.2) und `tr_send(..)` Funktionen wurde mittels der `SO_RCVTIMEO` und `SO_SNDTIMEO` Socketoptionen [53] realisiert. Durch einen Aufruf der `setsockopt(..)` Operation werden die Socketoptionen gesetzt und ermöglichen eine maximale Anzahl von Sekunden zu spezifizieren die der Socket bei Lese- bzw. Schreiboperation blockiert. Falls der Timeout abläuft bevor die Operation abgeschlossen ist, wird an den Aufrufer der Rückgabewert `TR_WOULDBLOCK` zurückgegeben. Die Unterbrechung der Empfangs- und Lesefunktionen aufgrund eines Signals wird an den Aufrufer signalisiert, indem der Wert `TR_INTR` zurückgegeben wird.

```
1  int tr_tcp_recv(const void* tr_tcp_sock, void* pdu, const
    size_t len, const time_t timeout){
2      const tr_tcp_socket* tcp_socket = tr_tcp_sock;
3      int rtval;
4      if(timeout == 0)
5          rtval = recv(tcp_socket->socket, pdu, len,
            MSG_DONTWAIT);
6      else{
7          struct timeval t = { timeout, 0 };
8          if(setsockopt(tcp_socket->socket, SOL_SOCKET,
            SO_RCVTIMEO, &t, sizeof(t)) == -1)
9              return TR_ERROR;
10         rtval = recv(tcp_socket->socket, pdu, len, 0);
11     }

13     if(rtval == -1){
14         if(errno == EAGAIN || errno == EWOULDBLOCK)
15             return TR_WOULDBLOCK;
16         if(errno == EINTR)
17             return TR_INTR;
18         return TR_ERROR;
19     }
20     if(rtval == 0)
21         return TR_ERROR;
22     return rtval;
23 }
```

Listing 6.2: tr_tcp_recv(..) Funktion

6.2.2. SSH-Transport-Socket

Für die Implementierung des SSH-Transport-Sockets wurde die *libssh* Bibliothek [14] verwendet. Diese bietet einfach zu verwendende Funktionen um eine Authentifizierung mit einem SSH-Server durchzuführen, SSH-Verbindungen aufzubauen und Daten über diese zu übertragen. und Ubuntu 11.04 enthalten. Benutzer der RTR-Bibliothek können die libssh mit dem Paketmanager vieler Betriebssysteme mit geringem Aufwand installieren. Die libssh Bibliothek, in der Version 0.4.x, ist u. a. in den Paket-Repositories der verbreiteten Unix-Varianten FreeBSD 8.2, Gentoo, Debian 6.0 Im Build-System der RTR-Bibliothek ist die libssh als optionale Abhängigkeit konfiguriert. Falls die libssh auf dem System nicht gefunden werden kann, wird die Kompilierung ohne den SSH-Transport-Socket durchgeführt.

Um eine `tr_socket` Struktur für SSH-Verbindungen zu nutzen, kann diese mithilfe der `tr_ssh_init(..)` Funktion initialisiert werden. Dieser wird eine `tr_ssh_config` Struktur übergeben, welche folgende für die Verbindungsherstellung nötigen Daten speichert:

- den Hostnamen des SSH-Servers,
- die Portnummer auf dem der SSH-Server betrieben wird,
- den Benutzername mit welchem der Login auf dem Server erfolgen soll,
- die Pfade zum öffentlichen und privaten Schlüssel für die Authentifizierung des Clients
- sowie den Pfad zum Host-Schlüssel des Servers.

Der Host-Schlüssel ist ein optionaler Parameter. Falls ein Pfad zu einem Host-Schlüssel angegeben wurde, wird mit diesem beim Verbindungsaufbau die Identität des Servers verifiziert. Die Implementierung der Timeout-Funktionalität der Empfangs- und Sendefunktionen konnte in der SSH-Implementierung nicht optimal gelöst werden. Aufgrund eines Implementierungsfehlers [13] in den libssh Versionen 0.4.x und 0.5 funktioniert die `channel_select(..)` Funktion nicht korrekt, welche zur Implementierung von Timeouts bei Lese- und Schreiboperationen genutzt hätte werden können. Statt nach Ablauf des Timeouts zurückzukehren, blockiert die `channel_select(..)` Funktion bis die übergebenen SSH-Kanäle bereit zum Datenempfang bzw. -versand sind. Als Abhilfe wurde ein Empfangstimeout in der `tr_ssh_recv(..)` Funktion mit einer Schleife emuliert (siehe Listing 6.3).

```
1  int tr_ssh_recv(const void* tr_ssh_sock, void* buf, const
    size_t buf_len, const time_t timeout){
2      if(timeout == 0)
3              return tr_ssh_recv_async(tr_ssh_sock, buf,
                buf_len);

5      time_t end_time;
6      get_monotonic_time(&end_time);
7      end_time += timeout;
8      time_t cur_time;
9      do{
10         int rtval = channel_poll(((tr_ssh_socket*)
            tr_ssh_sock)->channel, false);
11         if(rtval > 0)
12             return tr_ssh_recv_async(tr_ssh_sock, buf,
                buf_len);
13         else if(rtval == SSH_ERROR){
14             return TR_ERROR;
15         }

17         sleep(1);
18         get_monotonic_time(&cur_time);
19     } while((end_time - cur_time) >0);
20     return TR_WOULDBLOCK;;
21 }
```

Listing 6.3: tr_tcp_recv(..) Funktion

Falls der Funktion `tr_ssh_recv(..)` ein `timeout` Argument übergeben wurde, welches nicht dem Wert `Null` entspricht, wird jede Sekunde geprüft, ob neue Daten empfangen werden können. Die Funktion wird beendet, sobald Daten empfangen wurden oder der Timeout abgelaufen ist. Der Aufruf der `sleep(..)` Funktion ist nötig, da ansonsten die **`while`** Schleife 100% CPU-Auslastung erzeugen würde aufgrund des ununterbrochenen Aufrufs der `channel_poll(..)` Funktion. Durch die einsekündige Pause kann der Protokollablauf verlangsamt werden, falls nicht bei Aufruf der `tr_ssh_recv(..)` Funktion bereits Daten zum Empfang bereitstehen. Die generelle Funktionalität des Protokollablaufs wird aber nicht beeinflusst. Die Sendefunktion des SSH-Transport-Sockets ignoriert den übergebenen Timeout-Wert, da in der libssh Version 0.4.x keine Funktion verfügbar ist, um die Timeout-Funktionalität nachzubilden.

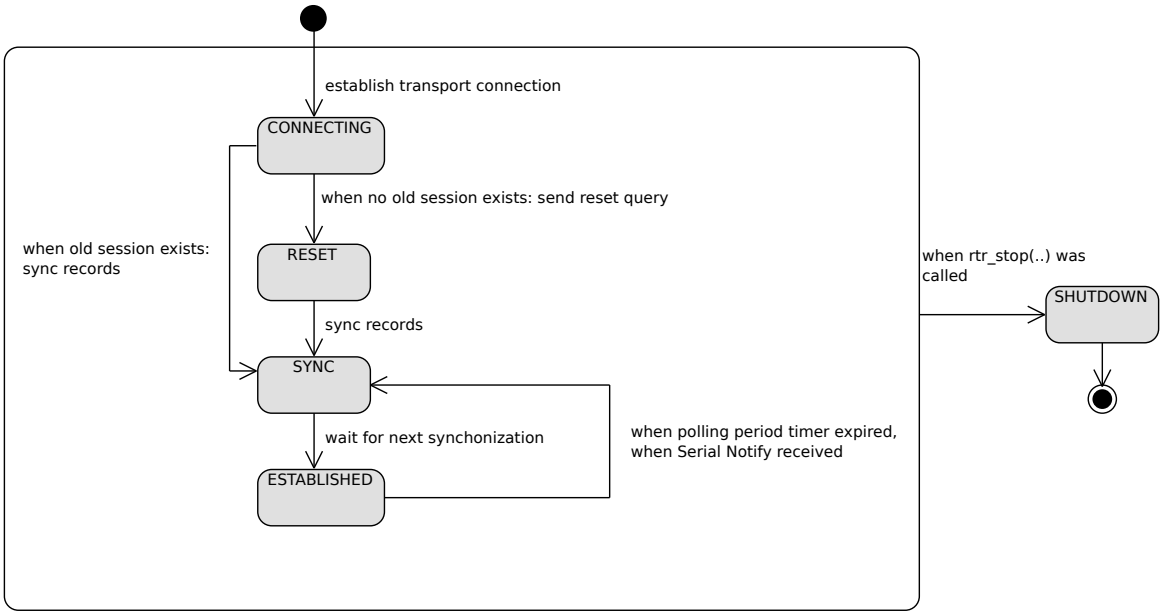

Abbildung 6.1.: RPKI-RTR Zustandsautomat

6.2.3. Funktionalitätstests

Die Funktionalität der Implementierung wurde getestet indem der TCP-Sockets mit einem Netcat [10] TCP-Server verbunden wurde. Netcat ist ein Unix-Tool, welches Daten von der Standardeingabe einliest und über den TCP-Kanal an den Kommunikationspartner übertragt. Empfange Daten werden von Netcat auf der Standardausgabe ausgegeben. Der SSH-Transport-Socket wurde mithilfe eines SSH-Server und rtr-origin [6] als SSH-Subsystem getestet. Es wurden Daten mithilfe der Transport-Sockets empfangen und versendet, die empfangenen Daten mit den versendeten verglichen, Verbindungsabbrüche simuliert indem die Serveranwendung abrupt beendet wurde sowie die Funktionalität der Empfang- und Sendetimeouts überprüft.

6.3. Implementierung des RTR-Sockets

6.3.1. Realisierung des RTR-Protokolls als Zustandsautomat

Um das RPKI-RTR Protokoll in Programmcode zu übersetzen, wurde ein Zustandsautomat (siehe Abbildung 6.1) entworfen und implementiert. Der derzeitige Zustand des Automaten wird in dem Datenelement `rtr_socket.state` gespeichert und in einer Endlos-Schleife ausgewertet. Zu Beginn befindet sich der Automat im Zustand `RTR_CONNECTING`.

Wenn bereits zu einem vorherigen Zeitpunkt der RTR-Socket eine Datensynchronisation mit dem RTR-Server durchgeführt hat, wird geprüft, ob die letzte Synchronisation länger als ein konfigurierbarer `cache_timeout` Wert in der Vergangenheit liegt. Falls dies der Fall ist, werden die veralteten Datensätze aus der **`pfx_table`** entfernt. Anschließend wird die Transportverbindung zum RTR-Server hergestellt. Falls der RTR-Socket über keine Nonce von einer vorherigen RTR-Sitzung verfügt oder zuvor veraltete Datensätze aus der **`pfx_table`** gelöscht worden sind wird in den `RTR_RESET` Zustand gewechselt. Falls der Socket über eine Nonce von einer vorherigen Sitzung verfügt wird eine *Serial Query* PDU an den Server gesendet und in den Zustand `RTR_SYNC` gewechselt. Im `RTR_RESET` Zustand wird eine *Reset Query* PDU an den Server gesendet und die Nonce aus der *Cache Response* PDU für die weitere Kommunikation gespeichert. Anschließend wechselt der Socket in den Zustand `RTR_SYNC`. Im `RTR_SYNC` Zustand werden die Validierungsdatensätze mit dem RTR-Server synchronisiert. Nach erfolgreicher Synchronisation wechselt der Socket in den Zustand `RTR_ESTABLISHED`. In diesem Zustand wartet der RTR-Client eine konfigurierbare Zeitspanne (`polling_period`) auf den Empfang einer *Serial Notify* PDU. Falls bis zum Ablauf der `polling_period` keine Daten empfangen worden sind, wird eine *Serial Query* PDU an den Server gesendet und in den Zustand `RTR_SYNC` gewechselt. Ergänzend zu den genannten Zuständen sind folgende Zustände definiert:

RTR_SHUTDOWN Die Transportverbindung zum Server ist getrennt und der Zustandsautomat wird nicht ausgeführt.

RTR_ERROR_NO_INCR_UPDATE_AVAIL Der Server hatte auf die letzte versendete *Serial Query* oder *Reset Query* PDU geantwortet, dass keine Aktualisierungen für die gesendete Seriennummer bereitgestellt werden können.

RTR_ERROR_NO_DATA_AVAIL Der Server hatte auf die letzte versendete *Serial Query* oder *Reset Query* PDU mit einer *Cache Reset* PDU geantwortet.

RTR_ERROR_FATAL Es ist ein schwerwiegender Fehler auf Client- oder Serverseite aufgetreten.

RTR_ERROR_TRANSPORT Bei der Ausführung einer Transport-Socket Operation ist ein Fehler aufgetreten.

In fast allen Zuständen kann ein Fehler auf der Transportschicht auftreten. In diesem Fall wechselt der Zustandsautomat in den Zustand `RTR_ERROR_TRANSPORT`, schließt den Transport Socket und wechselt nach Ablauf eines Fehler Timeouts in den Zustand `RTR_CONNECTING`. Die Nonce zur Identifizierung der RTR-Sitzung sowie empfangene Validierungsdatensätze werden beibehalten. Falls ein interner Fehler auftritt, z. B. kehrt eine aufgerufene Funktion mit einem Fehlerwert zurück oder es wird eine *Error* PDU vom Server empfangen, wechselt der Socket in den Zustand `RTR_ERROR_FATAL` und verhält sich wie bei Auftritt eines Transportfehlers. Falls eine *Error* PDU vom Typ „Cache has No Data

Available“ oder eine *Cache Reset* PDU empfangen worden ist wechselt der Automat in den Zustand `RTR_ERROR_NO_DATA_AVAIL` bzw. `RTR_ERROR_NO_INCR_UPDATE_AVAIL`. In diesen Zuständen wird **`rtr_socket`**`.request_nonce` auf den Wert „true“ gesetzt und nach Ablauf des Fehlertimeouts gegebenenfalls alte Validierungsdatensätze aus der **`pfx_table`** gelöscht. Anschließend wechselt der RTR-Socket in den Zustand `RTR_RESET`. Alte Validierungsdatensätze, die aus der vorherigen RTR-Sitzung stammen, werden in diesen Fällen erst bei Empfang einer *Cache Response* PDU im `RTR_SYNC` Zustand gelöscht. Dies gewährleistet, dass Datensätze aus der vorherigen Sitzung so lange wie möglich für Präfix-Validierungen erhalten bleiben.

6.3.2. Verarbeitung der RTR-Nachrichten

Die PDUs, welche zwischen RTR-Server und -Client ausgetauscht werden, sind als Datenstruktur abgebildet worden. Für die Datenelemente wurden Variablentypen gewählt, die der jeweiligen Länge des PDU-Feldes entsprechen. Eine *Error* PDU enthält 2 Felder mit einer variablen Länge zum Versenden eines Fehlertextes und der PDU, welche den Fehler aufgelöst hast. Da die Längen der Felder zum Zeitpunkt der Kompilierung unbekannt sind, können diese nicht mit Datentypen fester Länge abgebildet werden. Alle Felder, die nach dem letzten Feld mit fester Länge in der PDU angeordnet sind, werden deshalb in einem variablen Längen Array zusammengefasst (siehe Listing 6.4). Bei Empfang einer *Error* PDU wird zur Laufzeit mithilfe der Längenangaben in der PDU die Speicheradressen der restlichen Felder berechnet und ausgelesen.

```
1 typedef struct pdu_error{
2     uint8_t ver;
3     uint8_t type;
4     uint16_t error_code;
5     uint32_t len;
6     uint32_t len_enc_pdu;
7     char rest[];
8 } pdu_error;
```

Listing 6.4: PDU Datenstrukturen

Zum Empfangen von PDUs von einem Transport-Socket wurde die Funktion `rtr_recv_pdu(..)` definiert. Nach Empfang einer PDU wird geprüft, ob die empfangenen Daten einer PDU entsprechen. Der Ablauf der Funktion ist im Sequenzdiagramm in Abbildung 6.2 dargestellt. Nach Aufruf der Funktion wird zuerst nur der Header einer PDU empfangen und die Daten in die Bytereihenfolge der lokalen Rechnerarchitektur umgewandelt (Host Byte Order). Anschließend werden folgende Überprüfungen durchgeführt:

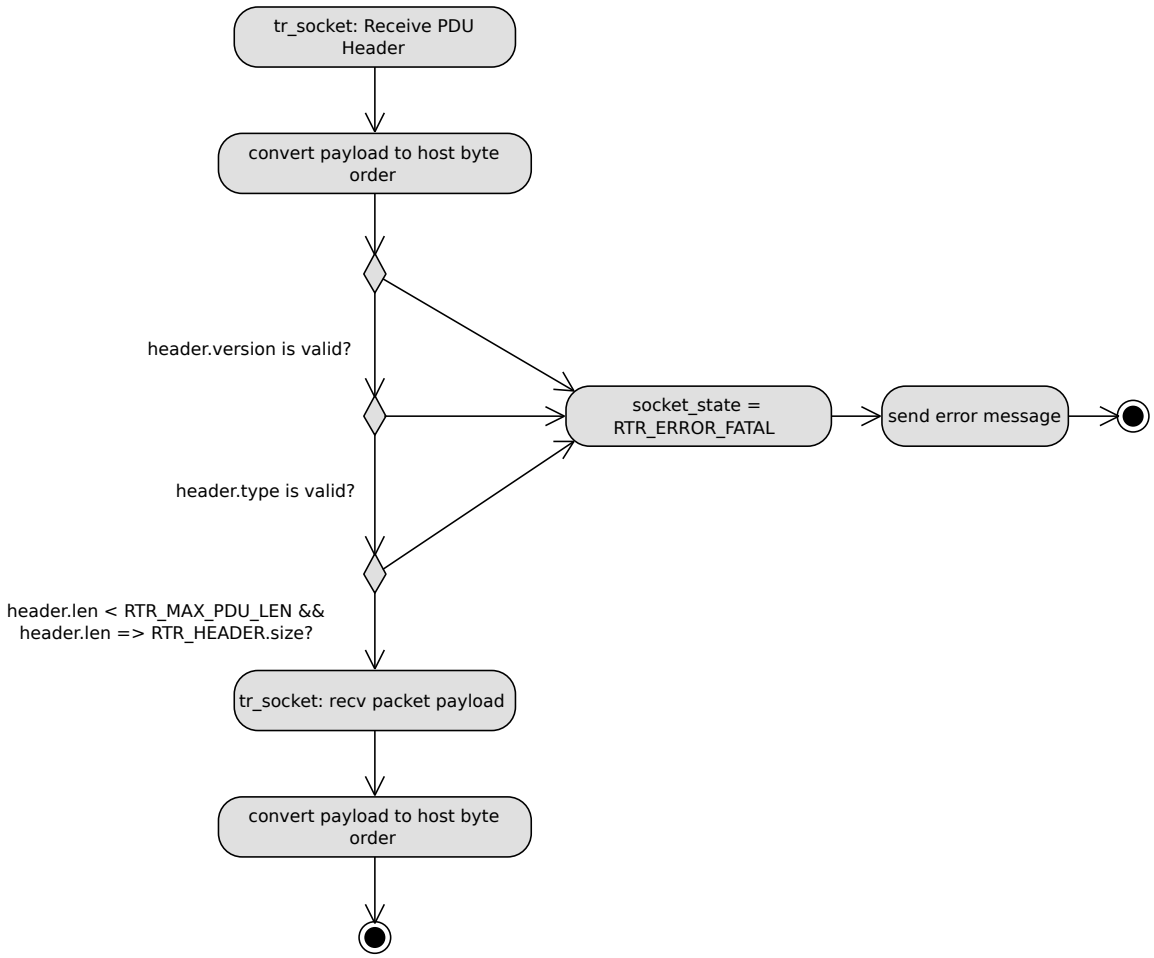

Abbildung 6.2.: Sequenzdiagramm der rtr_recv_pdu(..) Funktion

- Das `Version` Feld des Headers wird überprüft, ob es den Wert 0 beinhaltet, der die implementierte RTR-Protokoll-Version identifiziert.
- Das Feld `Type` wird überprüft, ob es einen Wert enthält welcher einem PDU-Typ entspricht (0-10).
- Das `Length` Feld der PDU wird überprüft, ob es einen Wert enthält der mindestens so groß ist wie die kleinste PDU und maximal so groß ist wie der Wert der Konstante `RTR_MAX_PDU_LEN`. `RTR_MAX_PDU_LEN` ist der Wert 3248 (Bytes) zugewiesen. Dies entspricht einer *Error* PDU, welche die größte PDU (*IPv6 Prefix* PDU) sowie einen 400 Zeichen langen Fehlertext enthält.

Falls die empfangenen Daten die Prüfung nicht bestehen, wird eine *Error* PDU an den Server gesendet, welche die empfangenen Daten und einen Fehlertext enthält. Anderenfalls wird der restliche Teil der PDU empfangen und in Host Byte Order umgewandelt.

Die Synchronisation der Validierungsdatensätze, im `RTR_SYNC` Zustand wird durch die Funktion `rtr_sync(..)` implementiert (siehe Listing A.1 im Anhang). Nach dem Aufruf der Funktion wird eine PDU vom Transport-Socket mithilfe der `rtr_recv_pdu(..)` empfangen. Falls es sich hierbei um eine *Serial Notify* PDU handelt, wird diese ignoriert und die nächste PDU vom Transport Socket eingelesen. Da der Server mehrere *Serial Notify* Nachrichten in Folge an den Client senden kann, hat wird der Programmteil in einer **`while`**-Schleife ausgeführt bis eine PDU von einem anderen Typ empfangen wurde. Falls es sich um eine *Cache Response* PDU handelt, wird der Wert des **`rtr_socket`**`.request_nonce` Datenelements überprüft. Falls dieser „true“ ist, soll die RTR-Sitzung neugestartet werden und der Nonce Wert aus der PDU wird in der **`rtr_socket`** Datenstruktur gespeichert. Anderenfalls wird die Nonce des RTR-Sockets mit der Nonce in der *Cache Response* PDU verglichen. Wenn diese nicht übereinstimmen, wird eine *Error* PDU an den Server gesendet und der Socket-Zustand in `RTR_ERROR_FATAL` geändert. Anschließend erfolgt der Transfer der Validierungsdatensätze. Es werden in einer Schleife solange neue Präfix-PDUs vom Transport-Socket gelesen bis eine *End of Data* PDU empfangen worden ist. Empfangene *Serial Notifiy* PDUs werden in der Schleife ignoriert. Der Empfang von unerwarteten PDU-Typen führt zum Versand einer *Error* PDU und in den Wechsel eines Fehlerzustandes. Falls keine Präfix PDUs übertragen wurden, sondern nur eine *End of Data* PDU ist der Datenbestand des Clients bereits aktuell und die Funktion wird beendet.

6.3.3. Funktionalitätstests

Um die RTR-Sockets auf Funktionalität zu überprüfen, wurde ein Testprogramm *rtr-client* geschrieben, welches die Verbindung zu einem RTR-Server herstellt und die empfangenen Validierungsdatensätze sowie den aktuellen Zustand des Zustandsautomaten auf der

Konsole ausgibt. Die Bibliothek wurde für die Tests als Debug-Release kompiliert, um die Ausgabe von Statusmeldungen zu aktivieren. Als RTR-Server wurde rtr-origin [6] und eine Entwicklungsversion des RPKI-Validators [16] benutzt. rtr-origin wurde für die Tests über einen TCP-Transport-Socket als xinetd Service (Konfiguration im Anhang, Listing A.7) installiert. Um auch RTR-Verbindungen über SSH zu testen, wurde RTR-Origin als SSH-Subsystem eines OpenSSH-Servers konfiguriert (Konfiguration im Anhang, Listing A.8). Die Verarbeitung von Fehlerzuständen wurde geprüft indem:

1. die Transportverbindung getrennt wurde.
2. Fehler auf der Serverseite hervorgerufen wurden, indem eine serverseitige Datensatz-Datei gelöscht wurde, was zum Versenden einer *Error* PDU führte und zum Wechsel in den `RTR_ERROR_FATAL` Zustand des RTR-Sockets.
3. ein Fehler bei einem Funktionsaufruf auf der Clientseite emuliert wurde, welcher zum Versand einer *Error*- PDU an den Server führte und zum Wechseln in den `RTR_ERROR_FATAL` Zustand.
4. der Server ohne vorhandene Datensätze gestartet wurde, dies führte zum Wechsel in den `RTR_ERROR_NO_INCR_UPDATE_AVAIl` Zustand.
5. eine ungültige Serienummer in einem *Serial Query* an den Server gesendet wurde, um den `RTR_ERROR_NO_INCR_UPDATE_AVAIL` Zustand herbeizuführen.

6.4. Implementierung der Prefix Validation Table

6.4.1. Auswahl einer Datenstruktur zur Verwaltung der Validierungsdatensätze

Für die Verwaltung der Validierungsdatensätze wurde ein modifizierter *Longest Prefix First Search Tree* (LPFST) [62] implementiert. Der LPFST wurde als Datenstruktur ausgewählt aufgrund guter Ergebnisse bei einem Performanzvergleich [62] von Datenstrukturen zum Verwalten IP-Präfixen. Ein LPFST ist ein Binärbaum, bei dem jeder Knoten mit einem IP-Präfix und einer Präfixlänge assoziiert ist. Die Knoten werden anhand der Präfixlänge im Baum sortiert. Präfixe, welche eine kleinere oder gleichgroße Länge haben wie der Elternknoten, werden je nach Wert des Präfixbits an der Position, die der Baumtiefe entspricht, als linker oder rechter Kindknoten platziert. Längere IP-Präfixe werden somit näher an der Wurzel platziert als kürzere Präfixe. Aufgrund dieser Anordnung eignet sich ein LPFST sehr gut für das Verwalten von Routingtabellen. Bei der Suche nach einem IP-Präfix im Baum wird der übereinstimmende Knoten mit der längsten Präfixlänge als erstes im Baum gefunden. Wenn der LPFST zur Speicherung von Validierungsdatensätze genutzt wird,

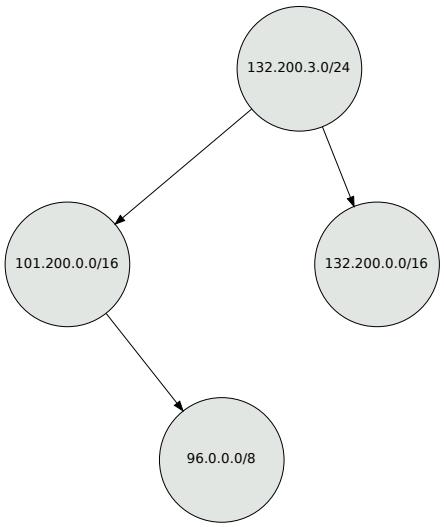

Abbildung 6.3.: Longest Prefix First Search Tree

kann die Suche nicht immer bei dem ersten Knoten, auf den das gesuchte IP-Präfix passt abgeschlossen werden. Wenn andere Datenelemente des Knotens (z. B. die ASN) mit dem gesuchten Datensatz nicht übereinstimmen muss die Suche fortgesetzt werden[1].

In Abbildung 6.3 ist ein exemplarischer LPFST dargestellt. In der Wurzel des Baumes wurde das Präfix das Präfix mit der größten Länge, `132.200.3.0/24`, gespeichert. Wenn ein Präfix an der Bitposition `Baumtiefe - 1` (das höchstwertige Bit ist an Position 0) den Wert `0` speichert, wird es als linkes Kind eingefügt, andernfalls wird es als rechtes Kind gespeichert. Im linken Kindknoten der Wurzel ist das Präfix `101.200.0.0/16` gespeichert. Der Wert des ersten Oktetts der IP-Adresse `101`, entspricht Binär der Zahlenfolge `0110 0101`, da an der Position 0 eine `0` gespeichert wird, wurde es als linkes Kind eingeordnet. Das rechte Kind der Wurzel speichert das Präfix `132.200.0.0/16`, das erste Oktett entspricht dem Binärwert `1000 0100` und an Bitposition 0 wird der Wert `1` gespeichert. Das erste Oktett des Präfixes `96.0.0.0/8`, wird im Binärformat durch den Wert `0100 0000` dargestellt. Da die Präfixlänge kleiner gleich als die der Wurzel ist und an der Bitposition 0 eine `0` gespeichert ist, wird das Präfix im linken Teilbaum der Wurzel gespeichert. Der linke Kindknoten der Wurzel speichert ebenso eine größere Präfixlänge. Das Präfix im linken Kindknoten wird deshalb nicht durch `96.0.0.0/8` ersetzt. Da an Bitposition 1, eine `1` gespeichert ist, wurde das Präfix `96.0.0.0/8` als rechter Kindknoten von `101.200.0.0/16` gespeichert.

[1] Beispielsweise könnten ROAs für die Routen *10.10.0.0/24-24 AS1* sowie *10.10.0.0/16-24 AS2* existieren und es wird eine Validierungsanfrage für *10.10.0.0/24 AS2* durchgeführt.

Ob sich für die `pfx_table` eine andere Datenstruktur besser eignet, in welcher die Präfixe aufsteigend statt absteigend sortiert werden ist unklar. Bei einer absteigende Sortierung, decken Knoten nahe der Wurzel einen größeren Adressbereich ab. Somit ist die Wahrscheinlichkeit größer, die Suche schnell abschließen zu können, wenn viele Präfixe existieren, die große Netzbereiche abdecken. In den IETF-Dokumenten [32] wird allerdings empfohlen, dass die Länge der Präfixe in den Validierungsdatensätzen möglichst genau den annoncierten BGP-Routen entsprechen. Dies führt zu einem Datenbestand, bei dem die Präfix die selbe maximale wie minimale Längenangabe haben und somit nur mit genau einer spezifischen BGP-Route übereinstimmen. Eine absteigende Sortierung wäre in diesem Fall kein Vorteil. Die Eignung der Datenstruktur ist folglich stark abhängig von den gespeicherten Validierungsdatensätzen sowie den zu bearbeitenden Suchanfragen. Stetiger Eigentümerwechsel der IP-Präfixe sowie Änderungen der annoncierten Routen haben zur Folge, dass sich die Eignungsfaktoren der Datenstrukturen fortlaufend ändern. Um die beste Datenstruktur für die `pfx_table` zu evaluieren, müssten mehrere Datenstrukturen implementiert werden und regelmäßig Benchmarks mit aktuellen ROA Datenbeständen durchgeführt werden. Die Performanzunterschiede zwischen den Datenstrukturen können auf Hardware mit geringer Leistungsfähigkeit die Ausführungszeit der Operationen stark verlangsamen. Da aber eine generell performante Datenstruktur gewählt wurde, die nicht erkennbar ungüstig ist und da auf heutiger Hardware sehr effizient gearbeitet werden kann ist nicht zu erwarten, dass signifikant bessere Performanz durch die Wahl einer anderen Datenstruktur erreicht wird.

In der LPFST Spezifikation [62] ist vorgesehen, dass die Datenstruktur mit zwei verschiedenen Knotentypen implementiert wird. Der zweite Knotentyp speichert im Gegensatz zum Ersten, die Daten für zwei verschiedene Präfixe. Wenn ein Blattknoten das gleiche Präfixe wie der Elternknoten speichert und die Präfixlänge kleiner ist als die des Elternknotens werden beide Präfixe in einem Knoten gespeichert. Da die Performanz und der benötigte Speicherplatz durch die Optimierung nur minimal verbessert wird [2], wurde auf diese Optimierung verzichtet, um die Implementierung zu erleichtern und die Lesbarkeit des Quellcodes zu erhöhen. Die minimalen Unterschiede im Speicherbedarf und in der CPU-Nutzung sollte auf aktueller Desktop- und Serverhardware keine Performanzeinbusse zur Folge haben.

In dem implementierten LPFST wird die maximale Präfix-Länge, ASN und die Socket-ID eines Validierungsdatensatzes in einem Array-Element des Knotens gespeichert. Wenn mehrere ROAs für ein Präfix existieren, wird für jedes ROA ein Element im Array des Knotens angelegt. Eine Suchoperation auf einem Array mit vielen Elementen ist grundsätzlich nicht performant, da alle Elemente des Arrays einzeln überprüft werden müssen. Validierungsdatensätze sollten aber möglichst genau den BGP-Routen entsprechen [32]

[2] In einem LPFST Benchmark [62] wurde die durschn. Knotentiefe um ca. 1,4% verringert im Gegensatz zu einem Binärbaum, welcher nur ein Präfix pro Knoten speichert.

Element	Speicherplatzbedarf
`lpfst_node.prefix`	20 Bytes
`lpfst_node.len`	1 Byte
`lpfst_node.rchild`	8 Bytes
`lpfst_node.lchild`	8 Bytes
`lpfst_node.parent`	8 Bytes
`lpfst_node.data`	8 Bytes
`node_data.len`	4 Bytes
`node_data.ary`	8 Bytes
`data_elem.asn`	4 Bytes
`data_elem.max_len`	1 Byte
`data_elem.socket_id`	8 Bytes
Gesamtgröße	**78 Bytes**

Tabelle 6.1.: Speicherbedarf eines Validierungsdatensatzes

und es gibt nur sehr wenige valide BGP-Routenk, welche von mehreren ASen annonciert werden. Folglich sollte im Normalfall im Array nur ein Element gespeichert sein oder bei der Verwendung von mehreren RTR-Servern mehrere gleichartige Elemente, welche mit denselben BGP-Routen positiv validiert werden. Der Aufwand um ein Element aus einem Array zu extrahieren, welches sich am ersten Index befindet ist konstant und kann somit vernachlässigt werden. Ein LPFST garantiert keinen vollständig balancierten Baum. Dies verringert den Aufwand von Einfüge- und Löschoperation führt aber auch dazu, dass eine Komplexität von $O \log(N)$[3] beim Suchen von Datensätzen nicht garantiert werden kann. Im schlechtesten Fall beträgt die Komplexität für Suchoperationen $O(N)$.

Um die Verteilung der Knoten der aktuellen ROAs zu überprüfen, wurde die Datenstruktur mit IPv4 Validierungsdatensätze von einem RTR-Server gefüllt, welcher sich mit den Repository Servern der RIRs synchronisierte (Stand: 28.10.2011). In der LPFST wurden 1036 Datensätze gespeichert, die maximale Baumtiefe betrug 20, die maximale Anzahl von Elementen im `node_data.ary` Array eins. Wäre der Baum optimal balanciert, hätte die Tiefe 11 betragen. Die Komplexität der Suchoperation nähert sich somit dem besten Fall eines vollständig balancierten Baumes an. Die Untersuchung zeigt zudem, dass zurzeit nicht mehr als ein ROA für ein IP-Präfix existiert und somit Knoten nur ein Element im `node_data.ary` enthalten.

Der Speicherplatzbedarf der LPFST Implementierung wurde in Tabelle 6.1 berechnet. Zeiger wurden mit einem Speicherplatzbedarf von 8 Bytes berechnet, was dem Speicherplatzbedarf auf einer 64-Bit-Architektur entspricht. Padding-Bytes die eventuell vom Compiler in die Datenstruktur eingefügt werden, wurde in der Kalkulation nicht berücksichtigt. Zur

[3] N entspricht der Anzahl der Knoten im LPFST.

Speicherung von 336.072 [24] Validierungsdatensätzen, welches die Anzahl der Routing-Präfixe in einer BGP-Routing-Tabelle am 22.09.2011 entsprach werden folglich 25,00 MB Arbeitsspeicher benötigt.

6.4.2. Realisierung der Prefix Validation Table

Ein Knoten im LPFST wird dargestellt durch die in Listing 6.5 abgebildete Datenstruktur. Das Element `len` speichert die minimale Länge des IP-Präfixes, für welches der Validierungsdatensatz gültig ist. Die Kinder des Knotens werden in den Elementen `rchild` und `lchild` gespeichert. Um Operationen auf dem Baum zu erleichtern, wird zusätzlich ein Verweis auf den Elternknoten im Element `parent` gespeichert. Dies wäre nicht zwingend notwendig gewesen, erleichtert aber die Implementierung von Operationen auf dem Baum. Die restlichen Attribute einer **`pfx_record`** Datenstruktur wie die ASN und die maximale Präfix-Länge werden in einem `node_date` Array (siehe Listing 6.6), auf den der **`lpfst_node`**`.data` Zeiger verweist, gespeichert.

```
typedef struct lpfst_node_t {
    ip_addr prefix;
    uint8_t len;
    struct lpfst_node_t* rchild;
    struct lpfst_node_t* lchild;
    struct lpfst_node_t* parent;
    void* data;
} lpfst_node;
```

Listing 6.5: lpfst_node Datenstruktur

```
typedef struct data_elem_t{
    uint32_t asn;
    uint8_t max_len;
    uintptr_t socket_id;
} data_elem;

typedef struct node_data_t{
    unsigned int len;
    data_elem* ary;
} node_data;
```

Listing 6.6: node_data Array

IP-Adressen werden als `ip_addr` Struktur (siehe Listing 6.7) im Baum hinterlegt.

```
typedef struct {
    uint64_t addr[2];
} ipv6_addr;

typedef struct {
    uint32_t addr;
 ipv4_addr;

typedef enum ip_version {
    IPV4,
    IPV6
} ip_version;

typedef struct {
    ip_version ver;
    union{
        ipv4_addr addr4;
        ipv6_addr addr6;
    } u;
} ip_addr;
```

Listing 6.7: ip_addr Struktur

Eine `ip_addr` speichert entweder eine IPv4- oder eine IPv6-Adresse in einer Union sowie einen Aufzählungswert zur Beschreibung der IP-Version der gespeicherten Adresse.

Die implementierten Such-, Einfüge und Löschoperationen auf dem LPFST, entsprechen zum größten Teil der Spezifikation [62]. Es wurden lediglich kleine Modifikationen vorgenommen, um die Operationen auf einem LPFST mit nur einem Knotentyp durchzuführen. Die `pfx_table` Operationen verwenden POSIX Schreib-Lese-Sperren (RW-Locks), um parallele Zugriffe ohne Seiteneffekte durchführen zu können. RW-Locks ermöglichen den Zugriff von mehreren Threads parallel, wenn nur Leseoperationen auf der Datenstruktur ausgeführt werden. Schreiboperation auf der Datenstruktur können hingegen nur exklusiv erfolgen. Das heißt es kann nur ein Thread zur Zeit eine Schreiboperation ausführen. Alle anderen Threads, welche zur gleichen Zeit Lese- oder Schreiboperationen ausführen sollen, müssen warten bis die vorherige Schreiboperation abgeschlossen ist. Zu Beginn der `pfx_table` Funktionen wird eine Sperre beantragt und sobald alle Schreib- bzw. Leseoperationen abgeschlossen wurden wieder freigegeben. In der `pfx_table` werden jeweils zwei seperate LPFSTs zum Speichern von IPv4- und IPv6-Validierungsdatensätze genutzt. Operationen auf den separaten Bäumen können schneller als auf einer einzigen LPFST ausgeführt werden, da die Baumtiefen geringer sind. Zur Einordnung der Knoten im LPFST

wurden Bitoperationen implementiert, welche zwei `ip_addr` Strukturen sowie bestimmte Bits einer gespeicherten IP-Addresse mit einem übergebenen Wert vergleichen.

Die Realisierung der `pfx_table_validate(..)` Funktion zur Validierung von IP-Präfixen ist in Listing 6.8 dargestellt.

```
1  int pfx_table_validate(struct pfx_table* pfx_table, const
   uint32_t asn, const ip_addr *prefix, const uint8_t mask_len,
   pfxv_state* result){
2      pthread_rwlock_rdlock(&(pfx_table->lock));
3      lpfst_node* root = pfx_table_get_root(pfx_table,
       prefix->ver);
4      if(root == NULL){
5          pthread_rwlock_unlock(&pfx_table->lock);
6          *result = BGP_PFXV_STATE_NOT_FOUND;
7          return PFX_SUCCESS;
8      }
10     unsigned int lvl = 0;
11     lpfst_node* node = lpfst_lookup(root, prefix, mask_len,
       &lvl);
12     if(node == NULL){
13         pthread_rwlock_unlock(&pfx_table->lock);
14         *result = BGP_PFXV_STATE_NOT_FOUND;
15         return PFX_SUCCESS;
16     }
18     while(!pfx_table_elem_matches(node->data, asn, mask_len)){
19         if(ip_addr_is_zero(ip_addr_get_bits(prefix, lvl, lvl)))
20             node = node->lchild;
21         else
22             node = node->rchild;
23         lvl++;
24         node = lpfst_lookup(node, prefix, mask_len, &lvl);
25         if(node == NULL){
26             pthread_rwlock_unlock(&pfx_table->lock);
27             *result = BGP_PFXV_STATE_INVALID;
28             return PFX_SUCCESS;
29         }
30     }
31 }
```

Listing 6.8: pfx_table_validate Funktion

Zu Beginn der Funktion wird eine Lesesperre beantragt, welche vor jeder **return** Anweisung wieder freigegeben wird. Der Wurzelknoten des LPFST wird je nach IP-Version des gesuchten Datensatzes von der Funktion `pfx_table_get_root(..)` evaluiert und in der `root` Variable gespeichert. Wenn der LPFST über keinen Datensatz verfügt, ist der `root` Variable der Wert `NULL` zugewiesen und die Funktion wird erfolgreich beendet mit dem Validierungsstatus `BGP_PFXV_STATE_NOT_FOUND`. Anderenfalls wird eine Suchoperation auf dem LPFST ausgeführt. Falls kein Knoten mit dem gesuchten Präfix gefunden werden konnte wird die Funktion ebenso mit dem Validierungsstatus `BGP_PFXV_STATE_NOT_FOUND` erfolgreich beendet. Ansonsten wird das **`node_data.ary`** des Knoten nach einem Eintrag mit der gesuchten ASN und einer passenden maximalen Präfixlänge durchsucht. Falls kein passender Eintrag gefunden werden konnte, wird die Suche im Baum bei den Kindknoten, des gefundenen Knoten fortgeführt. Wenn die `lpfst_lookup(..)` den Wert `NULL` zurückgibt, konnte kein weiterer Knoten mit einem passenden Präfix im Baum gefunden werden. Da bereits vorher mindestens ein Knoten gefunden wurde, auf welchem das Präfix zutraf aber die ASN oder die maximale Präfixlänge nicht übereinstimmte, wird die Funktion erfolgreich mit dem Validierungsstatus `BGP_PFXV_STATE_INVALID` beendet. Falls die Funktion `pfx_table_elem_matches(..)`, welche im Schleifenkopf ausgeführt wird `true` zurückgibt, wurde ein Baumknoten gefunden, welcher mit dem gesuchten Datensatz übereinstimmt und die Funktion wird mit dem Validierungsstatus `BGP_PFXV_STATE_VALID` beendet.

6.4.3. Funktionalitätstests

Um die Funktionalität der implementierten LPFST Datenstruktur zu testen, wurde ein Unit-Test geschrieben (siehe Anhang, Listing A.4). Der Test überprüft die Korrektheit der implementierten Bit-Operationen, welche zum Vergleichen von IP-Präfixen im LPFST benutzt werden. Anschließend werden mehrere Knoten im LPFST eingefügt, die Position der Knoten im Baum überprüft und das Ergebnis der Suchoperation `lpft_lookup(..)` validiert. Nach dem Löschen von Knoten aus dem Baum wurde geprüft, ob die Zusammenführung der beiden entstandenen Teilbäume korrekt durchgeführt wurde.

Die Funktionsfähigkeit der **`pfx_table`** wurde ebenso mit einem Unit-Test überprüft (siehe Anhang, Listing A.5). Es wurden über 196.000 `pfx_records` in die Datenstruktur eingefügt, validiert und gelöscht. Zudem wurden gezielt Datensätze eingefügt und Validierungsanfragen gestellt, um die Ergebnisse der `pfx_table_validate(..)` Funktion zu überprüfen. Um die **`pfx_table`** auf Deadlocks zu überprüfen, wurden 15 Threads gestartet, welche parallel, mehrere Stunden in randomisierten Abständen, Operationen auf derselben **`pfx_table`** ausgeführt haben (siehe Anhang, Listing A.6). Ein solcher Test kann nicht mit Sicherheit beweisen, dass keine Deadlocks in der Implementierung auftreten. Wenn aber über eine längere Zeitspanne parallele Operationen erfolgreich ausgeführt worden sind, ist die Wahrscheinlichkeit gering, dass Fehler in der Implementierung der Sperren vorhanden sind.

6.5. Implementierung des RTR Connection Managers

Eine Instanz des RTR Connection Managers wird durch einen Aufruf der Funktion `rtr_mgr_init(..)` initialisiert. Bei der Initialisierung werden die `rtr_mgr_group` Elemente der Konfiguration aufsteigend anhand des Präferenzwertes sortiert (siehe Listing 6.9).

```
1  int rtr_mgr_init(rtr_mgr_config* config, const unsigned int
    polling_period, const unsigned int cache_timeout,
    pfx_update_fp update_fp){
2      if(pthread_mutex_init(&(config->mutex), NULL) != 0)
3          MGR_DBG1("Mutex initialization failed");
4
5      qsort(config->groups, config->len, sizeof(rtr_mgr_group),
        &rtr_mgr_config_cmp);
6      for(unsigned int i = 0; i < config->len; i++){
7          if(config->groups[i].sockets_len == 0){
8              MGR_DBG1("Error Socket group contains an empty
                sockets array");
9              return RTR_ERROR;
10         }
11         if(i > 0 && config->groups[i-1].preference ==
            config->groups[i].preference){
12             MGR_DBG1("Error Socket group contains 2 members
                with the same preference value");
13             return RTR_ERROR;
14         }
15     }
16     pfx_table* pfxt = malloc(sizeof(pfx_table));
17     if(pfxt == NULL) return RTR_ERROR;
18     pfx_table_init(pfxt, update_fp);
19
20     for(unsigned int i = 0; i < config->len; i++){
21         config->groups[i].status = RTR_MGR_CLOSED;
22         for(unsigned int j = 0; j <
            config->groups[i].sockets_len; j++){
23             rtr_init(config->groups[i].sockets[j], NULL, pfxt,
                polling_period, cache_timeout, rtr_mgr_cb,
                config);
24         }
25     }
26     return RTR_SUCCESS;
27 }
```

Listing 6.9: rtr_mgr_init(..) Funktion

Durch die Sortierung werden spätere Operationen erleichtert, um z. B. bei Auftritt eines Fehlers die Gruppe mit dem nächsthöheren Präferenzwert in der Konfiguration zu finden, muss nicht mehr das komplette Array durchsucht werden, sondern die Gruppe ist an dem nächsthöheren Index Wert zu finden. Anschließend wird Speicher für eine **pfx_table** Struktur alloziert und diese initialisiert. Bei der Initialisierung der **pfx_table** wird der pfx_table_init(..) Funktion der **pfx_update_fp** Parameter übergeben. Danach erfolgt eine Initialisierung der RTR-Sockets, die Konfigurationsparameter der rtr_mgr_init(..) Funktion werden an die RTR-Sockets weitergereicht. Das Reagieren auf Zustandsänderungen von einzelnen RTR-Sockets wird durch die rtr_mgr_cb(..) Funktion realisiert, welche als **rtr_connection_state_fp**(..) Callback in den einzelnen RTR-Sockets installiert wird. Bei der Ausführung der rtr_mgr_start(..) Funktion werden die RTR-Sockets der **rtr_mgr_group**, welche sich im ersten Element im **rtr_mgr_config**.groups Array befinden gestartet.

Der aktuelle Status (**rtr_mgr_group**.status) einer RTR-Manager-Gruppe ist abhängig vom vorherigen Status sowie den Zuständen der RTR-Sockets der Gruppe. **rtr_mgr_config**.status kann folgende Werte annehmen:

RTR_MGR_CLOSED Die RTR-Sockets der Gruppe befinden sich im Zustand RTR_SHUTDOWN.

RTR_MGR_CONNECTING Der RTR Connection Manager hat die rtr_start(..) Funktion der RTR-Sockets ausgeführt, um die Verbindung mit den RTR-Servern zu starten.

RTR_MGR_ESTABLISHED Alle RTR-Sockets in der **rtr_mgr_config** Struktur haben erfolgreich eine Transportverbindung zum jeweiligen RTR-Server hergestellt und bereits mindestens eine Datensatzsynchronisierung durchgeführt. Die Zustandsautomaten der RTR-Sockets müssen sich im Zustand RTR_ESTABLISHED, RTR_RESET oder RTR_SYNC befinden.

RTR_MGR_ERROR Mindestens ein RTR-Socket der Gruppe befindet sich in einem Fehlerzustand.

Wenn der Zustandsautomat eines RTR-Sockets in einen neuen Zustand wechselt, wird der RTR Connection Manager über die Änderung informiert und kann je nach Zustand reagieren. Wenn ein RTR-Socket in den Zustand RTR_NO_INCR_UPDATE_AVAIL wechselt, wird falls eine **rtr_mgr_config** mit einem kleineren Präferenzwert existiert, die Verbindung zu den RTR-Sockets der Gruppe hergestellt. Falls keine Konfiguration mit einem niedrigeren Präferenzwert existiert, wird keine Maßnahme vom RTR Connection Manager durchgeführt.

Die RTR-Sockets behandeln selbständig den Auftritt von Fehlern. Nach einem Fehler Timeout wird die betroffene RTR-Verbindung durch eine *Cache Reset* PDU zurückgesetzt. Wenn ein RTR-Socket in den Zustand RTR_NO_INCR_UPDATE_AVAIL wechselt, wird der Gruppenstatus auf RTR_MGR_ERROR geändert. Anschließend wird im **rtr_mgr_config** Array eine Gruppe mit einem niedrigeren Präferenzwert gesucht, welche sich im Status RTR_MGR_CLOSED befindet. Falls eine andere Gruppe gefunden wurde, werden die RTR-Sockets der Konfiguration gestartet und der Status auf RTR_MGR_CONNECTING geändert. Falls keine solche Konfiguration gefunden wurde, werden keine weiteren Operationen ausgeführt (siehe Listing 6.10).

```
1     else if(state == RTR_ERROR_NO_INCR_UPDATE_AVAIL){
2         config[ind].status = RTR_MGR_ERROR;
3         int next_config = ind - 1;
4         while(next_config >= 0 && config[next_config].status
           != RTR_MGR_CLOSED){
5             next_config--;
6         }
7         if(next_config >= 0)
8             rtr_mgr_start_sockets(&(config[next_config]));
9     }
```

Listing 6.10: Behandlung des RTR_ERROR_NO_INCR_UPDATE_AVAIL Fehlerzustands im RTR Connection Manager

Falls ein RTR-Socket den Fehlerzustand RTR_ERROR_FATAL, RTR_ERROR_TRANSPORT oder RTR_ERROR_NO_DATA_AVAIL meldet wird der Gruppenstatus in RTR_MGR_ERROR geändert. Anschließend wird eine Gruppe mit dem nächsthöheren Präferenzwert gesucht. Falls keine Gruppe mit einem höheren Präferenzwert existiert, wird die Gruppe mit dem nächstniedrigeren Präferenzwert gesucht. Die gefundenene **rtr_mgr_group** wechselt in den Zustand RTR_MGR_CONNECTING und startet die Zustandsautomaten der RTR-Sockets.

Nach dem Auftritt eines Fehlerzustandes in einem RTR-Socket können folgende Situationen auftreten:

- Eine Gruppe befindet sich im Status RTR_MGR_ERROR, andere Gruppen sind nicht vorhanden oder befinden sich im Status RTR_MGR_CLOSED.

  ⇒ Die Gruppe wechselt in den Status RTR_MGR_ESTABLISHED, sobald alle Sockets die Anforderungen des Status erfüllen.

- Mindestens eine Gruppe befindet sich im Status RTR_MGR_ERROR, eine oder mehrere Gruppen befinden sich im Status RTR_MGR_CONNECTING.

  ⇒ Sobald eine Gruppe die Anforderungen des RTR_MGR_ESTABLISHED Status erfüllt, wird der Status der Konfiguration geändert. Der RTR-Zustandsautomat aller

anderen Konfigurationen wird beendet und die beendeten Gruppen bekommen den Status `RTR_MGR_CLOSED` zugewiesen.

Da der RTR Connection Manager auf allen anderen Komponenten aufbaut, wird bei Funktionalitätstest die Funktionsfähigkeit der kompletten Bibliothek überprüft. Für die Tests wurden drei **`rtr_mgr_group`** Gruppen angelegt, denen jeweils mehrere RTR-Sockets zugewiesen waren. Es wurden RTR-Server benutzt die sowohl TCP- als auch das SSH-Protokoll zur Kommunikation nutzten. Als Server-Anwendung wurde rtr-origin [6] und RPKI-Validator [16], eingesetzt. Der RTR Connection Manager wurde gestartet und über mehrere Tage betrieben. Während dem Betrieb wurden Fehlerzustände mit verbundenen RTR-Servern hervorgerufen, indem z. B. die Serveranwendung abrupt beendet und RTR-Server ohne Datenbestände betrieben wurden.

7. Evaluation

Die Bibliothek wurde evaluiert, indem aktuelle BGP-Annoncierungen vom Live-Data Dienst des BGP Monitoring Systems (BGPmon) [2] empfangen und validiert wurden. Während des Messzeitraums wurde die CPU-Auslastung und Speichernutzung des Testprogramms minütlich protokolliert. Für jede durchgeführte Validierung wurde zudem das Präfix, die ASN, die Präfixlängenangaben sowie der Validierungsstatus aufgezeichnet.

Das Testzenario, in dem die Evaluierung durchgeführt wurde, ist in Abbildung 7.1 skizziert. Der Rechner *livebgp.netsec.colostate.edu* gehört zum BGPmon und stellt BGP-Annoncierungen in einem XML-Format über einen TCP-Stream zur Verfügung. Auf dem Benchmark-Rechner empfängt ein Python-Programm (siehe Anhang, Listing A.2) die XML-Daten von *livebgp.netsec.colostate.edu* und extrahiert *BGP-UPDATE*-Nachrichten, die Routen annoncieren. Für jede annoncierte Route gibt das Python-Programm das IP-Präfix, die Präfixlänge sowie das Herkunfts-AS aus. Ein in C geschriebenes Benchmark Programm (siehe Anhang, Listing A.3) baut mithilfe der RTR Bibliothek eine Verbindung zu einem rtr-origin [6] Server auf. Als Ersatz RTR-Server wurde eine RPKI-Validator Instanz verwendet. Nach der ersten erfolgreichen Datensynchronisation mit dem RTR-Server, liest das Benchmark Programm die vom Python-Script extrahierten Routen-Annoncierungen über eine Unix-Pipe ein und validiert jede Annoncierung mithilfe der `rtr_mgr_validate(..)` Funktion. Für jede Validierung wurde das IP-Präfix, die Präfix-Länge, die ASN sowie das Ergebnis der Validierung aufgezeichnet. Ein separater Thread, der vom Benchmark Programm gestartet wurde, protokollierte, wie viele Routen mit den Status *Valid*, *Invalid* oder *Not found* ausgezeichnet wurden, die relative CPU-Last sowie die Menge des durch das Benchmark Programm belegten Arbeitsspeichers. Das Empfangen und Verarbeiten der XML-Daten vom BGPmon Live-Data Dienst wurde in einem separaten Python-Programm durchgeführt, um die Systemlast, welche durch das Parsen der XML-Nachrichten entsteht, von der Systemlast der Bibliothek und der Validierungsoperationen zu trennen. Die Programmiersprache Python wurde gewählt, weil es sich hierbei um eine Hochsprache handelt, mit der das Extrahieren der BGP-Annoncierungen mit minimalem Implementierungsaufwand möglich ist.

Die verwendete RTRlib, sowie das Benchmark Programm wurden als Release-Build kompiliert. Das heißt, dem gcc Compiler wurden Parameter übergeben, um Codeoptimierungen

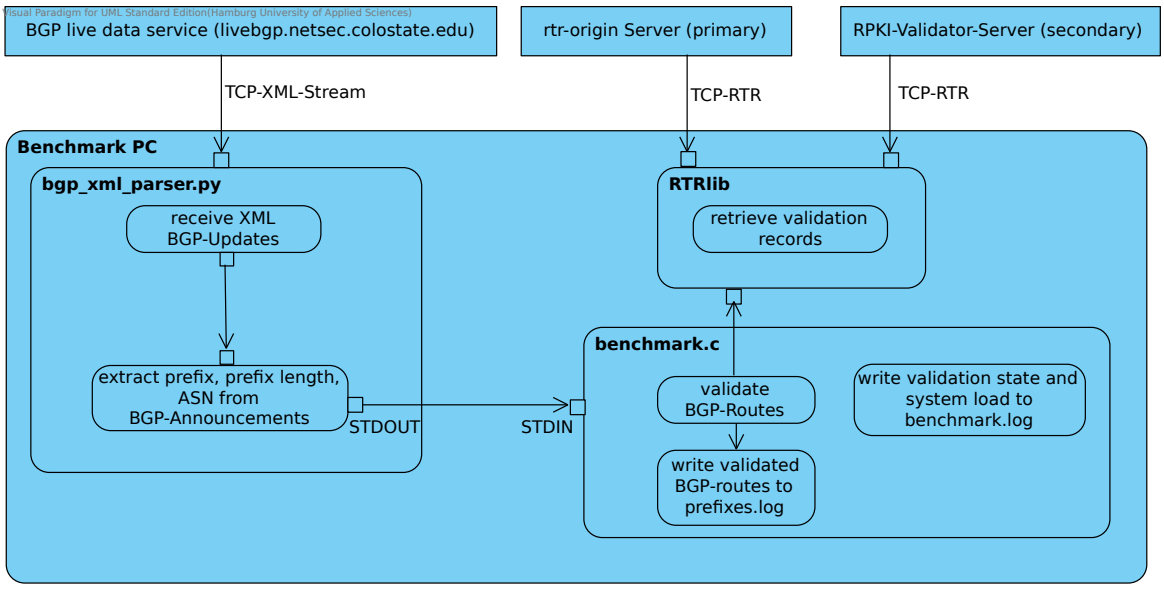

Abbildung 7.1.: Testaufbau

bei der Übersetzung durchzuführen. Der Benchmark-Rechner war mit 2 GB Arbeitsspeicher und einer AMD Athlon 64 X2 CPU ausgestattet. Als Betriebssystem wurde Debian 6.0 eingesetzt. Beim durchgeführten Benchmark wurden zu Beginn 1317 Datensätze vom RTR-Server empfangen und in der `pfx_table` gespeichert. Im Diagramm 7.2 wird die Anzahl der Validierungen der verursachten CPU-Last gegenübergestellt. Im oberen Graphen sind die Anzahl der Validierungen pro Minute dargestellt. Die x-Achse beschreibt die Anzahl der Validierungen. Die y-Achse stellt bei beiden Graphen den Zeitpunkt der Validierung da. Auf dem unteren Graphen ist die CPU-Last des Benchmarks Programms aufzeichnet. Beide Graphen wurden anhand der y-Achse ausgerichtet, sodass die CPU-Last auf dem unteren Diagramm bei den dargestellten Validierungen auf dem oberen Diagramm aufgetreten ist.

Auffällig ist die sehr geringe CPU-Last, welche weniger als 0,2% der verfügbaren Rechenleistung entsprach. Aufgrund der gewählten LPFST Datenstruktur sind Validierungsoperationen in der `pfx_table` mit sehr wenig Vergleichs- und Dereferenzierungsoptionen durchführbar. Der periodische Abgleich der Validierungsdatensätze benötigt ebenfalls wenig Rechenleistung, in den meisten Fällen werden keine neuen Datensätze übertragen und es müssen nur zwei RTR-Nachrichten mit dem Server ausgetauscht und verarbeitet werden. Die Anzahl der Validierungen korreliert zudem sehr stark mit der CPU-Last. Selbst wenn innerhalb eines Messzeitraumes tiefer als durchschnittlich in den Suchbaum abgestiegen werden muss, werden bei einer beobachteten Baumtiefe von 20 nur wenig mehr Operationen ausgeführt als wenn das Ergebnis nahe der Wurzel gefunden werden konnte. Während

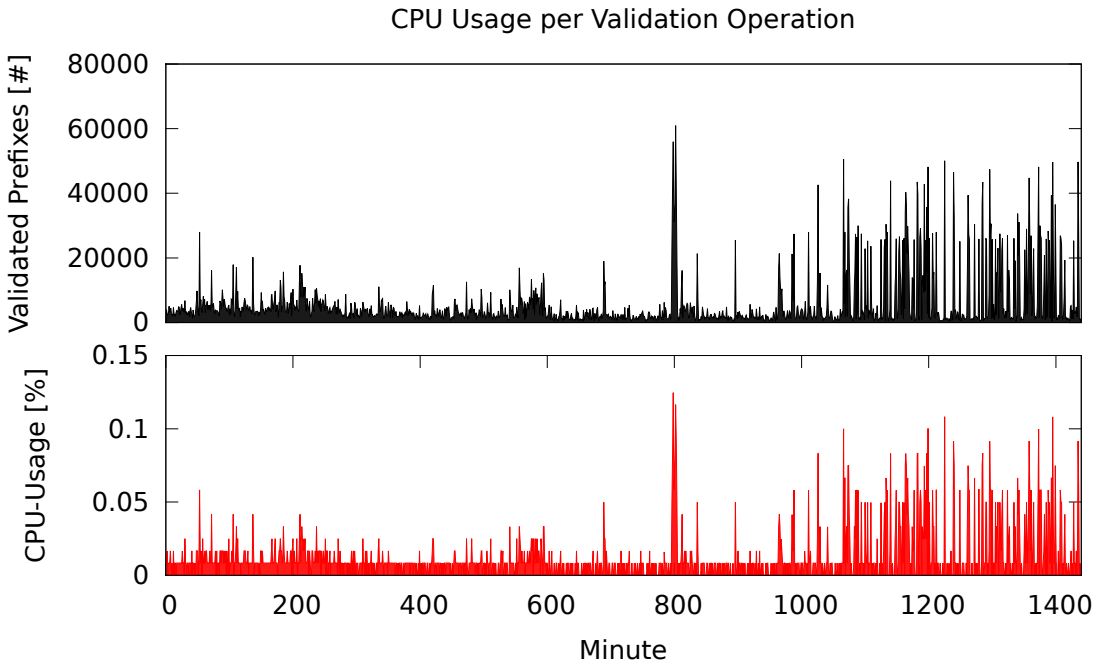

Abbildung 7.2.: Erzeugte CPU-Last durch Validierungsoperationen (1336 Datensätze)

der Messung wurde konstant, die initiale Datensynchronisierung ausgeschlossen, 976 KB Arbeitsspeicher belegt.

Aus den aufgezeichneten Validierungsstatus der einzelnen BGP-Routen wurden doppelte Einträge entfernt, um zu ermitteln wie viele Routen im Messzeitraum mit der RPKI als *Valid*, *Not Found* oder *Invalid* ausgezeichnet werden konnten. Die Häufigkeit der einzelnen Validierungsergebnisse ist im Diagramm 7.3 dargestellt. Auf der y-Achse ist die Anzahl der validierten Präfixe logarithmisch skaliert dargestellt. Auf der y-Achse sind die einzelnen Validierungsstatus aufgetragen. 2264 annoncierte BGP-Routen wurden im Messzeitraum als *Valid* validiert. In der `pfx_table` waren allerdings nur Datensätze für 1336 Präfixe vorhanden. Dies ist damit zu erklären, dass viele Validierungsdatensätze eine variable Längenangabe besaßen. Ein Validierungsdatensatz ist somit gültig für einen großen Netzbereich, für den viele unterschiedliche verschiedene Routen annonciert werden und als gültig ausgezeichnet werden können.

1630 annoncierte BGP-Routen wurden als ungültig validiert. Bei diesen Routen kann es sich um beabsichtigte Präfix-Hijacks handeln. Eine *Invalid* Auszeichnung kann aber auch durch eine Fehlkonfiguration eines Routers oder durch veraltete Informationen in einem ROA-Objekt zustande gekommen sein. Die Mehrheit, der annoncierten Präfixe wurde mit

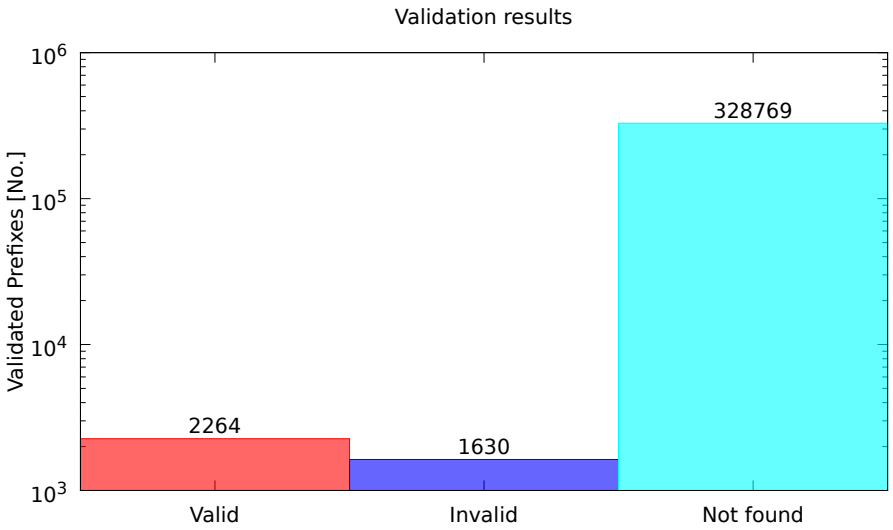

Abbildung 7.3.: Validierungsergebnisse

dem Validierungsstatus *Not Found* klassifiziert, dies entspricht den Erwartungen, da nur Validierungsdaten für einen Bruchteil der annoncierten IP-Präfixe existierten.

Im ersten Benchmark war die verursache CPU-Last durch die Bibliothek sehr gering, da nur wenige Datensätze in der Prefix Validation Table verwaltet wurden. Um auch die Leistung der Bibliothek zu untersuchen, wenn eine große Anzahl von Validierungsdatensätze gespeichert wird, wurden die empfangenen BGP-Routen aus dem vorherigen Test sowie künstlich erzeugte Datensätze zur `pfx_table` hinzugefügt. Ingesamt wurden 2.093.971 Validierungsdatensätze in die `pfx_table` hinzugefügt. Mithilfe des Benchmark Progammes wurden erneut 24 Stunden BGP-Annoncierungen vom BGPmon Live-Data Dienst validiert. Die Ergebnisse sind im Diagramm 7.4 dargestellt. Im Vergleich zum ersten Benchmark sind im Meßzeitraum deutlich mehr annoncierte Präfixe empfangen und verarbeitet worden. Obwohl 156.734 mal mehr Datensätze in der Prefix Validation Table gespeichert worden sind, ist die CPU-Last vergleichbar mit dem ersten Versuch. Die verursachte Last war nach wie vor sehr gering. In keinem Testzeitraum wurde mehr als 0,15% der vorhandenen CPU-Leistung benötigt. In Kapitel 6.4.1 wurde eine benötigte Speichermenge von 78 Bytes pro Validierungsdatensatz errechnet (siehe Tabelle 6.1). Zur Haltung von 2.093.971 Datensätze würden folglich 155,76 MB benötigt. Das Benchmark Programm benötigte 158,45 MB Speicher, welches sich der errechneten Speichermenge, die nur die Validierungsdatensätze einbezieht, annähert.

Die durchgeführten Benchmarks lieferten sehr gute Performanz-Ergebnisse. Mehrere Zehntausend Validierungen pro Minute verursachten nur eine geringe CPU-Last auf einer älteren AMD Dual-Core CPU. Die benötigte Speichermenge des Testprogrammes entsprach den

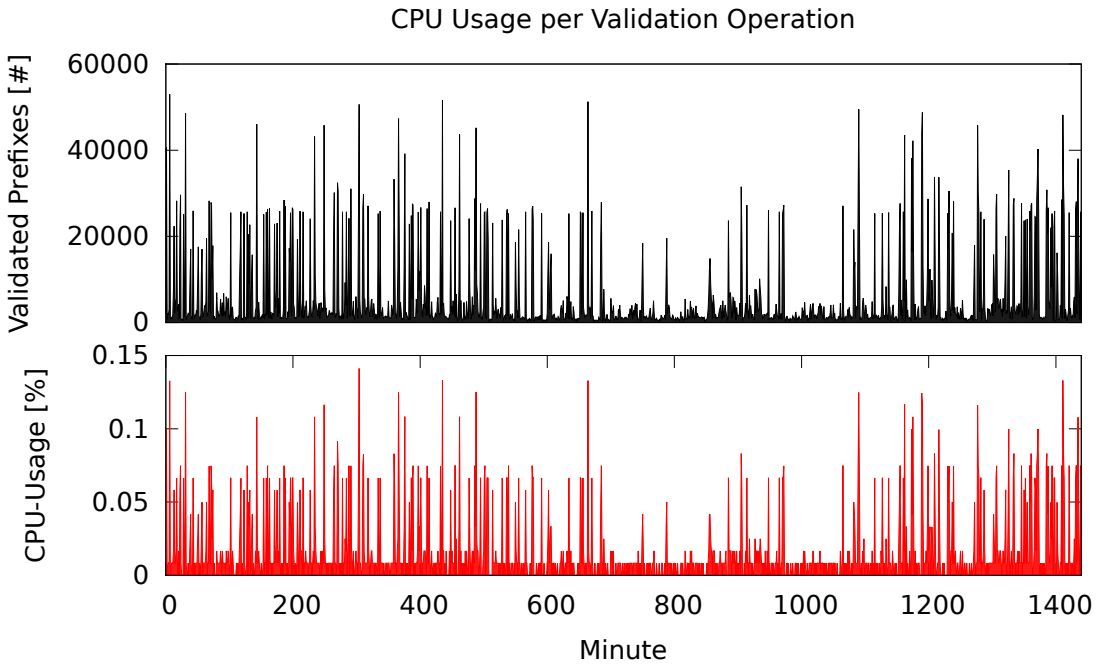

Abbildung 7.4.: Erzeugte CPU-Last durch Validierungsoperationen (2.093.971 Datensätze)

Berechnungen aus dem Implementierungskapitel an. Zur Verwaltung von 1317 Validierungsdatensätzen wurden nur 976 KB Speicher benötigt. Demnach kann die Bibliothek auf älterer Hardware mir geringer Leistungsfähigkeit eingesetzt werden. Während des Meßzeitraumes konnte nur ein kleiner Anteil der Routen als positiv validiert werden, da nur für wenige Routen ROA Objekte existieren. Der Großteil der Routen wurde mit dem Status *Not Found* klassifiziert.

8. Zusammenfassung und Ausblick

Das Ziel der Arbeit war die Implementierung einer Bibliothek, für die Client-seite des RPKI-RTR-Protokolls. Es wurden die benötigten Grundlagen zum Verständniss des Themas vermittelt sowie eine Analyse vorhandener Sicherheitsprobleme und Lösungen im BGP vorgenommen. Für die Bibliothek wurde eine komponentenzentrierte Architektur gewählt und in der Programmiersprache C implementiert. Jede Komponente ist zuständig für einen abgeschlossenen Aufgabenbereich und kommuniziert über definierte Schnittstellen mit anderen Komponenten. Die ausgewählte Architektur ermöglicht es einzelne Komponenten später auszutauschen und erleichtert die Erweiterung und Modifizierung der Bibliothek.

Verschiedene Transportprotokolle können von der Bibliothek über eine abstrahierte Schnittstelle, ohne spezielles Protokollwissen einheitlich bedient werden. Die Bibliothek unterstützt TCP und das SSH-Protokoll für die Kommunikation mit RTR-Servern. Eine modulare Erweiterung um zusätzliche Transportprotokolle wird von der Architektur unterstützt. Das RTR-Protokoll wurde als Zustandsdiagramm modelliert und implementiert. Es wurde eine Failover-Funktionalität integriert, welche mehrere RTR-Server in Gruppen zusammenfasst und beim Auftreten von Fehlern automatisiert Verbindungen zu anderen RTR-Servern einer anderen Gruppe herzustellen. Die Interoperabilität der Implementierung wurde mithilfe den RTR-Servern rtr-origin [6] sowie RPKI-Validator [16] erfolgreich getestet. In einer abschließenden Evaluation wurde die Performanz der Bibliothek bei der Validierung von BGP-Annoncierungen untersucht. Auf einem Rechner mit einer AMD Athlon 64 X2 CPU und 2 GB wurde die Hardware nur minimal belastet und bescheinigte die Einsetzbarkeit der Bibliothek auch auf Hardware mit geringer Leistungsfähigkeit.

Die Implementierung der Bibliothek wurde unter der *GNU Lesser General Public License* [9] lizenziert und steht für jeden zur freien Benutzung auf der Internetseite `http://rpki.realmv6.org/` zur Verfügung. Um die Installation der Bibliothek zu erleichtern, wurde mithilfe des CMake Buildsystems [5] Scripte zum Kompilieren und Installieren der Bibliothek erstellt. Mithilfe von Doxygen [8] wurde ein API Dokumentationen erzeugt, welche die Benutzung der Bibliotheksfunktionen beschreibt. Die erste Alpha Version wurde am 07.09.2011 auf der Webseite der Öffentlichkeit zur Verfügung gestellt. Die Veröffentlichung wurde unter anderem mit einem Artikel auf der Internetseite der RIPE von Matthias Wählisch [59] sowie einem Beitrag in Mailingliste der SIDR Arbeitsgruppe bekanntgegeben [18]. Während der Entwicklung der Bibliothek wurde auf der 80. und 81. IETF-Konferenz

die Bibliothek von Matthias Wählisch vorgestellt und diskutiert. Erste Anwendung fand die Bibliothek bei der RIPE zum Testen des RPKI-Validators.

Damit die Bibliothek auch in Zukunft für RTR-Implementierungen in Routern oder zum Testen von RTR-Servern benutzt werden kann, muss die Implementierung des Protokolls fortlaufend anhand der RPKI-RTR Spezifikation aktualisiert werden. Eine Erweiterung der Bibliothek um weitere sichere Netzwerkprotokolle wäre empfehlenswert, um Anwendern eine größere Auswahl zu bieten und die optionalen Anforderungen der Spezifikation vollständig zu implementieren. Die Architektur der Bibliothek unterstützt nicht verschiedene RTR-Protokollversionen. Falls zukünftig mehrere RTR-Versionen gleichzeitig existieren und verwendet werden, muss die Architektur angepasst werden.

Für die Bibliothek existieren momentan nur wenige Unit-Tests, welche nur einen kleinen Teil der Implementierung abdecken. Es ist empfehlenswert, möglichst die komplette Funktionalität mit Unit-Tests verifizieren zu können. Eine große Abdeckung durch Tests verhindert, dass Fehler nicht entdeckt werden und infolgedessen fehlerhafte Versionen veröffentlicht werden. Falls benötigt kann die Speichernutzung der `pfx_table` Implementierung noch weiter optimiert werden. Beispielsweise benötigen IPv4-Adressen in der Prefix Validation Table zurzeit denselben Speicherplatz wie eine IPv6 Adresse, was nicht optimal ist. Die Performanz der Prefix Validation Table kann ebenso weiter optimiert werden, indem beispielsweise der verwendete LPFST mit geringem Aufwand durch ein *Partition Longest Prefix First Search Tree* [62] ersetzt wird. Um die Einsatzfähigkeit der Bibliothek auf BGP-Routern genauer zu untersuchen, sollte die Bibliothek in einen freien BGP-Daemon integriert und die Funktionfähigkeit, z. B. mit BRITE [4], überprüft werden. Ergänzend wäre eine weiterführende Analyse interessant, in der untersucht wird wieviele von den als *Invalid* klassifizierten Routen tatsächliche Präfix-Hijacks sind.

Literaturverzeichnis

[1] *BBN Technolgies Website*. http://www.bbn.com/

[2] *BGP Monitoring System - Colorado State University*. http://bgpmon.netsec.colostate.edu/

[3] *BGP Secure Routing Extension (BGP-SRx)*. http://www-x.antd.nist.gov/bgpsrx/

[4] *BRITE - BGPSEC / RPKI Interoperability Test and Evaluation*. http://brite.antd.nist.gov/statics/about

[5] *CMake - Cross Plattform Make*. http://www.cmake.org

[6] *Cynical rsync*. http://subvert-rpki.hactrn.net/rcynic/

[7] *Discussion on the SIDR mailinglist about the RTR transport protocol*. http://www.mail-archive.com/sidr@ietf.org/msg02495.html

[8] *Doxygen*. http://www.stack.nl/~dimitri/doxygen/index.html

[9] *GNU Lesser General Public License*. http://www.gnu.org/licenses/lgpl.html

[10] *GNU Netcat*. http://netcat.sourceforge.net/

[11] *Internet Assigned Numbers Authority*. http://www.iana.org

[12] Internet Assigned Numbers Authority: Number Resources. http://www.iana.org/numbers/

[13] *libssh - channel_select bug report*. https://red.libssh.org/issues/56

[14] *libssh - The SSH Library*. http://www.libssh.org/

[15] *Quagga Routing Suite*. http://www.quagga.net

[16] *RIPE RPKI-Validator*. http://labs.ripe.net/Members/agowland/ripe-ncc-validator-for-resource-certification

[17] *Rsync Website*. http://rsync.samba.org/

[18] *RTRlib announcement on the SIDR mailinglist*. http://www.ietf.org/mail-archive/web/sidr/current/msg03286.html

[19] *Secure Inter-Domain Routing Working Group*. http://datatracker.ietf.org/wg/sidr/charter/

[20] *The BIRD internet routing daemon*. http://bird.network.cz/

[21] *The Internet Engineering Task Force (IETF)*. http://www.ietf.org/

[22] YouTube Hijacking: A RIPE NCC RIS case study. (2008), März. http://www.ripe.net/internet-coordination/news/industry-developments/youtube-hijacking-a-ripe-ncc-ris-case-study

[23] Chinese ISP hijacks the Internet. (2010), Januar. http://www.omninerd.com/articles/Did_China_Hijack_15_of_the_Internet_Routers_BGP_and_Ignorance

[24] *BGPmon route statistics*. http://bgpmon.net/stat.php. Version: November 2011

[25] *Hurriance Electric Internet Services: BGP Peer Report*. http://bgp.he.net/report/peers. Version: November 2011

[26] *RIPE's number of RPKI certificate statistic*. http://certification-stats.ripe.net/. Version: Oktober 2011

[27] RPKI. (2011), Januar. http://bgpmon.net/blog/?p=414

[28] *The BGP Instability Report*. http://bgpupdates.potaroo.net/instability/bgpupd.html. Version: November 2011

[29] Aboba, B. ; Dixon, W.: *IPsec-Network Address Translation (NAT) Compatibility Requirements*. RFC 3715 (Informational). http://www.ietf.org/rfc/rfc3715.txt. Version: März 2004 (Request for Comments)

[30] Adams, C. ; Farrell, S.: *Internet X.509 Public Key Infrastructure Certificate Management Protocols*. RFC 2510 (Proposed Standard). http://www.ietf.org/rfc/rfc2510.txt. Version: März 1999 (Request for Comments). – Obsoleted by RFC 4210

[31] Bellovin, Steven ; Bush, Randy ; Ward, David: Security Requirements for BGP Path Validation / IETF. 2011 (00). – Internet-Draft – work in progress

[32] Bush, Randy: RPKI-Based Origin Validation Operation / IETF. 2011 (11). – Internet-Draft – work in progress

[33] Bush, Randy ; Austein, Rob: The RPKI/Router Protocol / IETF. 2011 (18). – Internet-Draft – work in progress

[34] Butler, K. ; Farley, T. R. ; McDaniel, P. ; Rexford, J.: A Survey of BGP Security Issues and Solutions. In: *Proceedings of the IEEE* 98 (2010), Januar, Nr. 1, 100–122. http://dx.doi.org/10.1109/JPROC.2009.2034031. – DOI 10.1109/JPROC.2009.2034031. – ISSN 0018–9219

[35] Butler, Kevin ; Farley, Toni ; McDaniel, Patrick ; Rexford, Jennifer: A Survey of BGP Security Issues and Solutions. In: *Proc. of the IEEE* 98 (2010), January, Nr. 1, S. 100–122

[36] Carl Kalapesi, Paul Z. Sarah Willersdorf W. Sarah Willersdorf: *The connected Kingdom*. Boston Consulting Group, 2010 http://www.connectedkingdom.co.uk/the-report/

[37] Cavedon, Ludovico ; Kruegel, Christopher ; Vigna, Giovanni: Are BGP routers open to attack? an experiment. In: *Proceedings of the 2010 IFIP WG 11.4 international conference on Open research problems in network security*. Berlin, Heidelberg : Springer-Verlag, 2011 (iNetSec'10). – ISBN 978–3–642–19227–2, 88–103

[38] Dierks, T. ; Rescorla, E.: *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246 (Proposed Standard). http://www.ietf.org/rfc/rfc5246.txt. Version: August 2008 (Request for Comments). – Updated by RFCs 5746, 5878, 6176

[39] Eddy, W.: *TCP SYN Flooding Attacks and Common Mitigations*. RFC 4987 (Informational). http://www.ietf.org/rfc/rfc4987.txt. Version: August 2007 (Request for Comments)

[40] Ermert, Monika: Report of the RIPE 62. In: *RIPE 62*, 2011

[41] Heffernan, A.: *Protection of BGP Sessions via the TCP MD5 Signature Option*. RFC 2385 (Proposed Standard). http://www.ietf.org/rfc/rfc2385.txt. Version: August 1998 (Request for Comments). – Obsoleted by RFC 5925

[42] Hubbard, K. ; Kosters, M. ; Conrad, D. ; Karrenberg, D. ; Postel, J.: *Internet Registry IP Allocation Guidelines*. RFC 2050 (Best Current Practice). http://www.ietf.org/rfc/rfc2050.txt. Version: November 1996 (Request for Comments)

[43] Huston, Geoff ; Michaelson, George ; Loomans, Robert: A Profile for X.509 PKIX Resource Certificates / IETF. 2011 (22). – Internet-Draft – work in progress

[44] Kent, S.: Securing the border gateway protocol: A status update. In: *Communications and Multimedia Security. Advanced Techniques for Network and Data Protection* (2003), S. 40–53

[45] Kent, S. ; Lynn, C. ; Seo, K.: Secure border gateway protocol (S-BGP). In: *Selected Areas in Communications, IEEE Journal on* 18 (2000), Nr. 4, S. 582–592

[46] Kent, S. ; Seo, K.: *Security Architecture for the Internet Protocol*. RFC 4301 (Proposed Standard). http://www.ietf.org/rfc/rfc4301.txt. Version: Dezember 2005 (Request for Comments). – Updated by RFC 6040

[47] Kumari, Warren: Deprecation of BGP AS_SET, AS_CONFED_SET. / IETF. 2010 (01). – Internet-Draft – work in progress

[48] Lepinski, Matt: BGPSEC Protocol Specification / IETF. 2011 (00). – Internet-Draft – work in progress

[49] Lepinski, Matt ; Kent, Stephen: An Infrastructure to Support Secure Internet Routing / IETF. 2011 (13). – Internet-Draft – work in progress

[50] Lynn, Charles: Secure BGP (S-BGP) / IETF. 2003 (01). – Internet-Draft – work in progress

[51] Mohapatra, Pradosh ; Scudder, John ; Ward, David ; Bush, Randy ; Austein, Rob: BGP Prefix Origin Validation / IETF. 2011 (02). – Internet-Draft – work in progress

[52] Nordström, Ola ; Dovrolis, Constantinos: Beware of BGP attacks. In: *SIGCOMM Comput. Commun. Rev.* 34 (2004), April, 1–8. http://dx.doi.org/http://doi.acm.org/10.1145/997150.997152. – DOI http://doi.acm.org/10.1145/997150.997152. – ISSN 0146–4833

[53] POSIX.1-2008 ; Fox, Cathy (Hrsg.): *The Open Group Base Specifications*. Also published as IEEE Std 1003.1-2008. http://dx.doi.org/10.1109/IEEESTD.2008.4694976. Version: Juli 2008

[54] Postel, J.: *Transmission Control Protocol*. RFC 793 (Standard). http://www.ietf.org/rfc/rfc793.txt. Version: September 1981 (Request for Comments). – Updated by RFCs 1122, 3168, 6093

[55] Rekhter, Y. ; Li, T. ; Hares, S.: *A Border Gateway Protocol 4 (BGP-4)*. RFC 4271 (Draft Standard). http://www.ietf.org/rfc/rfc4271.txt. Version: Januar 2006 (Request for Comments). – Updated by RFC 6286

[56] Touch, J. ; Mankin, A. ; Bonica, R.: *The TCP Authentication Option*. RFC 5925 (Proposed Standard). http://www.ietf.org/rfc/rfc5925.txt. Version: Juni 2010 (Request for Comments)

[57] Touch, Joseph ; Mankin, Allison ; Bonica, Ronald: The TCP Authentication Option / IETF. 2008 (01). – Internet-Draft – work in progress

[58] Vohra, Q. ; Chen, E.: *BGP Support for Four-octet AS Number Space*. RFC 4893 (Proposed Standard). http://www.ietf.org/rfc/rfc4893.txt. Version: Mai 2007 (Request for Comments)

[59] Wählisch, Matthias: RIPE - Beta Version of the RPKI RTR Client C Library Released. (2011), September. https://labs.ripe.net/Members/waehlisch/beta-version-of-the-rpki-rtr-client-c-library-released

[60] Watson, P.: *Slipping in the Window: TCP Reset attacks*. 2004

[61] Wong, Edmund L. ; Shmatikov, Vitaly: Get off my prefix! the need for dynamic, gerontocratic policies in inter-domain routing. In: *Dependable Systems and Networks, International Conference on* 0 (2011), S. 233–244. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/DSN.2011.5958222. – DOI http://doi.ieeecomputersociety.org/10.1109/DSN.2011.5958222. ISBN 978–1–4244–9232–9

[62] Wuu, Lih-Chyau ; Liu, Tzong-Jye ; Chen, Kuo-Ming: A longest prefix first search tree for IP lookup. In: *Comput. Netw.* 51 (2007), August, 3354–3367. http://dx.doi.org/10.1016/j.comnet.2007.01.023. – DOI 10.1016/j.comnet.2007.01.023. – ISSN 1389–1286

[63] Ylonen, T. ; Lonvick, C.: *The Secure Shell (SSH) Authentication Protocol*. RFC 4252 (Proposed Standard). http://www.ietf.org/rfc/rfc4252.txt. Version: Januar 2006 (Request for Comments)

A. Anhang

A.1. Quellcodeausschnitte der Bibliothek

```
1  int rtr_sync(rtr_socket* rtr_socket){
2      char pdu[RTR_MAX_PDU_LEN];
3
4      int rtval = rtr_receive_pdu(rtr_socket, pdu,
       RTR_MAX_PDU_LEN, RTR_RECV_TIMEOUT);
5      if(rtval == TR_WOULDBLOCK){
6          rtr_change_socket_state(rtr_socket,
           RTR_ERROR_TRANSPORT);
7          return RTR_ERROR;
8      }
9      else if(rtval < 0)
10         return RTR_ERROR;
11     pdu_type type = rtr_get_pdu_type(pdu);
12
13     //ignore serial_notify PDUs
14     while(type == SERIAL_NOTIFY){
15         RTR_DBG1("Ignoring Serial Notify");
16         rtval = rtr_receive_pdu(rtr_socket, pdu,
           RTR_MAX_PDU_LEN, RTR_RECV_TIMEOUT);
17         if(rtval == TR_WOULDBLOCK){
18             rtr_change_socket_state(rtr_socket,
               RTR_ERROR_TRANSPORT);
19             return RTR_ERROR;
20         }
21         else if(rtval < 0)
22             return RTR_ERROR;
23     type = rtr_get_pdu_type(pdu);
24     }
25
26     if(type == ERROR){
27         rtr_handle_error_pdu(rtr_socket, pdu);
28         return RTR_ERROR;
```

```
29      }
30      else if(type == CACHE_RESET){
31          RTR_DBG1("Cache Reset PDU received");
32          rtr_change_socket_state(rtr_socket,
             RTR_ERROR_NO_INCR_UPDATE_AVAIL);
33          return RTR_ERROR;
34      }

36      if(type == CACHE_RESPONSE){
37          RTR_DBG1("Cache Response PDU received");
38          pdu_header* cr_pdu = (pdu_header*) pdu;
39          //set connection nonce
40          if(rtr_socket->request_nonce){
41              if(rtr_socket->last_update != 0){
42                  //if this isnt the first sync, but we already
                     received records, delete old records in the
                     pfx_table
43                  pfx_table_src_remove(rtr_socket->pfx_table,
                     (uintptr_t) rtr_socket);
44                  rtr_socket->last_update = 0;
45              }
46              rtr_socket->nonce = cr_pdu->reserved;
47          }
48          else{
49              if(rtr_socket->nonce != cr_pdu->reserved){
50                  char* txt = "Wrong NONCE in CACHE RESPONSE
                     PDU";
51                  rtr_send_error_pdu(rtr_socket, NULL, 0,
                     CORRUPT_DATA, txt, sizeof(txt));
52                  rtr_change_socket_state(rtr_socket,
                     RTR_ERROR_FATAL);
53                  return RTR_ERROR;
54              }
55          }
56      }

58      //receive IPV4/IPV6 PDUs till EOD
59      do {
60          rtval = rtr_receive_pdu(rtr_socket, pdu,
             RTR_MAX_PDU_LEN, RTR_RECV_TIMEOUT);
61          if(rtval == TR_WOULDBLOCK){
62              rtr_change_socket_state(rtr_socket,
                 RTR_ERROR_TRANSPORT);
63              return RTR_ERROR;
```

```
64          }
65          else if(rtval < 0)
66              return RTR_ERROR;
67          type = rtr_get_pdu_type(pdu);
68          if(type == IPV4_PREFIX || type == IPV6_PREFIX){
69              if(rtr_update_pfx_table(rtr_socket, pdu) ==
                 RTR_ERROR)
70                  return RTR_ERROR;
71          }
72          else if(type == EOD){
73              pdu_eod* eod_pdu = (pdu_eod*) pdu;

75              if(eod_pdu->nonce != rtr_socket->nonce){
76                  char txt[67];
77                  snprintf(txt, sizeof(txt),"Expected Nonce: %u,
                     received Nonce. %u in EOD PDU",
                     rtr_socket->nonce, eod_pdu->nonce);
78                  rtr_send_error_pdu(rtr_socket, pdu,
                     RTR_MAX_PDU_LEN, CORRUPT_DATA, txt,
                     strlen(txt) + 1);
79                  rtr_change_socket_state(rtr_socket,
                     RTR_ERROR_FATAL);
80                  return RTR_ERROR;
81              }
82              rtr_socket->serial_number = eod_pdu->sn;
83          }
84          else if(type == ERROR){
85              rtr_handle_error_pdu(rtr_socket, pdu);
86              return RTR_ERROR;
87          }
88          else if(type == SERIAL_NOTIFY){
89              RTR_DBG1("Ignoring Serial Notify");
90          }
91          else{
92              RTR_DBG("Received unexpected PDU (Type: %u)",
                 ((pdu_header*) pdu)->type);
93              char* txt = "unexpected PDU received during data
                 sync";
94              rtr_send_error_pdu(rtr_socket, pdu,
                 sizeof(pdu_header), CORRUPT_DATA, txt,
                 sizeof(txt));
95              return RTR_ERROR;
96          }
97      } while(type != EOD);
```

```
98      rtr_socket->request_nonce = false;
99      if (rtr_set_last_update(rtr_socket) == RTR_ERROR)
100         return RTR_ERROR;
101     RTR_DBG("Sync successfull, Nonce: %u, SN: %u",
         rtr_socket->nonce, rtr_socket->serial_number);
102     return RTR_SUCCESS;
103 }
```

Listing A.1: rtr_sync Funktion

A.2. Evaluationsprogramme

```
1  from socket import *
2  import re
3  import xml
4  import xml.dom.minidom as dom
5  import string
6  import signal

8  def parse(xml):
9    l = []
10   try:
11     tree = dom.parseString(xml)
12   except xml.parsers.expat.ExpatError:
13     print >> sys.stderr, xml
14     return []

16   d = {}
17   for i in tree.firstChild.childNodes:
18     if i.nodeName == "ASCII_MSG":
19       schluessel = wert = None
20       for elems in i.childNodes:
21         if elems.nodeName == "UPDATE":
22           for update in elems.childNodes:
23             if(update.nodeName == "NLRI"):
24               for nlri in update.childNodes:
25                 if(nlri.nodeName == "PREFIX"):
26                   for prefix in nlri.childNodes:
27                     if(prefix.nodeName == "ADDRESS"):
28                       for txt in prefix.childNodes:
29                         s = txt.data.split("/")
30                         if not "prefix" in d:
```

```
                                d["prefix"] = []
                            p = {}
                            p["address"] =  s[0]
                            p["len"] = s[1]
                            d["prefix"].append(p)
                if(update.nodeName == "PATH_ATTRIBUTES"):
                  for pattr in update.childNodes:
                    if(pattr.nodeName == "ATTRIBUTE"):
                      for pattr in pattr.childNodes:
                        if(pattr.nodeName == "AS_PATH"):
                          for asPath in pattr.childNodes:
                            if(asPath.nodeName == "AS_SEG"):
                              length =
                               int(asPath.getAttribute("length"))
                              origin_as =
                               asPath.childNodes[length-1]
                              d["origin_as"] =
                               str(origin_as.childNodes[0].data)
        break;
    if("prefix" in d):
      l.append(d)
    return l


cli = socket( AF_INET,SOCK_STREAM)
cli.connect(("livebgp.netsec.colostate.edu", 50001))
data =""
msg = ""
signal.signal(signal.SIGPIPE, signal.SIG_DFL)
signal.signal(signal.SIGINT, signal.SIG_IGN)

while True:
    data = cli.recv(1024) #14= </BGP_MESSAGE>
    if(re.search('</BGP_MESSAGE>', msg)):
      l = msg.split('</BGP_MESSAGE>', 1)
      bgp_update = l[0] + "</BGP_MESSAGE>"
      bgp_update = string.replace(bgp_update, "<xml>", "")
      d = parse(bgp_update)
      msg = ''.join(l[1:])
      for i in d:
        for j in i["prefix"]:
          print i["origin_as"]+ "\t"+ j["address"] + "\t" +
           j["len"]
    msg += str(data)
```

Listing A.2: Python BGPmon XML-Parser

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netdb.h>
#include "rtrlib/rtrlib.h"
#include "getusage.c"
#include <signal.h>
#include <stdio_ext.h>

FILE* pref_f;
FILE* bench_f;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

unsigned long int valid_state = 0;
unsigned long int invalid_state = 0;
unsigned long int not_found_state = 0;

void sig_handler(){
    fprintf(stderr, "FLUSHING\n");
    fflush(NULL);
}

void read_announcement(unsigned int* asn, char* prefix,
 unsigned int* prefix_len){
    char line[512];

    if(fscanf(stdin,"%u\t%s\t%u\n", asn, prefix, prefix_len)
     == EOF)
        perror("FSCANF ERROR\n");
        exit(EXIT_FAILURE);
}
void* log_thr(){
    pid_t pid = getpid();

    struct pstat last_cpu;
    struct pstat cur_cpu;
    double ucpu;
```

```
40      double scpu;
41      time_t curtime;
42      time_t starttime = time(NULL);

44      if(get_usage(pid, &cur_cpu) == -1){
45      fprintf(stderr, "GET USAGE ERROR\n");
46          exit(EXIT_FAILURE);
47      }

49      bench_f = fopen("benchmark.log", "w");
50      fprintf(bench_f, "%s",
         "#Second\tVALID_STATES\tINVALID_STATES\tNOTFOUND_STATES\
51      \tCPU_USAGE\tVSIZE\tRSS\n");
52      while(true){
53          last_cpu = cur_cpu;
54          get_usage(pid, &cur_cpu);
55          calc_cpu_usage(&cur_cpu, &last_cpu, &ucpu, &scpu);
56          curtime = time(NULL);
57          pthread_mutex_lock(&mutex);
58          fprintf(bench_f,"%lu\t%lu\t%lu\t%lu\t%f\t%lu\t%lu\n",
             curtime - starttime, valid_state, invalid_state,
             not_found_state, ucpu+scpu, cur_cpu.vsize,
             cur_cpu.rss);
59          valid_state = 0;
60          invalid_state = 0;
61          not_found_state = 0;
62          pthread_mutex_unlock(&mutex);
63          if((curtime -starttime) >= 86400){
64              fflush(NULL);
65              exit(EXIT_SUCCESS);
66          }
67          sleep(60);
68      }
69  }

71  void fill_pfx_table(pfx_table* pfxt){
72      FILE* df = fopen("pfx_records.txt", "r");
73      if(df == NULL)
74          exit(EXIT_FAILURE);
75      char ip[256];
76      unsigned int pref_len;
77      unsigned int asn;
78      unsigned int count = 0;
79      pfx_record rec;
```

```
80      while (fscanf(df, "%s %u %u", ip, &pref_len, &asn) != EOF){
81          rec.max_len = pref_len;
82          rec.min_len = pref_len;
83          rec.asn = asn;
84          rec.socket_id = 990;
85          printf("%u: IP: %s/%u %u\n", count, ip, pref_len, asn);
86          if(rtr_str_to_ipaddr(ip, &(rec.prefix)) == -1){
87              fprintf(stderr, "rtr_str_to_ipaddr_error\n");
88              exit(EXIT_FAILURE);
89          }
90          if(pfx_table_add(pfxt, &rec) == PFX_ERROR){
91              fprintf(stderr, "pfx_table_add error\n");
92              exit(EXIT_FAILURE);
93          }
94          char t[256];
95          rtr_ipaddr_to_str(&(rec.prefix), t, sizeof(t));

97          pfxv_state state;
98          pfx_table_validate(pfxt, rec.asn, &(rec.prefix),
             rec.min_len, &state);
99         if(state != BGP_PFXV_STATE_VALID){
100            printf("Error added pfx_record couldnt be validated
                as valid\n");
101            exit(EXIT_FAILURE);
102        }
103         count++;
104     }
105     printf("added %u records to the pfx_table\n", count);
106 }

108 int main()
109 {
110     signal(SIGINT, &sig_handler);

112     tr_tcp_config tcp_config = {
113         "141.22.26.252",          //IP
114         "8282"                      //Port
115     };
116     tr_socket* tr_tcp;
117     tr_tcp_init(&tcp_config, &tr_tcp);
118     rtr_socket rtr_tcp;
119     rtr_tcp.tr_socket = tr_tcp;

121     tr_tcp_config tcp1_config = {
```

```
122         "141.22.27.161",           //IP
123         "42420"                       //Port
124     };
125     tr_socket* tr_tcp1;
126     tr_tcp_init(&tcp1_config, &tr_tcp1);
127     rtr_socket rtr_tcp1;
128     rtr_tcp1.tr_socket = tr_tcp1;

130     rtr_mgr_group groups[2];
131     groups[0].sockets_len = 1;
132     groups[0].sockets = malloc(1 * sizeof(rtr_socket*));
133     groups[0].sockets[0] = &rtr_tcp;
134     groups[0].preference = 4;
135     groups[1].sockets_len = 1;
136     groups[1].sockets = malloc(1 * sizeof(rtr_socket*));
137     groups[1].sockets[0] = &rtr_tcp1;
138     groups[1].preference = 2;

140     rtr_mgr_config conf;
141     conf.groups = groups;
142     conf.len = 2;

144     rtr_mgr_init(&conf, 240, 520, NULL);
145     rtr_mgr_start(&conf);

147     //fill_pfx_table(groups[1].sockets[0]->pfx_table);
148     while(!rtr_mgr_group_in_sync(&conf)){
149         sleep(1);
150     }


153     unsigned int asn;
154     char prefix[256];
155     unsigned int prefix_len;
156     ip_addr ip_addr;
157     pfxv_state state;
158     pref_f = fopen("prefixes.log", "w");
159     if(pref_f ==NULL)
160         exit(EXIT_FAILURE);

162     //purge stdin buffer
163     __fpurge(stdin);

165     pthread_t thrd_id;
```

```
166     pthread_create(&thrd_id, NULL, &log_thr, NULL);


169     printf("Benchmark started\n");

171     while(true){
172         read_announcement(&asn, prefix, &prefix_len);
173         if(rtr_str_to_ipaddr(prefix, &ip_addr) == -1){
174             fprintf(stderr, "ERROR STR TO IPADDR\n");
175             exit(EXIT_FAILURE);
176         }
177         if(rtr_mgr_validate(&conf, asn, &ip_addr, prefix_len,
            &state) == -1){
178             fprintf(stderr, "VALIDATE ERROR\n");
179             exit(EXIT_FAILURE);
180         }
181         if(fprintf(pref_f, "%s %u %u ", prefix, prefix_len,
            asn) < 0)
182             perror("printf error");
183         if(state == BGP_PFXV_STATE_VALID){
184             pthread_mutex_lock(&mutex);
185             valid_state++;
186             pthread_mutex_unlock(&mutex);
187             if (fprintf(pref_f, "VALID\n") < 0 )
188                 perror("printf error");
189         }
190         else if (state == BGP_PFXV_STATE_INVALID){
191             pthread_mutex_lock(&mutex);
192             invalid_state++;
193             pthread_mutex_unlock(&mutex);
194             if (fprintf(pref_f, "INVALID\n") < 0)
195                 perror("printf error");
196         }
197         else if (state == BGP_PFXV_STATE_NOT_FOUND){
198             pthread_mutex_lock(&mutex);
199             not_found_state++;
200             pthread_mutex_unlock(&mutex);
201             if (fprintf(pref_f, "NOT_FOUND\n") < 0)
202                 perror("printf error");
203         }
204     }
205 }
```

Listing A.3: Benchmark tool

A.3. Unit-Tests

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <assert.h>
#include <string.h>
#include <arpa/inet.h>
#include "rtrlib/lib/ipv6.h"
#include "rtrlib/lib/log.h"
#include "rtrlib/lib/utils.h"
#include "rtrlib/pfx/lpfst/lpfst.c"

void get_bits_testv4(){
    ip_addr addr;
    addr.ver = IPV4;

    ip_addr result;
    addr.u.addr4.addr=0xAABBCC22;

    result = ip_addr_get_bits(&addr, 0, 32);
    assert(result.u.addr4.addr == 0xAABBCC22);

    result = ip_addr_get_bits(&addr, 0, 1);
    assert(result.u.addr4.addr == 0x80000000);

    result = ip_addr_get_bits(&addr, 1, 1);
    assert(result.u.addr4.addr == 0);

    result = ip_addr_get_bits(&addr, 2, 1);
    assert(result.u.addr4.addr == 0x20000000);

    result = ip_addr_get_bits(&addr, 0, 8);
    assert(result.u.addr4.addr == 0xAA000000);

    result = ip_addr_get_bits(&addr, 8, 8);
    assert(result.u.addr4.addr == 0x00BB0000);


    rtr_str_to_ipaddr("10.10.10.0", &addr);

    result = ip_addr_get_bits(&addr, 0, 8);
    assert(rtr_cmp_ipv4(&result, "10.0.0.0"));
```

```
43      result = ip_addr_get_bits(&addr, 0, 16);
44      assert(rtr_cmp_ipv4(&result, "10.10.0.0"));

46      result = ip_addr_get_bits(&addr, 8, 8);
47      assert(rtr_cmp_ipv4(&result, "0.10.0.0"));

49      result = ip_addr_get_bits(&addr, 8, 24);
50      assert(rtr_cmp_ipv4(&result, "0.10.10.0"));

52      result = ip_addr_get_bits(&addr, 31, 1);
53      assert(result.u.addr4.addr == 0);

55      result = ip_addr_get_bits(&addr, 0, 1);
56      assert(result.u.addr4.addr == 0);

58      result = ip_addr_get_bits(&addr, 3, 3);
59      assert(rtr_cmp_ipv4(&result, "8.0.0.0"));

61      assert(rtr_str_to_ipaddr("132.200.0.0", &addr) == 0);
62      result = ip_addr_get_bits(&addr, 0, 1);
63      assert(result.u.addr4.addr == 0x80000000);

65      assert(rtr_str_to_ipaddr("101.200.0.0", &addr) == 0);
66      result = ip_addr_get_bits(&addr, 0, 1);
67      printf("%x\n", result.u.addr4.addr);
68      assert(result.u.addr4.addr == 0);

70      addr.u.addr4.addr = 0x6D698000;
71      result = ip_addr_get_bits(&addr, 0, 19);
72      assert(result.u.addr4.addr == 0x6D698000);
73      /*
74      rtr_str_to_ipaddr("109.105.128.0", &addr);
75      result = ip_addr_get_bits(&addr, 0, 8);
76      printf("%u\n", result.u.addr4.addr);
77      */

79      char buf[INET_ADDRSTRLEN];
80      assert(rtr_str_to_ipaddr("10.10.10.5", &addr) == 0);
81      assert(rtr_ipaddr_to_str(&addr, buf, sizeof(buf)) == 0);
82      assert(strcmp("10.10.10.5", buf) == 0);

84  }

86  void get_bits_testv6(){
```

```
87      ip_addr addr;
88      addr.ver = IPV6;
89      addr.u.addr6.addr[0] = 0x22AABBCC;
90      addr.u.addr6.addr[1] = 0xDDEEFF99;
91      addr.u.addr6.addr[2] = 0x33001122;
92      addr.u.addr6.addr[3] = 0x33445566;

94      ip_addr result;

96      result = ip_addr_get_bits(&addr, 0, 128);
97      assert(result.u.addr6.addr[0] == addr.u.addr6.addr[0] &&
         result.u.addr6.addr[1] == addr.u.addr6.addr[1] &&
         result.u.addr6.addr[2] == addr.u.addr6.addr[2] &&
         result.u.addr6.addr[3] == addr.u.addr6.addr[3]);

99      result = ip_addr_get_bits(&addr, 0, 64);
100     assert(result.u.addr6.addr[0] == addr.u.addr6.addr[0] &&
         result.u.addr6.addr[1] == addr.u.addr6.addr[1] &&
         result.u.addr6.addr[2] == 0 && result.u.addr6.addr[3] ==
         0);

102     bzero(&result, sizeof(result));
103     result = ip_addr_get_bits(&addr, 64, 64);
104     assert(result.u.addr6.addr[0] == 0);
105     assert(result.u.addr6.addr[1] == 0);
106     assert(result.u.addr6.addr[2] == addr.u.addr6.addr[2]);
107     assert(result.u.addr6.addr[3] == addr.u.addr6.addr[3]);


110     result = ip_addr_get_bits(&addr, 0, 8);
111     assert(result.u.addr6.addr[0] == 0x22000000 &&
         result.u.addr6.addr[1] == 0);

113     result = ip_addr_get_bits(&addr, 64, 8);
114     assert(result.u.addr6.addr[1] == 0 &&
         result.u.addr6.addr[2] == 0x33000000);

116     result = ip_addr_get_bits(&addr, 7, 8);
117     assert(result.u.addr6.addr[0] == 0xAA0000 &&
         result.u.addr6.addr[1] == 0);

119     result = ip_addr_get_bits(&addr, 68, 7);
120     assert(result.u.addr6.addr[0] == 0 &&
         result.u.addr6.addr[2] == 0x03000000);
```

```
122     char buf[INET6_ADDRSTRLEN];

124     rtr_str_to_ipaddr("fe80::862b:2bff:fe9a:f50f", &addr);
125     addr.ver=IPV6;
126     assert(addr.u.addr6.addr[0] == 0xfe800000);
127     assert(addr.u.addr6.addr[1] == 0);
128     assert(addr.u.addr6.addr[2] == 0x862b2bff);
129     assert(addr.u.addr6.addr[3] == 0xfe9af50f);

131     assert(rtr_str_to_ipaddr("2001::", &addr) == 0);
132     assert(addr.u.addr6.addr[0] == 0x20010000);
133     assert(addr.u.addr6.addr[1] == 0);
134     assert(addr.u.addr6.addr[2] == 0);
135     assert(addr.u.addr6.addr[3] == 0);

137     assert(rtr_ipaddr_to_str(&addr, buf, sizeof(buf)) == 0);
138     assert(strcmp("2001::", buf) == 0);

140     rtr_str_to_ipaddr("2001:0db8:85a3:08d3:1319:8a2e:0370:7344",
         &addr);
141     assert(rtr_ipaddr_to_str(&addr, buf, sizeof(buf)) == 0);
142     printf("%s\n", buf);
143     assert(strcmp("2001:db8:85a3:8d3:1319:8a2e:370:7344", buf)
         == 0);

145     result = ip_addr_get_bits(&addr, 0, 16);
146     assert(rtr_ipaddr_to_str(&result, buf, sizeof(buf)) == 0);
147     assert(rtr_cmp_ipv6(&result, "2001::"));

149     result = ip_addr_get_bits(&addr, 16, 16);
150     assert(rtr_ipaddr_to_str(&result, buf, sizeof(buf)) == 0);
151     printf("%s\n",buf);
152     assert(rtr_cmp_ipv6(&result, "0:db8::"));
153     result = ip_addr_get_bits(&addr, 0, 1);
154     assert(rtr_cmp_ipv6(&result, "::"));
155 }

157 void lpfst_test(){
158     ip_addr addr;
159     addr.ver = IPV4;
160     lpfst_node* result;
161     unsigned int lvl = 0;
```

```
163     //n1
164     lpfst_node n1;
165     n1.len = 16;
166     n1.lchild = NULL;
167     n1.rchild = NULL;
168     n1.parent = NULL;
169     n1.data = NULL;
170     rtr_str_to_ipaddr( "100.200.0.0", &(n1.prefix));

172     rtr_str_to_ipaddr( "100.200.0.0", &(addr));
173     lvl = 0;
174     result = lpfst_lookup(&n1, &addr, 16, &lvl);
175     assert(result != NULL);
176     assert(rtr_cmp_ipv4(&(result->prefix), "100.200.0.0"));

178     rtr_str_to_ipaddr( "100.200.30.0", &(addr));
179     lvl = 0;
180     result = lpfst_lookup(&n1, &addr, 16, &lvl);
181     assert(result != NULL);
182     assert(rtr_cmp_ipv4(&(result->prefix), "100.200.0.0"));


185     //n2
186     lpfst_node n2;
187     n2.len = 16;
188     n2.lchild = NULL;
189     n2.rchild = NULL;
190     n2.parent = NULL;
191     n2.data = NULL;
192     rtr_str_to_ipaddr("132.200.0.0", &(n2.prefix));
193     lpfst_insert(&n1, &n2, 0);

195     rtr_str_to_ipaddr("132.200.0.0", &(addr));
196     lvl = 0;
197     result = lpfst_lookup(&n1, &addr, 16, &lvl);
198     assert(result != NULL);
199     assert(rtr_cmp_ipv4(&(result->prefix), "132.200.0.0"));
200     assert(n1.rchild == &n2);


203     //n3
204     lpfst_node n3;
205     n3.len = 16;
206     n3.lchild = NULL;
```

```
207     n3.rchild = NULL;
208     n3.parent = NULL;
209     n3.data = NULL;
210     rtr_str_to_ipaddr("101.200.0.0", &(n3.prefix));
211     lpfst_insert(&n1, &n3, 0);

213     rtr_str_to_ipaddr("101.200.0.0", &(addr));
214     lvl = 0;
215     result = lpfst_lookup(&n1, &addr, 16, &lvl);
216     assert(result != NULL);
217     assert(rtr_cmp_ipv4(&(result->prefix), "101.200.0.0"));
218     assert(n1.lchild == &n3);

220     //n4
221     lpfst_node n4;
222     n4.len = 24;
223     n4.lchild = NULL;
224     n4.rchild = NULL;
225     n4.parent = NULL;
226     n4.data = NULL;
227     rtr_str_to_ipaddr("132.200.3.0", &(n4.prefix));
228     lpfst_insert(&n1, &n4, 0);

230     rtr_str_to_ipaddr("132.200.3.0", &(addr));
231     lvl = 0;
232     result = lpfst_lookup(&n1, &addr, 24, &lvl);
233     assert(result != NULL);
234     assert(rtr_cmp_ipv4(&(result->prefix), "132.200.3.0"));
235     assert(rtr_cmp_ipv4(&(n1.prefix), "132.200.3.0"));
236     assert(rtr_cmp_ipv4(&(n1.lchild->prefix), "101.200.0.0"));
237     assert(rtr_cmp_ipv4(&(n1.rchild->prefix), "132.200.0.0"));
238     assert(rtr_cmp_ipv4(&(n1.lchild->rchild->prefix),
        "100.200.0.0"));

240     rtr_str_to_ipaddr("132.200.0.0", &(addr));
241     lvl = 0;
242     result = lpfst_lookup(&n1, &addr, 16, &lvl);
243     assert(result != NULL);
244     assert(rtr_cmp_ipv4(&(result->prefix), "132.200.0.0"));

246     rtr_str_to_ipaddr("132.200.3.0", &(addr));
247     lvl = 0;
248     result = lpfst_lookup(&n1, &addr, 16, &lvl);
249     assert(result != NULL);
```

```
250     assert(rtr_cmp_ipv4(&(result->prefix), "132.200.0.0"));


253     lvl = 0;
254     rtr_str_to_ipaddr("132.0.0.0", &(addr));
255     bool found;
256     result = lpfst_lookup_exact(&n1, &addr, 16, &lvl, &found);
257     assert(!found);

259     lvl = 0;
260     rtr_str_to_ipaddr("132.200.3.0", &(addr));
261     result = lpfst_lookup_exact(&n1, &addr, 24, &lvl, &found);
262     assert(found);
263     assert(rtr_cmp_ipv4(&(result->prefix), "132.200.3.0"));

265     result = lpfst_remove(&n1, &addr, 0);
266     assert(rtr_cmp_ipv4(&(n1.prefix), "132.200.0.0"));
267     assert(rtr_cmp_ipv4(&(n1.lchild->prefix), "101.200.0.0"));
268     assert(rtr_cmp_ipv4(&(n1.lchild->rchild->prefix),
         "100.200.0.0"));
269     assert(n1.rchild == NULL);

271     rtr_str_to_ipaddr("101.200.0.0", &(addr));
272     result = lpfst_remove(&n1, &addr, 0);
273     assert(rtr_cmp_ipv4(&(n1.lchild->prefix), "100.200.0.0"));
274     assert(n1.rchild == NULL);

276     rtr_str_to_ipaddr("100.200.0.0", &(addr));
277     result = lpfst_remove(&n1, &addr, 0);
278     assert(n1.lchild == NULL);
279     assert(n1.rchild == NULL);
280 }


283 int main(){
284     get_bits_testv4();
285     get_bits_testv6();
286     lpfst_test();
287     printf("Test successfull\n");
288 }
```

Listing A.4: LPFST Unit-Test

```
1 #include <stdlib.h>
2 #include <stdio.h>
```

```
#include <sys/types.h>
#include <assert.h>
#include <string.h>
#include <stdbool.h>
#include <arpa/inet.h>
#include "rtrlib/lib/ip.h"
#include "rtrlib/lib/utils.h"
#include "rtrlib/pfx/lpfst/lpfst-pfx.h"
void print_bytes(void* buf, size_t len){
    for(unsigned int i = 0;i < len; i++){
        if(len != 0)
            printf(":");
        printf("%02x", *((uint8_t*) ((char*) buf + i)));
    }
    printf("\n");
}

void print_state(const pfxv_state s){
    if(s == BGP_PFXV_STATE_VALID)
        printf("VALID\n");
    else if(s == BGP_PFXV_STATE_NOT_FOUND)
        printf("NOT FOUND\n");
    else if(s == BGP_PFXV_STATE_INVALID)
        printf("INVALID\n");
}

void remove_src_test(){
    pfx_table pfxt;
    pfx_table_init(&pfxt, NULL);

    pfx_record pfx;
    pfx.min_len = 32;
    pfx.max_len = 32;

    pfx.asn = 80;
    pfx.socket_id = 1;
    rtr_str_to_ipaddr("10.11.10.0", &(pfx.prefix));
    assert(pfx_table_add(&pfxt, &pfx) == PFX_SUCCESS);

    pfx.asn = 90;
    pfx.socket_id = 2;
    rtr_str_to_ipaddr("10.11.10.0", &(pfx.prefix));
    assert(pfx_table_add(&pfxt, &pfx) == PFX_SUCCESS);
```

```
47      pfx.socket_id = 2;
48      pfx.min_len = 24;
49      rtr_str_to_ipaddr("192.168.0.0", &(pfx.prefix));
50      assert(pfx_table_add(&pfxt, &pfx) == PFX_SUCCESS);

52      pfx.socket_id = 1;
53      pfx.min_len = 8;
54      rtr_str_to_ipaddr("10.0.0.0", &(pfx.prefix));
55      assert(pfx_table_add(&pfxt, &pfx) == PFX_SUCCESS);

57      unsigned int len = 0;
58      lpfst_node** array = NULL;
59      assert(lpfst_get_children(pfxt.ipv4, &array, &len) != -1);
60      free(array);
61      array = NULL;
62      assert((len + 1) == 3);

64      pfx_table_src_remove(&pfxt, 1);
65      len=0;
66      assert(lpfst_get_children(pfxt.ipv4, &array, &len) != -1);

68      free(array);
69      assert((len + 1) == 2);

71      pfxv_state res;
72      assert(pfx_table_validate(&pfxt, 90, &(pfx.prefix), 8,
         &res) == PFX_SUCCESS);
73      assert(res == BGP_PFXV_STATE_NOT_FOUND);
74      rtr_str_to_ipaddr("10.11.10.0", &(pfx.prefix));

76      assert(pfx_table_validate(&pfxt, 90, &(pfx.prefix), 32,
         &res) == PFX_SUCCESS);
77      assert(res == BGP_PFXV_STATE_VALID);
78      assert(pfx_table_validate(&pfxt, 80, &(pfx.prefix), 32,
         &res) == PFX_SUCCESS);
79      assert(res == BGP_PFXV_STATE_INVALID);

81      printf("remove_src_test successfull\n");

83      pfx_table_free(&pfxt);
84  }

86  void mass_test(){
87      pfx_table pfxt;
```

```
88      pfx_table_init(&pfxt, NULL);

90      pfx_record rec;
91      pfxv_state res;
92      const uint32_t min_i = 0xFFFF0000;
93      const uint32_t max_i = 0xFFFFFFF0;

95      printf("Inserting %u records\n", (max_i - min_i) * 3);
96      for(uint32_t i = max_i; i >= min_i; i--){
97          rec.min_len = 32;
98          rec.max_len = 32;
99          rec.socket_id = i;
100         rec.asn = i;
101         rec.prefix.u.addr4.addr = htonl(i);
102         rec.prefix.ver = IPV4;
103         assert(pfx_table_add(&pfxt, &rec) == PFX_SUCCESS);
104         rec.asn = i + 1;
105         assert(pfx_table_add(&pfxt, &rec) == PFX_SUCCESS);

107         rec.min_len = 128;
108         rec.max_len = 128;
109         rec.prefix.ver = IPV6;
110         rec.prefix.u.addr6.addr[1] = min_i + 0xFFFFFFFF;
111         rec.prefix.u.addr6.addr[0] = htonl(i) + 0xFFFFFFFF;
112         assert(pfx_table_add(&pfxt, &rec) == PFX_SUCCESS);
113     }

115     printf("validating..\n");
116     for(uint32_t i = max_i; i >= min_i; i--){
117         rec.min_len = 32;
118         rec.max_len = 32;
119         rec.prefix.ver = IPV4;
120         rec.prefix.u.addr4.addr = htonl(i);
121         assert(pfx_table_validate(&pfxt, i, &(rec.prefix),
            rec.min_len, &res) == PFX_SUCCESS);
122         assert(res == BGP_PFXV_STATE_VALID);
123         assert(pfx_table_validate(&pfxt, i + 1, &(rec.prefix),
            rec.min_len, &res) == PFX_SUCCESS);
124         assert(res == BGP_PFXV_STATE_VALID);


127         rec.min_len = 128;
128         rec.max_len = 128;
129         rec.prefix.ver = IPV6;
```

```
130         rec.prefix.u.addr6.addr[1] = min_i + 0xFFFFFFFF;
131         rec.prefix.u.addr6.addr[0] = htonl(i) + 0xFFFFFFFF;

133         assert(pfx_table_validate(&pfxt, i + 1, &(rec.prefix),
             rec.min_len, &res) == PFX_SUCCESS);
134         assert(res == BGP_PFXV_STATE_VALID);
135     }

137     printf("removing records\n");
138     for(uint32_t i = max_i; i >= min_i; i--){
139         rec.socket_id = i;
140         rec.min_len = 32;
141         rec.max_len = 32;
142         rec.asn = i;
143         rec.prefix.ver = IPV4;
144         rec.prefix.u.addr4.addr = htonl(i);
145         assert(pfx_table_remove(&pfxt, &rec) == PFX_SUCCESS);

147         rec.asn = i + 1;
148         assert(pfx_table_remove(&pfxt, &rec) == PFX_SUCCESS);

150         rec.prefix.ver = IPV6;
151         rec.min_len = 128;
152         rec.max_len = 128;
153         rec.prefix.u.addr6.addr[1] = min_i + 0xFFFFFFFF;
154         rec.prefix.u.addr6.addr[0] = htonl(i) + 0xFFFFFFFF;
155         assert(pfx_table_remove(&pfxt, &rec) == PFX_SUCCESS);
156     }
157     pfx_table_free(&pfxt);
158     printf("Done\n");
159 }

161 int main(){
162     pfx_table pfxt;
163     pfx_table_init(&pfxt, NULL);

165     pfx_record pfx;
166     pfx.asn = 123;
167     pfx.prefix.ver = IPV4;
168     rtr_str_to_ipaddr("10.10.0.0", &(pfx.prefix));
169     pfx.min_len = 16;
170     pfx.max_len = 24;

172     assert(pfx_table_add(&pfxt, &pfx) == PFX_SUCCESS);
```

```
174     pfxv_state res;
175     assert(pfx_table_validate(&pfxt, 123, &(pfx.prefix), 16,
         &res) == PFX_SUCCESS);
176     assert(res == BGP_PFXV_STATE_VALID);
177     assert(pfx_table_validate(&pfxt, 124, &(pfx.prefix), 16,
         &res) == PFX_SUCCESS);
178     assert(res == BGP_PFXV_STATE_INVALID);

180     assert(pfx_table_validate(&pfxt, 123, &(pfx.prefix), 24,
         &res) == PFX_SUCCESS);
181     assert(res == BGP_PFXV_STATE_VALID);

183     rtr_str_to_ipaddr("10.10.10.0", &(pfx.prefix));
184     assert(pfx_table_validate(&pfxt, 123, &(pfx.prefix), 20,
         &res) == PFX_SUCCESS);
185     assert(res == BGP_PFXV_STATE_VALID);

187     assert(pfx_table_validate(&pfxt, 123, &(pfx.prefix), 25,
         &res) == PFX_SUCCESS);
188     assert(res == BGP_PFXV_STATE_INVALID);

190     rtr_str_to_ipaddr("10.11.10.0", &(pfx.prefix));
191     assert(pfx_table_validate(&pfxt, 123, &(pfx.prefix), 16,
         &res) == PFX_SUCCESS);
192     assert(res == BGP_PFXV_STATE_NOT_FOUND);

194     rtr_str_to_ipaddr("2a01:4f8:131::", &(pfx.prefix));
195     pfx.prefix.ver = IPV6;
196     pfx.min_len = 48;
197     pfx.max_len = 48;
198     pfx.asn = 124;
199     assert(pfx_table_add(&pfxt, &pfx) == PFX_SUCCESS);
200     assert(pfx_table_validate(&pfxt, 124, &(pfx.prefix), 48,
         &res) == PFX_SUCCESS);
201     assert(res == BGP_PFXV_STATE_VALID);

203     rtr_str_to_ipaddr("2a01:4f8:131:15::", &(pfx.prefix));
204     assert(pfx_table_validate(&pfxt, 124, &(pfx.prefix), 56,
         &res) == PFX_SUCCESS);
205     assert(res == BGP_PFXV_STATE_INVALID);

207     assert(rtr_str_to_ipaddr("1.0.4.0", &(pfx.prefix)) == 0);
208     pfx.min_len=22;
```

```
209     pfx.max_len=22;
210     pfx.asn=56203;
211     assert(pfx_table_add(&pfxt, &pfx) == PFX_SUCCESS);
212     assert(pfx_table_validate(&pfxt, pfx.asn, &(pfx.prefix),
         pfx.min_len, &res) == PFX_SUCCESS);
213     assert(res == BGP_PFXV_STATE_VALID);

215     assert(rtr_str_to_ipaddr("1.8.1.0", &(pfx.prefix)) == 0);
216     pfx.min_len=24;
217     pfx.max_len=24;
218     pfx.asn=38345;
219     assert(pfx_table_add(&pfxt, &pfx) == PFX_SUCCESS);
220     assert(pfx_table_validate(&pfxt, pfx.asn, &(pfx.prefix),
         pfx.min_len, &res) == PFX_SUCCESS);
221     assert(res == BGP_PFXV_STATE_VALID);

223     assert(rtr_str_to_ipaddr("1.8.8.0", &(pfx.prefix)) == 0);
224     pfx.min_len=24;
225     pfx.max_len=24;
226     pfx.asn=38345;
227     assert(pfx_table_add(&pfxt, &pfx) == PFX_SUCCESS);
228     assert(pfx_table_validate(&pfxt, pfx.asn, &(pfx.prefix),
         pfx.min_len, &res) == PFX_SUCCESS);
229     assert(res == BGP_PFXV_STATE_VALID);
230     pfx_table_free(&pfxt);

232     assert(rtr_str_to_ipaddr("1.0.65.0", &(pfx.prefix)) == 0);
233     pfx.min_len=18;
234     pfx.max_len=18;
235     pfx.asn=18144;
236     assert(pfx_table_add(&pfxt, &pfx) == PFX_SUCCESS);
237     assert(pfx_table_validate(&pfxt, pfx.asn, &(pfx.prefix),
         pfx.min_len, &res) == PFX_SUCCESS);
238     assert(res == BGP_PFXV_STATE_VALID);
239     pfx_table_free(&pfxt);

241     assert(rtr_str_to_ipaddr("10.0.0.0", &(pfx.prefix)) == 0);
242     pfx.min_len=16;
243     pfx.max_len=16;
244     pfx.asn=123;
245     assert(pfx_table_add(&pfxt, &pfx) == PFX_SUCCESS);
246     assert(pfx_table_validate(&pfxt, pfx.asn, &(pfx.prefix),
         pfx.min_len, &res) == PFX_SUCCESS);
247     assert(res == BGP_PFXV_STATE_VALID);
```

```
249     assert(rtr_str_to_ipaddr("10.0.5.0", &(pfx.prefix)) == 0);
250     pfx.min_len=24;
251     pfx.max_len=24;
252     pfx.asn=124;
253     assert(pfx_table_validate(&pfxt, pfx.asn, &(pfx.prefix),
        pfx.min_len, &res) == PFX_SUCCESS);
254     assert(res == BGP_PFXV_STATE_INVALID);

256     pfx_table_free(&pfxt);
257     assert(rtr_str_to_ipaddr("109.105.96.0", &(pfx.prefix)) ==
        0);
258     pfx.min_len=19;
259     pfx.max_len=19;
260     pfx.asn=123;
261     assert(pfx_table_add(&pfxt, &pfx) == PFX_SUCCESS);
262     assert(rtr_str_to_ipaddr("109.105.128.0", &(pfx.prefix))
        == 0);
263     assert(pfx_table_validate(&pfxt, 456, &(pfx.prefix), 20,
        &res) == PFX_SUCCESS);
264     assert(res == BGP_PFXV_STATE_NOT_FOUND);

266     pfx_table_free(&pfxt);
267     assert(rtr_str_to_ipaddr("190.57.224.0", &(pfx.prefix)) ==
        0);
268     pfx.min_len=19;
269     pfx.max_len=24;
270     pfx.asn=123;
271     assert(pfx_table_add(&pfxt, &pfx) == PFX_SUCCESS);
272     assert(rtr_str_to_ipaddr("190.57.72.0", &(pfx.prefix)) ==
        0);
273     assert(pfx_table_validate(&pfxt, 123, &(pfx.prefix), 21,
        &res) == PFX_SUCCESS);
274     assert(res == BGP_PFXV_STATE_NOT_FOUND);

276     pfx_table_free(&pfxt);

278     assert(rtr_str_to_ipaddr("80.253.128.0", &(pfx.prefix)) ==
        0);
279     pfx.min_len=19;
280     pfx.max_len=19;
281     pfx.asn=123;
282     assert(pfx_table_add(&pfxt, &pfx) == PFX_SUCCESS);
```

```
283     assert(rtr_str_to_ipaddr("80.253.144.0", &(pfx.prefix)) ==
        0);
284     assert(pfx_table_validate(&pfxt, 123, &(pfx.prefix), 20,
        &res) == PFX_SUCCESS);
285     assert(res == BGP_PFXV_STATE_INVALID);

287     char tmp[512];
288     ipv6_addr_to_str(&(pfx.prefix.u.addr6), tmp, sizeof(tmp));
289     printf("%s\n", tmp);

291     pfx_table_free(&pfxt);
292     remove_src_test();
293     mass_test();
294 }
```

Listing A.5: PFX Table Unit-Test

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <sys/types.h>
5  #include <string.h>
6  #include <stdbool.h>
7  #include <arpa/inet.h>
8  #include <pthread.h>
9  #include <time.h>
10 #include "rtrlib/lib/ip.h"
11 #include "rtrlib/pfx/lpfst/lpfst-pfx.h"

13 void print_state(const pfxv_state s){
14     if(s == BGP_PFXV_STATE_VALID)
15         printf("VALID\n");
16     else if(s == BGP_PFXV_STATE_NOT_FOUND)
17         printf("NOT FOUND\n");
18     else if(s == BGP_PFXV_STATE_INVALID)
19         printf("INVALID\n");
20 }

22 void print_pfx_rtval(const int rtval){
23     if(rtval == PFX_SUCCESS)
24         printf("PXF_SUCCESS\n");
25     else if(rtval == PFX_ERROR)
26         printf("PXF_ERROR\n");
27     else if(rtval == PFX_DUPLICATE_RECORD)
28         printf("PXF_DUPLICATE_RECORD\n");
```

```
29      else if(rtval == PFX_RECORD_NOT_FOUND)
30          printf("PXF_RECORD_NOT_FOUND\n");
31  }
33  uint32_t min_i = 0xFF000000;
34  uint32_t max_i = 0xFFFFFFF0;
36  void rec_insert(pfx_table* pfxt){
37      const int tid = getpid();
38      pfx_record rec;
39      rec.min_len = 32;
40      rec.max_len = 32;
41      rec.prefix.ver = IPV4;
42      rec.prefix.u.addr4.addr = 0;
43      pfxv_state res;
45      printf("Inserting %u records\n", (max_i - min_i) * 3);
46      for(uint32_t i = max_i; i >= min_i; i--){
47          rec.min_len = 32;
48          rec.max_len = 32;
49          rec.socket_id = i;
50          rec.asn = tid % 2;
51          rec.prefix.u.addr4.addr = htonl(i);
52          rec.prefix.ver = IPV4;
53          pfx_table_add(pfxt, &rec);
54          rec.asn = (tid % 2) + 1;
55          pfx_table_add(pfxt, &rec);
57          rec.min_len = 128;
58          rec.max_len = 128;
59          rec.prefix.ver = IPV6;
60          rec.prefix.u.addr6.addr[1] = min_i + 0xFFFFFFFF;
61          rec.prefix.u.addr6.addr[0] = htonl(i) + 0xFFFFFFFF;
62          pfx_table_add(pfxt, &rec);
63          usleep(rand() / (RAND_MAX / 20));
64      }
65  }
66  void rec_validate(pfx_table* pfxt){
67      const int tid = getpid();
68      pfx_record rec;
69      rec.min_len = 32;
70      rec.max_len = 32;
71      rec.prefix.ver = IPV4;
72      rec.prefix.u.addr4.addr = 0;
```

```
73      pfxv_state res;
74      printf("validating..\n");
75      for(uint32_t i = max_i; i >= min_i; i--){
76          rec.min_len = 32;
77          rec.max_len = 32;
78          rec.prefix.ver = IPV4;
79          rec.prefix.u.addr4.addr = htonl(i);
80          pfx_table_validate(pfxt, (tid % 2), &(rec.prefix),
             rec.min_len, &res);
82          pfx_table_validate(pfxt, (tid % 2) + 1, &(rec.prefix),
             rec.min_len, &res);
84          rec.min_len = 128;
85          rec.max_len = 128;
86          rec.prefix.ver = IPV6;
87          rec.prefix.u.addr6.addr[1] = min_i + 0xFFFFFFFF;
88          rec.prefix.u.addr6.addr[0] = htonl(i) + 0xFFFFFFFF;
90          pfx_table_validate(pfxt, (tid % 2) + 1, &(rec.prefix),
             rec.min_len, &res);
91          usleep(rand() / (RAND_MAX / 20));
92      }
93  }
94  void rec_remove(pfx_table* pfxt){
95      const int tid = getpid();
96      int rtval;
97      pfx_record rec;
98      rec.min_len = 32;
99      rec.max_len = 32;
100     rec.prefix.ver = IPV4;
101     rec.prefix.u.addr4.addr = 0;
102     printf("removing records\n");
103     for(uint32_t i = max_i; i >= min_i; i--){
104         rec.socket_id = i;
105         rec.min_len = 32;
106         rec.max_len = 32;
107         rec.asn = tid %2;
108         rec.prefix.ver = IPV4;
109         rec.prefix.u.addr4.addr = htonl(i);
110         rtval = pfx_table_remove(pfxt, &rec);
112         rec.asn = (tid % 2) + 1;
113         pfx_table_remove(pfxt, &rec);
```

```
115         rec.prefix.ver = IPV6;
116         rec.min_len = 128;
117         rec.max_len = 128;
118         rec.prefix.u.addr6.addr[1] = min_i + 0xFFFFFFFF;
119         rec.prefix.u.addr6.addr[0] = htonl(i) + 0xFFFFFFFF;
120         pfx_table_remove(pfxt, &rec);
121         usleep(rand() / (RAND_MAX / 20));
122     }
123     printf("Done\n");
124 }

126 int main(){
127     unsigned int max_threads = 15;
128     pfx_table pfxt;
129     pfx_table_init(&pfxt, NULL);
130     pthread_t threads[max_threads];
131     srand(time(NULL));
132     for(unsigned int i=0;i<max_threads;i++){
133         int r = rand() / (RAND_MAX / 3);
134         if(r == 0)
135             pthread_create(&(threads[i]), NULL, (void *
                (*)(void *)) rec_insert, &pfxt);
136         else if(r == 1)
137                 pthread_create(&(threads[i]), NULL, (void *
                    (*)(void *)) rec_remove, &pfxt);
138         else if(r == 2)
139             pthread_create(&(threads[i]), NULL, (void *
                (*)(void *)) rec_validate, &pfxt);
140         printf("Started Thread %d\n",i);
141         usleep(200);

143     }
144     for(unsigned int i=0;i<max_threads;i++){
145         pthread_join(threads[i], NULL);
146         printf("Thread %i returned\n", i);
147     }
148 }
```

Listing A.6: PFX Table Unit-Test

A.4. Konfigurationsdateien

```
service 42420
{
        type = UNLISTED
        disable         = no
        server = /usr/local/bin/rtr-origin
        server_args = --server /var/rcynic
        port = 42420
        socket_type     = stream
        protocol        = tcp
        user            = root
        wait            = no
}
```

Listing A.7: rtr-origin xinetd Konfiguration

```
Port 22
Protocol 2
HostKey /etc/ssh/ssh_host_rsa_key
HostKey /etc/ssh/ssh_host_dsa_key
UsePrivilegeSeparation yes
KeyRegenerationInterval 3600
ServerKeyBits 768
LoginGraceTime 120
StrictModes yes
RSAAuthentication yes
IgnoreRhosts yes
RhostsRSAAuthentication no
HostbasedAuthentication no
UsePAM yes

Subsystem rpki-rtr /usr/local/bin/rtr-origin --server
 /var/rcynic/
```

Listing A.8: rtr-origin SSH-Subsystem Konfiguration

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 9. November 2011
Ort, Datum

Unterschrift