HAW HAMBURG

Masterarbeit

Colin Sames

Implementation and Evaluation of BGPsec for the FRRouting Suite

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Colin Sames

Implementation and Evaluation of BGPsec for the FRRouting Suite

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang *Master of Science Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas C. Schmidt
Zweitgutachter: Prof. Dr. Franz Korf

Eingereicht am: August 11, 2021

Colin Sames

Thema der Arbeit

Implementierung und Evaluation von BGPsec für die FRRouting Suite

Stichworte

BGP, BGPsec, Internetsicherheit, Leistung, Evaluierung

Kurzzusammenfassung

Das Internet stützt sich auf BGP, jedoch ist dieses Protokoll verwundbar für Angriffe. Informationen wie die Quelle von annoncierten IP-Präfixen oder ihr genommener Pfad, können von Dritten übernommen oder verändert werden. Dies kann sich negativ auf den Datenfluss eines Netzwerkes auswirken. Um diese Informationen zu authetifizieren gibt es zwei Ansätze: Route Origin Validierung und AS Pfad Validierung. Während Route Origin Validierung bereits erfolgreich verwendet wird, gibt es für AS Pfad Validierung, in Form von BGPsec, noch kaum Implementierungen.
In dieser Arbeit wird eine zweiteilige BGPsec Implementierung bereitgestellt. Ein Teil findet in der Open-Source Bibliothek RTRlib statt, der andere Teil wird in die Routingsoftware FRR integiert, welche ebenfalls Open-Source ist. Auf diese Weise bleiben RTRlib und FRR unabhängig voneinander und können leichter ausgetauscht werden, falls nötig. Zusätzlich zur Implementation wird die Implementierung evaluiert und mit der Referenzimplementierung, BGP-SRx, verglichen.
Die Richtigkeit der Implementierung ist durch das Kommunizieren mit der Referenzimplementierung bestätigt, indem in verschiedenen Szenarien gültige und fehlerhafte Nachrichten ausgetauscht wurden. Resultate der Leistungsmessung zeigen, dass BGPsec-Datenstrukturen in BGP-SRx 3 bis 8x so viel Speicher verbrauchen wie in FRR. Validierung in BGP-SRx hingegen verläuft 2x so schnell wie in FRR, signieren ist 3x so schnell. Die Skalierung verhält sich wie erwartet linear für Speicherverbrauch, Validierungen und Signaturen. Verglichen zu normalem BGP, müssen BGPsec Updates zudem an jeden Kommunikationspartner und für jedes IP-Präfix einzeln gesendet werden. Dies wirkt sich zusätzlich negativ auf die Leistung aus.
Mit dem Bereitstellen einer BGPsec Implementierung für eine aufstrebende Routingsoftware erhöhen sich die Chancen, dass BGPsec in Zukunft ausgerollt wird.

Colin Sames

Title of Thesis

Implementation and Evaluation of BGPsec for the FRRouting Suite

Keywords

BGP, BGPsec, Intenet security, Performance, Evaluation

Abstract

The Internet relies heavily on BGP, however this protocol is vulnerable to attacks. Information such as the announced IP-prefix or the path the announcement took can be hijacked or manipulated by third parties. This can have a negative impact on the data traffic of a network. To authenticate these information there are two approaches: route origin validation and AS path validation. While route origin validation was successfully deployed, there are still hardly any implementations for AS path validation in the form of BGPsec.

This work aims to provide a twofold BGPsec implementation. One part of it is integrated into the open-source library RTRlib, the other part is integrated into the routing suite FRR, whick is likewise open-source. This way, both RTRlib and FRR stay independent and can exchange one another, if needed. In conjunction with the implementation, its performance is evaluated and compared to the reference implementation, BGP-SRx.

The correctness of the implementation is verified in various scenarios by peering with BGP-SRx and processing valid and malformed messages. Performance results show that BGPsec structures in BGP-SRx take 3 to 8x the memory compared to FRR. However, validation in BGP-SRx performs 2x faster, signing even 3x faster. As expected, it scales linear for memory consumption and validation and signing time. Compared to BGP, BGPsec updates must be send individually for each peer and each prefix. This imposes another negative impact on the performance.

Nevertheless, having provided another BGPsec implementation to a thriving routing suite chances increase that BGPsec receives deployment in the future.

Contents

List of Figures viii

List of Tables xii

Acronyms xiii

1 Introduction 1

2 Inter-domain Routing with BGP 3
- 2.1 Autonomous Systems . . . 3
- 2.2 Border Gateway Protocol . . . 3
  - 2.2.1 BGP Specification . . . 4
  - 2.2.2 Attack Surface . . . 5

3 Protecting BGP 8
- 3.1 RPKI . . . 8
  - 3.1.1 Route Origin Validation . . . 8
  - 3.1.2 The RTR Protocol . . . 9
- 3.2 RPKI Implementations . . . 9
  - 3.2.1 RTRlib . . . 10
  - 3.2.2 FRR . . . 10
  - 3.2.3 BIRD . . . 11
  - 3.2.4 RIPE-NCC RPKI Validator 3 . . . 11
  - 3.2.5 Routinator 3000 . . . 11
- 3.3 Related Work . . . 11

4 BGPsec 13
- 4.1 AS Path Validation . . . 13
- 4.2 BGPsec Specification . . . 14
- 4.3 Deployment of BGPsec . . . 19

4.4 Related Work 20

5 Problem Statement 22

6 BGPsec Implementation 23

6.1 Validation Suite Design 23
6.1.1 Features 25
6.1.2 Crypto Library 26
6.1.3 AS Path Validation API 27
6.1.4 Project Structure 34
6.2 Validation Suite Implementation 35
6.2.1 Streams 35
6.2.2 Data Hashing 38
6.2.3 Signature Validation 40
6.2.4 Signature Generation 42
6.2.5 Revisiting Validation Suite Requirements 42
6.3 Protocol Handler Design 43
6.3.1 Choice for FRR 43
6.3.2 Design 44
6.3.3 The Module and Configuration 45
6.4 Protocol Handler Implementation 47
6.4.1 Commands 47
6.4.2 Capability Negotiation 49
6.4.3 Storing BGPsec Data 51
6.4.4 BGPsec Update Generation 52
6.4.5 BGPsec Update Parsing 53
6.4.6 Revisiting Protocol Handler Requirements 55

7 Performance Analysis 56

7.1 General 56
7.2 Tooling 56
7.2.1 Public/Private Key Generator 57
7.2.2 SPKI Cache 57
7.2.3 Dummy Router 58
7.3 Correctness of the Implementation 58
7.3.1 Open Message 59
7.3.2 Malformed Update Message 63

7.3.3 Parsing Update Message 65
7.3.4 Update Reconstruction 67
7.4 Memory Usage Performance 69
7.4.1 Router Keys 70
7.4.2 BGPsec Paths 73
7.5 Crypto Microbenchmarks 81
7.6 Processing Time Performance 86
7.6.1 Outgoing Updates 88
7.6.2 Incoming Updates 90

8 Conclusion and Outlook **93**

Bibliography **95**

A Appendix **102**

Selbstständigkeitserklärung **106**

List of Figures

3.1 Example work flow of a router fetching data from a cache server [1] 10

4.1 Capability as defined by the BGPsec specification [2]. The *Version* is currently 0. The direction bit, *Dir*, indicates whether a router can receive (0) or send (1) BGPsec updates. The *AFI* is either IPv4 or IPv6. *Unassigned* bits are currently not used. 15
4.2 Structure of the BGPsec PATH attribute in an UML like fashion. Both the Secure Path as well as the Signature Block may contain multiple Secure Path Segments and Signature Segments. 15
4.3 Signing chain from AS 1 up to AS 4. AS 1 signs the target AS, the BGPsec PATH and the prefix. AS 2 updates the target AS to match AS 3, appends its own AS to the path and signs everything, including the previous signature. AS 3 repeats the process. 17
4.4 AS 4 validates the BGPsec update it received from AS 3 (see Figure 4.3). AS 4 uses the public keys of AS 1, 2 and 3 to verify the signatures they generated. The order of execution starts with the last prepended signature and ends with the first. 17
4.5 Iterative process of update generation. The originating AS generates an update (inner box) by 1) hashing the relevant information, then 2) signing the resulting digest value and appending the generated signature. The next forwarding AS (outer box) repeats the process by 3) hashing the previous data plus its own and then 4) generates and appends the signature. 18
4.6 The BGPsec update on the left is processed top down until all signatures are verified. If one result is invalid, the process ends prematurely. The router keys are fetched from the RPKI cache servers. 19

6.1 The bgpsec struct holds references to the signature and secure path lists. They always point to the head of it. The list elements themselves hold references to the next element of the list. In this example, the AS path is (3 2 1). 29
6.2 (a) shows the BGPsec struct that holds a reference to the head of the secure path list, AS 1. When prepending a new segment, AS 2, the references get updated accordingly (b). AS 2 is now the head of the list. . 32
6.3 Truncated code of the validation for-loop. It generates a digest value with each iteration. With each iteration, the offset shifts to the next starting point. 39
6.4 Visual representation of the offset shifting code in Listing 6.3. The bytes are aligned from left to right as a stream. 40

7.1 Testing open messages. In a), malformed BGPsec capabilities are sent from a Dummy Router instance to FRR. In b), proper open capabilities are exchanged between FRR and BGP-SRx. These capabilities contain various combinations of direction and AFI. 59
7.2 Wireshark capture of a BGP session. The update include the AS PATH attribute, hence it must be a plain BGP update. 62
7.3 Wireshark capture of a BGPsec session. The BGPsec PATH attribute is included within the update. 62
7.4 Dummy Router sends malformed BGPsec updates to FRR. 63
7.5 BGPsec update correctness test. In a), the signing routine is evaluated. BGP-SRx 1 and 2 must successfully validate the BGPsec update forwarded by FRR N. Scenario b) evaluates validation. A BGPsec update forwarded by BGP-SRx N must be successfully validated, this time by FRR 1 and 2. 65
7.6 Test scenario for BGPsec update reconstruction. Because FRR N does not speak BGPsec, FRR N-1 needs to reconstruct a plain BGP update from the BGPsec update. 68
7.7 Memory usage of router keys for RTRlib and BGP-SRx. Between 100 and 1,000 router keys have been examined, all are of same size. 72
7.8 Left: total memory usage of FRR with 1,000 stored router keys. Router keys occupy 2% of it. Right: same scenario using BGP-SRx. 48.6% of the total memory is occupied by the 1,000 router keys. 73

7.9 Setup for testing memory usage of FRR. To test memory usage for AS paths of length N, there need to be N+1 FRR instances chained together. After validating all updates, the memory usage on every instance is checked via the *show memory* command on the appropriate daemon. . . 76
7.10 Estimated and real memory usage of stored BGPsec paths in FRR. Between 500 and 5,000 BGPsec paths of length 1 to 5 were processed. Estimate (a) and measurements (b) are almost identical. The initial spike in (b) is attributed to BGPsec initialization. 77
7.11 Estimated and real memory usage of stored BGPsec paths in BGP-SRx. Between 500 and 5,000 BGPsec paths of various lengths were processed. Estimate (a) is almost identical to test result (b). 78
7.12 Comparison of memory values for storing between 500 and 5,000 BGPsec paths of length 1 to 5. Although FRR has a higher base value of occupied memory, its memory consumption per path is lower compared to BGP-SRx 79
7.13 Left: memory usage of FRR storing 5,000 BGPsec paths of length 5. The BGPsec paths occupy 21.6% of the total memory, including router keys. Right: same scenario with BGP-SRx. Here, the BGPsec paths occupy 88.1% of the total memory in use. 79
7.14 Total memory usage of FRR and BGP-SRx running plain BGP. Between 500 and 5,000 BGP paths of various lengths were processed. Path length in this test has to visible impact on the memory consumption. 80
7.15 Microbenchmark of the OpenSSL validation (a) and signing (b) routines. The two OpenSSL version 1.0 and 1.1 were examined. Both routines were executed 10 times in succession with a hash representing a path length of 1 (Len 1) and a path length of 2 (Len 2). The length of the BGPsec path has no impact on the duration since the path is hashed to a fixed length of 256 bytes. The initial spike comes from OpenSSL initialization. The version has to visible impact on the execution time. 83
7.16 Average duration for generating BGPsec updates, 5,000 in total. The average slightly increases with growing path length. FRR measurements are split in signing and key loading. Due to repetitive loading of router keys in FRR, a constant overhead of 77 µs is added on top. 84
7.17 Average duration for validating a single BGPsec update. 5,000 BGPsec updates per path length were processed. Results for FRR are split into the validation and key loading part to visualize the overhead. 85

7.18 Left: FRR validation time in comparison to the whole update parsing routine. 99% of the time parsing the BGPsec update is spent verifying the signatures. Right: Comparing the generation of an entire BGPsec update to the time it takes to generate a signature. 94% of the time is spent generating the signature. 86
7.19 Setup for measuring validation and signing processing time. It is identical to the one for memory usage. 87
7.20 Measurement of BGPsec update generation. Processing time slightly increases with growing path length, as microbenchmarks have shown. 89
7.21 Update generation of plain BGP updates. Only one update is generated per hop as BGP updates can contain multiple prefixes at once. 90
7.22 Parsing and validating up to 5,000 BGPsec updates. 91
7.23 Update parsing of plain BGP updates. Despite the growing path length and amount of prefixes, only a single update is generated. FRR and BGP-SRx show no observable difference in performance for varying path length. 92

List of Tables

7.1 Memory required for storing a single router key in RTRlib and BGP-SRx. The minimum is 115 bytes. RTRlib allocates an additional 100 bytes per key, while BGP-SRx requires 1,744 bytes of extra memory per key. 71
7.2 Memory overhead for BGPsec paths of FRR and BGPsec. For reference, RFC values for BGPsec path size of 109 bytes base value and 100 bytes per hop are used [3]. 75

Acronyms

Adj-RIB-in Adjacent-RIB-Incoming.

Adj-RIB-out Adjacent-RIB-Outgoing.

AFI Address Family Identifier.

API Application Programming Interface.

AS Autonomous System.

ASN Autonomous System Number.

BGP Border Gateway Protocol.

DoS Denial of Service.

ECDSA Elliptic Curve Digital Signature Algorithm.

FFI Foreign Function Interface.

FIB Forwarding Information Base.

IANA Internet Assigned Numbers Authority.

IETF Internet Engineering Task Force.

LIFO last in, first out.

Loc-RIB Local RIB.

NIST National Institute of Standards and Technology.

NLRI Network Layer Reachability Information.

PKI Public Key Infrastructure.

RIB Routing Information Base.

RIR Regional Internet Registry.

ROA Route Origin Authorization.

ROV Route Origin Validation.

RP Relying Party.

RPKI Resource Public Key Infrastructure.

RTR RPKI to Router.

SAFI Subsequent Address Family Identifier.

SHA-256 Secure Hash Algorithm 2 (256 bits).

SKI Subject Key Identifier.

SPKI Subject Public Key Information.

1 Introduction

Back in the days when the Internet was still young, some attacks that disrupt Internet traffic had yet to surface. When they eventually did, preventing Internet outages and espionage became more and more relevant. The Internet was formed as a large network made of smaller networks. A network within the Internet, called Autonomous System (AS), exchanges reachability information with other ASes using the Border Gateway Protocol (BGP) [4]. Its latest version was standardized by the Internet Engineering Task Force (IETF) in 2006. BGP announcements carry information such as *which AS originates which address space* and *which path the announcement traversed*. The former are called *origin AS* and *prefix*, the latter is the *AS path*.

Both information can, if manipulated, cause Internet outages for ASes or espionage [5]. Two protection mechanisms detect invalid announcements - whether they originate in misconfiguration or malicious intent. The Resource Public Key Infrastructure (RPKI) [6] allows ASes to verify the origin AS of a BGP announcement utilizing Route Origin Validation (ROV) [7] and AS path validation in the form of BGPsec [2]. With ROV, cryptographic certificates attest ownership of IP address space. They can be used by other ASes to verify such claims. With BGPsec, ASes hash and sign parts of an announcement. These signatures can be validated by receiving parties.

Although various ROV implementations exist, there are only few for BGPsec, such as the reference implementation of the corresponding RFC. However, this implementation is part of a routing suite that is hardly receiving updates. This work aims to achieve two goals: to 1) provide a BGPsec implementation for a different, widely used open-source routing suite and to 2) evaluate this implementation for correctness and to benchmark its performance.

The implementation itself is twofold. The first part, which will cover validating and signing AS paths, takes place in *RTRlib* [8]. RTRlib is an open-source library that provides the RPKI to Router (RTR) protocol [1], which is necessary to fetch cryptographic

keys which are in turn required for BGPsec. The second part of the implementation, i.e., protocol handling, occurs in the *FRR routing suite*. This routing suite provides a BGP implementation, amongst other protocols.

RTRlib already offers all necessary features to implement AS path validation and signing routines. Since FRR already depends on RTRlib for ROV, it is convenient to enable AS path validation via the same library. By having the cryptographic routines in RTRlib, code duplication in FRR is avoided. Additionally, other software can include RTRlib to make use of its AS path validation features this way. For this reason, the implementation was separated.

This work is structured as follows. Chapter 2 gives an introduction to Internet routing and BGP. Chapter 3 shows the current protection mechanisms for BGP. BGPsec is introduced separately in Chapter 4 and is discussed in depth. The problem statement of this work is presented in Chapter 5. Chapter 6 describes the implementation process of BGPsec. The implementation is analyzed and benchmarked in Chapter 7. Chapter 8 concludes this work.

2 Inter-domain Routing with BGP

The Internet is not a single network in which everyone is directly connected with each other, but instead it is a large network of smaller networks. Such a network is called Autonomous System (AS). ASes exchange reachability information with each other, called control plane. Along the control plane, data traffic is forwarded through the Internet, the data plane.

2.1 Autonomous Systems

An AS consists of one or more networks itself which are governed by a single administered domain. To be accessible, ASes hold one or more *IP prefixes*, where each prefix is maintained by one AS only. The assignment of IP prefixes originates from the Internet Assigned Numbers Authority (IANA)[1]. This institution passes IP prefixes down to one of five institutions, a Regional Internet Registry (RIR) [9]. From there, IP prefixes and unique identifiers called Autonomous System Number (ASN) [10, 11] are distributed to ASes all over the world. ASNs are used to identify an AS anywhere on the Internet. Each RIR covers a certain region on the globe, e.g., the European region falls under the responsibility of *RIPE NCC*[2].

2.2 Border Gateway Protocol

An AS controls routers that are dedicated to running the path-vector routing protocol BGP [4]. Path-vector protocols track the path of an announcement through networks [12]. With BGP, ASes exchange reachability information with other ASes. Since two ASes are not necessarily directly connected to each other, they depend on intermediate

[1] https://www.iana.org/
[2] https://www.ripe.net/

ASes to forward the announcements to ASes they otherwise would not be able to reach. Such announcements contain information about *which IP prefix an AS announced* and *which path the announcement took.*

2.2.1 BGP Specification

Each BGP message consists of a header and a message content. There are four types of BGP messages: *open*, *update*, *keepalive* and *notification* [4, 13]. The open message contains information about configuration and capabilities of the sending router. The update message contains routing information such as prefix announcements or withdrawals. Keepalive messages are used to maintain the connection with a peering router. The notification message informs a peering router about any errors that occur during a session.

While open, keepalive and notification messages are for establishing and maintaining a BGP session, the heart of BGP is the update message. It contains information such as IP prefixes and the AS path. The AS path is a sequence of ASNs that indicates, along which path the update message was forwarded through the Internet. The origin AS, i.e., the AS that claims ownership of the announced IP prefix, is always found at the rightmost position of the AS path. In literature, the AS path is often written as *(30 20 10)*, with AS 10 being the origin AS. The AS path length is the amount of elements in the path, which is 3 in this case.

BGP routers announce their own prefixes or withdrawal them. Routes to other prefixes are stored within one of three BGP tables, the Routing Information Base (RIB) [14]. All incoming routes enter the Adjacent-RIB-Incoming (Adj-RIB-in) table. Based on policies and rules, the router then decides which routes to process further. Selected routes are stored within the Local RIB (Loc-RIB). Those routes that are selected for propagation to neighboring peers end up in the Adjacent-RIB-Outgoing (Adj-RIB-out) table.

There is a set of rules defined in Section 9.1 of the BGP RFC [15] which is applied to routes that are stored within the Adj-RIB-in table. Local policies take the highest precedence. After they have been applied and the route has not been rejected, the next rule, the AS path length, is applied. Shorter paths are preferred over longer paths. In case the path length comparison yields no decision, the rest of the tie-breaking rules are applied. The decision process is necessary to, e.g., choose which routes to put into the Adj-RIB-out, i.e., which routes to advertise to the routers peers.

Using the BGP RIB as input, the routers forwarding table, the Forwarding Information Base (FIB), is formed. This table is used when forwarding traffic.

Routers forward traffic based on longest common prefix matching. As an example, a router wants to forward a packet with destination IP address *1.2.3.4*. Having to choose a forwarding path between the prefixes *1.2.3.0/20* and *1.2.3.0/24*, the router would decide to forward via the latter [5].

The next section shows how the prefix and the AS path information within BGP updates can be targeted to manipulate the traffic flow between ASes.

2.2.2 Attack Surface

Manipulating the origin of a prefix or the AS path of a BGP update may lead to wrongly forwarded traffic. Although propagating wrong information is not always of malicious intent, it still causes undesirable effects. Either way, the outcome usually is manipulated packet forwarding that can lead to unreachable Internet services, online fraud or espionage. This section covers the most common attacks that can be launched against BGP.

To manipulate traffic forwarding on the Internet, common techniques exist [5]. Depending on the technique and the goal of an attacker, a new BGP update needs to be forged and propagated, or a received update is tampered with and then forwarded to other ASes. The targeted information are either the association of origin AS and prefix, or the AS path. An attacker tries to either announce themselves as the owner of a prefix of another AS (prefix hijacking), or to redirect traffic forwarding via the own AS (path hijacking) [16].

To hijack a prefix, it may be sufficient for an AS to originate the prefix of another AS. Receiving routers of this update can not tell that this update contains a false claim and might accept it. Any router that accepts such an announcement now holds an incorrect association between IP prefix and origin AS. This means that any traffic that was supposed to reach the original owner is instead directed towards the hijacker.

Another way to manipulate traffic is for an attacking AS to hijack the AS path [16]. An AS path can be shortened, inflated or elements can be exchanged. If the AS path to a prefix is artificially shortened, it is more attractive to receiving routers. Since a router has no way of telling, whether or not such path actually exists, it would blindly accept

it (assuming it holds no entry or an entry with a longer path to that prefix). Similarly, if the AS path within an announcement is inflated, it might be rejected by a receiving router (only if an entry with a shorter path to the announced prefix exists within its routing table). Path inflation, however, is a also a common, harmless practice among autonomous systems [17]. Another way to make an AS reject a route is by adding the victims ASN to the AS path. At the victims end, the BGP loop detection algorithm automatically filters out paths that already contain the own ASN.

Any of these methods may be used with malicious intent, but are also often cause of misconfigured routers. Because of this, directed attacks and misconfigurations are hard to distinguish. Nevertheless, it is possible to cause trouble for other ASes by, e.g., making them unreachable [18, 19].

When an AS successfully gains control over the prefix or AS path of another AS, there are three malicious ways to deal with the hijacked traffic [20].

The first is *Blackholing*. The attacking AS drops all hijacked traffic, instead of forwarding it. As a result, the hijacked AS is not reachable for potentially a lot of Internet users. This is a Denial of Service (DoS) [21] attack on the victim AS.

The second option is *Interception*. When intercepting traffic, the hijacker AS proceeds to forward the traffic to its original destination. This way, the attack is hard to discover since there are hardly any anomalies for the victim to notice. Such an attack must be carefully planned though. The hijacker AS still requires an intact path to the victim AS, otherwise the forwarded traffic would not reach the target. If maintaining a path to the victim is accomplished, the traffic is reaching its target, although it is forwarded via a different path. This attack can be used to eavesdrop on the traffic of Internet users.

The last of the three options is *Impersonating*. The attacker responds to the traffic that was hijacked. An Internet user that is responded to by the attacker might not notice any difference. The received responses appear to come from the Internet service within the AS where the requests were initially sent to. This way, an attacker can, e.g., mimic a website and trick users to reveal sensitive personal information such as credit card numbers [5].

There are numerous examples of BGP attacks all over the world. It is impossible to create a complete list of all attacks as some go unnoticed or only last for a few minutes [22].

There are some infamous BGP attacks that have been made public, though. One well-known attack is the *YouTube* hijack from 2008 [23]. On the 24th of February, the AS *Pakistan Telekom* claimed ownership of an IP prefix belonging to the video hosting service. The announcement spread across the Internet and *YouTube* was left unaccessible for about two hours.

Another, more recent case happened on April 26 in 2017. The AS *Rostelecom* announced 50 prefixes of other autonomous systems [24]. Among the victims were ASes like *Visa* and *MasterCard*. The whole incident lasted only a couple of minutes and it is unclear whether or not it was done on purpose.

More well-known incidents are the *AS 7007 incident* in 1997 [25], allegedly caused by misconfiguration, and *Googles May 2005 Outage* [26].

3 Protecting BGP

The two most important pieces of information within a BGP update are the prefix and its origin AS as well as the AS path. Both are not protected by BGP and must therefore be secured with external mechanisms. Such mechanisms exist for both the prefix/origin AS and the AS path. They are described in this section.

3.1 RPKI

The RPKI [6] is dedicated to protecting both the prefix/origin AS and the AS path. The RPKI is an effort to have an infrastructure in which verified cryptographic information can be managed and accessed. These information can be used to validate propagated prefix/origin AS and AS path information. The RIRs manage the cryptographic information. Cache servers obtain and store these information for any device to fetch.

3.1.1 Route Origin Validation

A router can perform ROV [7] to validate the origin AS of a BGP update. For this, the router relies on the RPKI. The RPKI holds cryptographically signed objects that contain four information: 1) a prefix, 2) its length, 3) a maximum defined prefix length and 4) the AS that holds the prefix, i.e., the origin AS. These objects are called Route Origin Authorization (ROA) [27]. They are created by either a RIR or an entity that holds IP address space and are pre-validated by the cache servers. By pre-validating the ROAs, this load is taken off the router.

When a router receives a BGP update, it can validate the contained origin AS-prefix-association. This is done by taking the prefix-length-origin data from the update and comparing it with the data contained in the ROA. There are three outcomes to this validation [19].

1. The announcement is *valid* if it is covered by at least one ROA
2. The announcement is *invalid* if it is announced by an unauthorized AS (i.e., the AS does not match the ROA entry for the announced prefix) or is more specific (the announced prefix is more specific than the prefix of a matching ROA)
3. The ROA is *not found* and the status hence unknown

Based on this validation, a router may decide how to proceed with the BGP update. It is not mandatory to accept a valid announcement, nor is it mandatory to deny an invalid one. This is completely up to the ASes routing policy [6, 19, 28].

For a router to fetch the ROA information, it needs to be able to speak the protocol which is presented next.

3.1.2 The RTR Protocol

Data between a router and a cache server is exchanged via the RTR protocol [1]. Based on TCP, different types of messages are exchanged between both parties. A typical session, consisting of connection, data exchange and termination, is depicted in Figure 3.1.

The *Payload PDUs* contain ROA information and router keys. Router keys are used to protect the AS path.

3.2 RPKI Implementations

An excerpt of software that offers RPKI functionalities, such as RTR protocol and ROV implementations, is given in the following sections. A broader list can be found in [29].

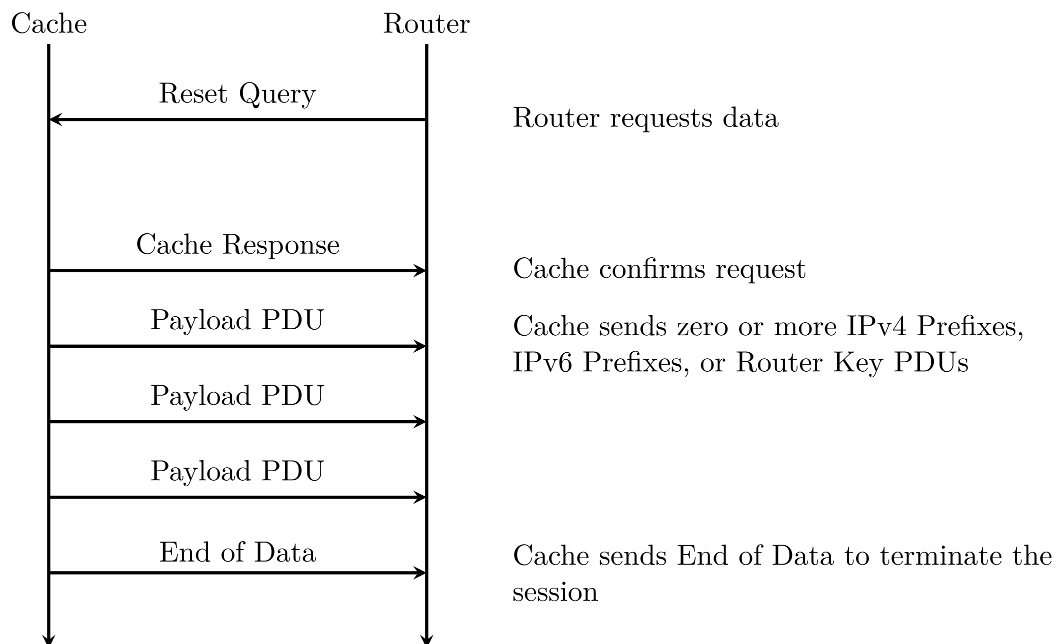


Figure 3.1: Example work flow of a router fetching data from a cache server [1]

3.2.1 RTRlib

RTRlib [8] is an open-source library[1] that implements the current version[2] of the client side RTR protocol and additionally offers ROV. By including this library, any application can make use of ROV to validate IP prefixes. There is, e.g., a REST interface implementation[3] that can be used for web based applications.

3.2.2 FRR

FRR[4] is a routing suite that supports an array of protocols for inter- and intra-domain routing. AS operators use FRR to handle routing for their AS and apply policies to filter routes. The suite supports ROV by including RTRlib. If configured appropriately, a router can apply route filtering based on the validity state of an announcement.

[1] https://github.com/rtrlib/rtrlib
[2] As of August 11, 2021, the current version is 1.
[3] https://github.com/rtrlib/rbv
[4] https://frrouting.org/

3.2.3 BIRD

BIRD is another routing suite that offers similar functionalities to FRR. Using parts of RTRlibs code base, the suite also offers RTR and ROV. However, only RTR version 0 [30] is supported. This is enough to perform ROV but crucial components for AS path validation are missing.

3.2.4 RIPE-NCC RPKI Validator 3

The European RIR RIPE-NCC develops the RPKI Validator 3 that supports a variety of RPKI features[5]. It provides server and client site deployment and offers ROV via different interfaces. Since it pulls in no other RPKI libraries such as RTRlib, it is completely independent from other RPKI tools. The software is open-source and hosted on GitHub[6].

3.2.5 Routinator 3000

The Routinator 3000 is an RPKI Validator, just like the RIPE-NCC Validator. It is developed by NLnet Labs[7] and is designed to be more lightweight, in contrast to, e.g., the RPKI Validator 3.

3.3 Related Work

RPKI deployment is supported by research with focus on measurements, security and deployment.

Wählisch *et al.* [31] and Iamartino *et al.* [32] measure the RPKI coverage, i.e., how many ROAs are covering IP prefixes. Institutions like the National Institute of Standards and Technology (NIST) and RIPE NCC provide monitoring applications that display global RPKI coverage information [33, 34]. In the year 2012, 2% of IP prefixes were covered by ROAs, as shown by Wählisch *et al.* [19]. Seven years later, in 2019, Chung *et al.* observed a coverage of 12.1% [35].

[5] https://github.com/RIPE-NCC/rpki-validator-3/wiki
[6] https://github.com/RIPE-NCC/rpki-validator-3
[7] https://nlnetlabs.nl/

The reason *why* the deployment progresses so slowly is examined by Goldberg [36] and Gilad *et al.* [37]. They conclude that ASes are hesitant to deploy RPKI for reasons of allegedly small security benefits, computational overhead and potential loss of traffic.

Not all ASes that perform ROV apply route filtering, though. Reuter *et al.* provide a methodology for investigating, whether or not routes are filtered by ASes based on ROV [38].

Research from 2020 reveals that issues with RPKI are not limited to how ASes adopt it. Kristoff *et al.* show in [39] that Relying Party (RP) cache servers are often unable to retrieve the cryptographic objects they need to verify. This leads to the distribution of incomplete data to BGP routers.

The next section covers BGPsec, which implements AS path protection.

4 BGPsec

To protect the integrity of the AS path, BGP routers need to have a way of detecting manipulations. The first design for AS path protection dates back to the year 2000. Stephen Kent *et al.* designed a BGP security method called Secure BGP (S-BGP) [40]. The IETF picked up on this idea and worked out a standard which eventually became BGPsec.

4.1 AS Path Validation

The general idea of AS path validation is the usage of asymmetric cryptography [41] in combination with a Public Key Infrastructure (PKI), in this case the RPKI. BGP routers hold private keys which allow them to sign the contents of BGP updates. Using the public keys, which are located within the RPKI, other routers are able to verify these signatures.

In more detail, the procedure works as follows: A router takes all the information of a BGP update that are to be secured and applies a digest function to them. The resulting digest value is then signed using the routers private key. This cryptographic signature is in turn appended to the update message. Receiving routers can verify the signature using the appropriate public key. The process is further described in section 4.2. The protection lies in the properties of cryptographic signatures.

Authentication Proves ownership of the private key. If a signature is created with a certain private key, it can only be successfully verified by the public key that belongs to that private key. Verifying a signature with anything other than the appropriate public key yields an invalid result. If a signature is successfully verified, it is confirmed that this signature was created by the entity that holds the private key.

Integrity Anyone is prevented from tampering with the signed contents. To verify a signature, the original message, in this case the digest value, is required. If an attacker would change any of the protected information, the calculated digest value would consequently change as well. The verification of a signature with the wrong digest value leads to an invalid result.

A summary of the BGPsec specification is presented next.

4.2 BGPsec Specification

The BGPsec protocol was developed by the IETF, the corresponding RFC 8205 [2] was published in 2017. Along with this RFC, additional documents were published that cover, e.g., algorithms, key and signature formats [42].

BGPsec is not a protocol on its own, but a protocol extension for BGP. This extension replaces a field within the BGP update message. No information are lost though, as the new field includes all information of the old field, as well as cryptographic signatures which protect the AS path.

Before a BGPsec capable speaker can originate or forward a BGPsec update, it first needs to check if the peering router also speaks BGPsec. BGP open messages contain capabilities which inform peering routers of special configuration details[1]. Routers signal each other the support of BGPsec by advertising the BGPsec capability. The contents of this capability are displayed in Figure 4.1. Each Address Family Identifier (AFI) and direction that is advertised is put in a separate capability. Thus, if a router wants to advertise full support for both sending and receiving IPv4 and IPv6 prefixes, four capabilities need to be added to the BGP open message. If both routers propagated BGPsec capabilities, BGPsec as well as plain BGP messages may be exchanged. If one of the two peers is unable to speak BGPsec, exchanging BGPsec updates is not possible. However, plain BGP messages may still be exchanged.

[1] A full list of all capabilities can be found here: https://www.iana.org/assignments/capability-codes/capability-codes.xhtml.

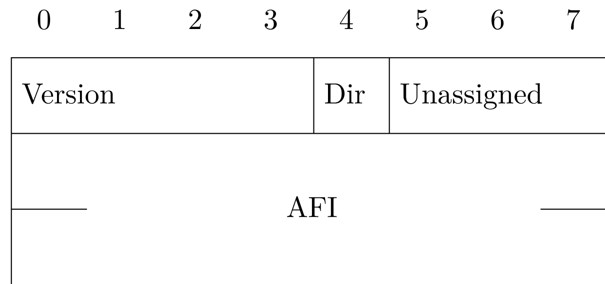


Figure 4.1: Capability as defined by the BGPsec specification [2]. The *Version* is currently 0. The direction bit, *Dir*, indicates whether a router can receive (0) or send (1) BGPsec updates. The *AFI* is either IPv4 or IPv6. *Unassigned* bits are currently not used.

To originate a BGPsec update, a BGPsec speaker needs to generate an update message that contains the BGPsec PATH attribute, which replaces the AS PATH attribute. Its structure and contents are shown in Figure 4.2.

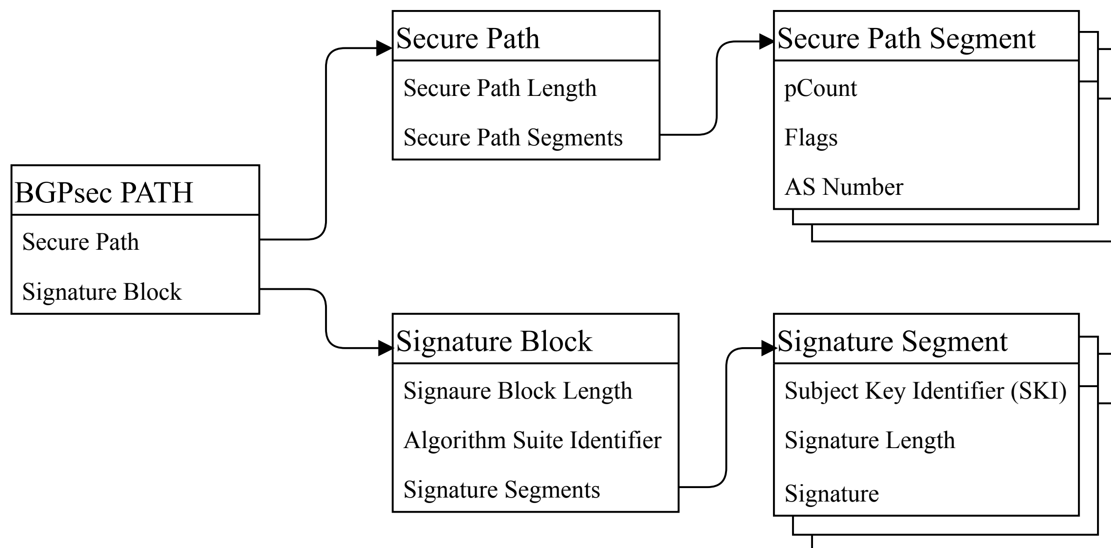


Figure 4.2: Structure of the BGPsec PATH attribute in an UML like fashion. Both the Secure Path as well as the Signature Block may contain multiple Secure Path Segments and Signature Segments.

The BGPsec PATH attribute contains a Secure Path and up to two Signature Blocks, one for each algorithm suite. Since there is currently only one suite defined, it is referred to as *the* Signature Block. Both the Secure Path and the Signature Block contain one Secure

Path Segment and Signature Segment for each inter-AS hop the announcement traversed. An update with path length of 3 therefore contains three Secure Path Segments and three Signature Segments.

When originating an announcement, there is no BGPsec PATH attribute yet, so the router has to generate one. At first, the router determines to which peer the update should be sent to. This is necessary because the target AS is part of the protected information. Second, the router takes the following information and applies a digest function to it:

- Secure Path Segment
- Target AS
- Algorithm Suite ID
- Address Family Identifier (AFI)
- Subsequent Address Family Identifier (SAFI)
- Network Layer Reachability Information (NLRI)

The Secure Path Segment contains the information of the own AS. Using the routers own private key, the router then signs the digest value. The resulting signature is wrapped in a Signature Segment which is then appended to the update. The update is now ready to be sent to the designated peer. This process must be repeated for each individual peer the update should be forwarded to: "[...] if a BGPsec speaker wishes to send a BGPsec UPDATE message to multiple BGP peers, it MUST generate a separate BGPsec UPDATE message for each unique peer AS to whom the UPDATE message is sent" [43]. Multiple prefixes must also be sent via separate updates: "A BGPsec UPDATE message MUST advertise a route to only a single prefix" [43]. All information of the list above are protected by the signature.

When the peer receives the BGPsec update, it repeats the same exacts steps to generate the digest value. If no one has altered the protected information, the hashing process yields the same exact digest value. Using the digest value, the signature and the public key of the originating AS, the signature can be verified. If the result yields a valid outcome, the update is deemed 'Valid'. Otherwise, the entire update is considered 'Not Valid'. These are the only two outcomes of BGPsec validation: "The validation procedure results in one of two states: 'Valid' and 'Not Valid'." [44].

Forwarding a BGPsec update is similar to originating one. The forwarding AS prepends its own credentials if form of a Secure Path Segment to the BGPsec PATH. It then takes

all the information from the BGPsec PATH (including the Signature Segments), the prefix and the target AS and applies the digest function to them. The resulting digest value is then signed using the routers private key. The signature is encapsulated into a Signature Segment which is then prepended to the BGPsec PATH. This completes the procedure and the BGPsec update can now forwarded. Figure 4.3 visualizes the process over multiple hops.

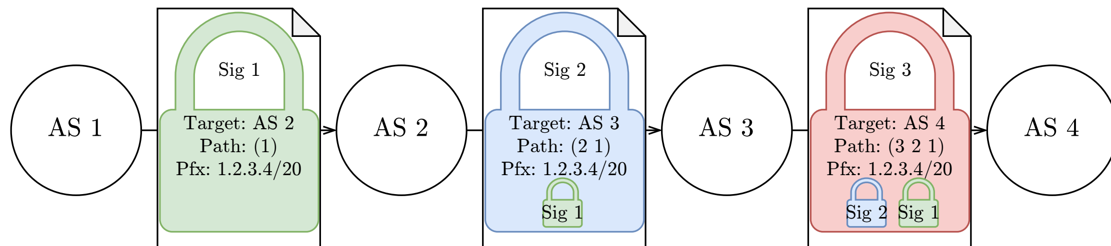


Figure 4.3: Signing chain from AS 1 up to AS 4. AS 1 signs the target AS, the BGPsec PATH and the prefix. AS 2 updates the target AS to match AS 3, appends its own AS to the path and signs everything, including the previous signature. AS 3 repeats the process.

In Figure 4.4, the validation process of a BGPsec update of length 3 is shown in detail. It describes the steps that AS 4 from Figure 4.3 has to perform in order to validate the update it received from AS 3.

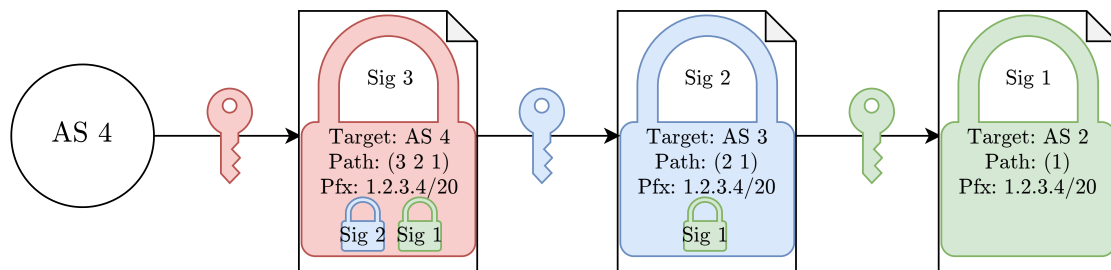


Figure 4.4: AS 4 validates the BGPsec update it received from AS 3 (see Figure 4.3). AS 4 uses the public keys of AS 1, 2 and 3 to verify the signatures they generated. The order of execution starts with the last prepended signature and ends with the first.

A more detailed view on the signing process itself is given in Figure 4.5. There, a BGPsec update undergoes the signing routine twice.

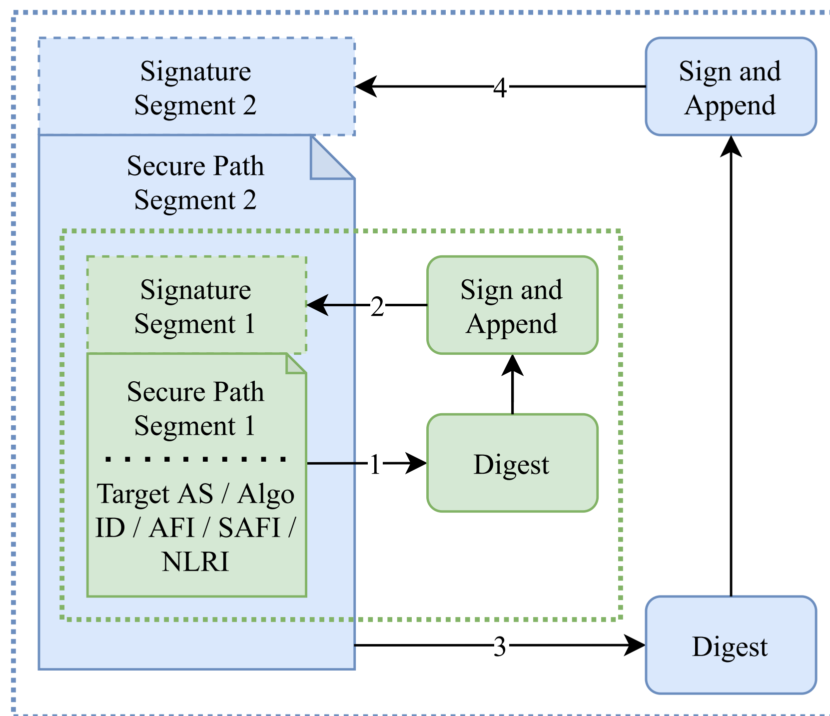


Figure 4.5: Iterative process of update generation. The originating AS generates an update (inner box) by 1) hashing the relevant information, then 2) signing the resulting digest value and appending the generated signature. The next forwarding AS (outer box) repeats the process by 3) hashing the previous data plus its own and then 4) generates and appends the signature.

Each time a BGPsec update is forwarded, the process depicted in Figure 4.5 is repeated. Validation is a reversed process of signing, it is shown in Figure 4.6. Here, an update of length two is verified. The validation follows a top down approach, i.e., the signature appended last is verified first. When hashing the data, the signature itself must be excluded from it. Next, the router key of the AS which generated the signature needs to be retrieved. The digest value and router key are then used to verify the corresponding signature. The outcome is either valid or invalid. If valid, the procedure continues until all signatures are verified this way. If invalid, one or more of either hash, signature or router key are erroneous and the validation ends prematurely, marking the update as 'Not Valid'. In case all signatures are valid, the update is considered 'Valid'. How to proceed with 'Valid' and 'Not Valid' announcements is left to the ASes policy. The specification suggests that the outcome should be considered for route selection: "It is expected that

the output of the validation procedure will be used as an input to BGP route selection. That said, BGP route selection, and thus the handling of the validation states, is a matter of local policy and is handled using local policy mechanisms" [44]. Note that regardless of the validation result, BGPsec updates should always be forwarded, even when the validation result is 'Not Valid': "A BGPsec router should sign and forward a signed update to upstream peers if it selected the update as the best path, regardless of whether the update passed or failed validation (at this router)" [45].

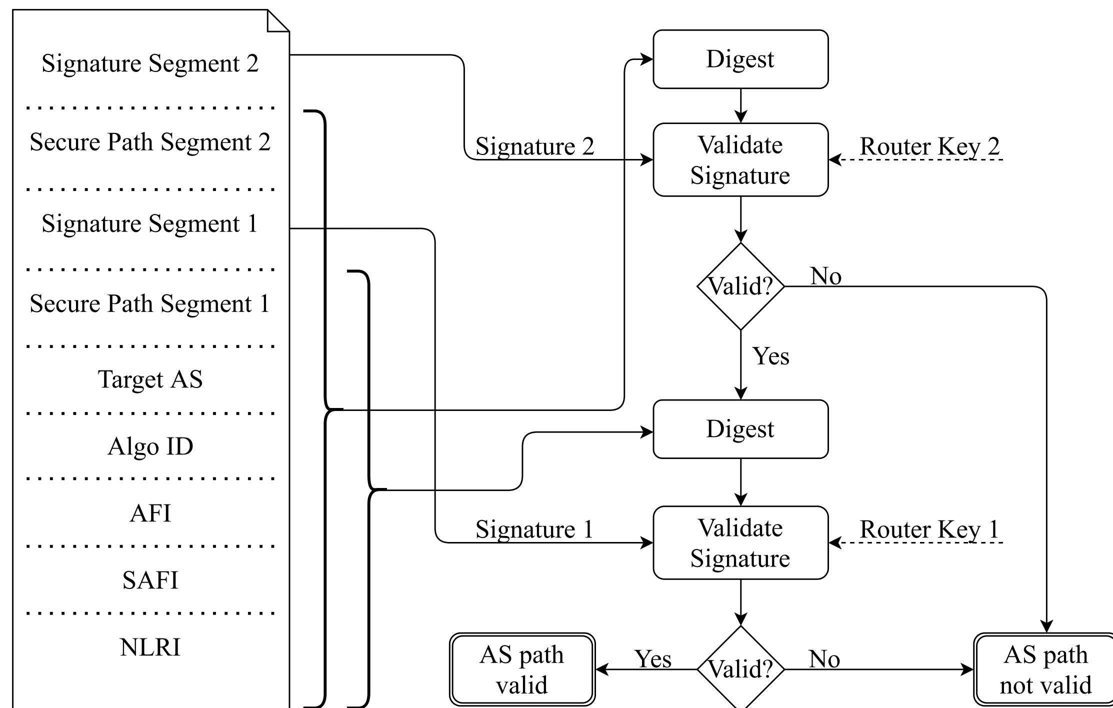


Figure 4.6: The BGPsec update on the left is processed top down until all signatures are verified. If one result is invalid, the process ends prematurely. The router keys are fetched from the RPKI cache servers.

4.3 Deployment of BGPsec

For ASes, rolling out BGPsec bears several issues [36]. Routers would have to run performance critical actions, i.e., cryptographic validations. To prevent the router from clogging, validations of BGPsec updates can be deferred, if there is heavy load on the router. Before a BGPsec update can be forwarded, though, it must be signed or otherwise downgraded to plain BGP. Since BGPsec requires each prefix to be announced via a

separate update, the performance issue is amplified even further. Additionally, each update must be generated separately for each receiving peer. Originating all of a routers n prefixes to all of its m peers would therefore require $n \cdot m$ individual updates. With plain BGP, the router would only need to generate a maximum of m individual updates.

Another concern with BGPsec is broad deployment across the Internet. If a BGPsec speaker peers with a non-BGPsec speaker, it is forced to transform BGPsec updates into plain, unsigned BGP updates. Thus, the announcement is stripped off all its protection mechanisms. One non-BGPsec speaker along the path is therefore enough to break the chain of forwarding a protected AS path. Using BGPsec without having BGPsec capable peers yields no benefits. ASes might hesitate to transition to BGPsec as long as no other ASes do the switch. Goldberg speaks of a "chicken-and-egg problem" [36] for AS operators. The idea of partial deployment of BGPsec is visioned in a way that initially, groups of connected ASes would simultaneously deploy BGPsec. These groups of ASes would grow with each neighboring AS that deploys BGPsec. Groups could also merge to form even larger groups and continue to grow until large portions the Internet are covered by AS path protection [46].

On a final note, there are only few implementations for BGPsec. One is the reference implementation, BGP-SRx, developed by the NIST. Using their SRxCryptoAPI library, the NIST included BGPsec support for BGP implementations such as GoBGPsec[2] and ExaBGPsec[3]. Those are, however, only forks of the original software.

4.4 Related Work

Since BGPsec has yet to experience deployment, little real-world research has been done with it. There is research that covers deployment and security concerns regarding the protocol extension.

Besides BGPsec, other path validation approaches for BGP exist, such as S-BGP [40] or soBGP [47]. BGPsec was based on the S-BGP idea of having X.509 certificates to sign and validate the AS path. soBGP additionally guaranteed that an announced path does physically exist. Both S-BGP and soBGP were abandoned for BGPsec along the way.

[2] https://github.com/usnistgov/gobgpsec
[3] https://github.com/usnistgov/exabgpsec

Lychev *et al.* [48] and Gill *et al.* [49] investigate partial deployment of various BGP security mechanisms, including BGPsec, and hand out suggestions.

A more grim deployment scenario of BGPsec is described in [37] by Gilad *et al.*. They predict an extremely long deployment period (if deployed at all) and Cohen *et al.* therefore suggest interim solutions such as path-end validation in [50].

To measure how a BGP security extension would perform, Kent *et al.* tested an S-BGP implementation and shared the results in [51].

Security concerns on BGP security mechanisms are given by Goldberg *et al.* in [52] and [36]. They elaborate on the security of BGPsec, S-BGP and soBGP.

The consensus in the research field of BGP security extensions is that deployment of such is a long and rocky road. The security gains are marginal compared to the computational overhead for BGP routers due to cryptographic operations. Because of the classic chicken and egg issue, ASes have little incentive to move towards BGPsec deployment, or any other solution such as path-end validation.

Nevertheless, there need to be implementations that offer the BGP security enhancements for AS path validation. Otherwise, the deployment can hardly be initialized. Therefore, this work aims to provide another implementation so that BGP operators have one more option to choose from.

The next chapter presents the problem statement that this thesis tackles.

5 Problem Statement

Currently, BGPsec has not received any deployment yet. Hence, ASes can not protect themselves against path hijacking. One problem that hinders deployment is that there are only few BGPsec implementations available. The reference implementation BGP-SRx [53] is integrated into the routing suite Quagga[1]. Unfortunately, the development state of Quagga is stale and BGP-SRx was never officially included into a release.

In 2017, Quagga was forked [54] into FRR, a more actively developed routing suite. FRR depends on RTRlib for ROV, but as of today, BGPsec is not part of it.

This work aims to provide a BGPsec implementation. It is divided into two separate parts: the *validation suite* and the *protocol handler*. The validation suite takes care of cryptographic routines and will be integrated with RTRlib. The protocol handler is responsible for protocol related operations and will take place within FRR. A concept for both implementations is worked out first. Then, the implementation itself occurs.

Another goal of this work is to provide a performance analysis of the BGPsec implementation. One reason why providers are hesitant to deploy BGPsec is because of the critical computational overhead on their BGP routers. The test results provided in this work will give providers a reference on how grave this performance overhead is. The performance of the implementation as a whole is tested in different scenarios. BGPsec in FRR will be compared against BGPsec in BGP-SRx. Both BGPsec implementations will also be compared to their plain BGP version.

By providing a BGPsec implementation for a thriving routing suite such as FRR, ASes might take BGPsec into consideration. Having as many ASes as possible to deploy BGPsec simultaneously is crucial for broad coverage of AS path protection throughout the Internet.

[1] https://www.quagga.net/

6 BGPsec Implementation

The BGPsec implementation is split in two parts: the cryptographic validation suite and the protocol handling. The validation suite is implemented within RTRlib, while the protocol handling takes place within the FRR routing suite.

This chapter is split into two parts. The first part discusses the design, requirements and implementation of the validation suite. The second part does the same for the protocol handler.

6.1 Validation Suite Design

The validation suite handles validation and signing procedures for the BGPsec path. This section discusses the design choices made for the implementation, establish feature and non-feature requirements, as well as showcase implementation details in form of code.

The validation suite is not specifically designed to be usable by FRR only. For this reason, the software that integrates or makes use of the validation suite is referred to as *parent software*. This allows to generalize use cases.

RTRlib is a fitting RPKI library to integrate the validation suite in. It is a client side implementation of the RTR protocol and additionally provides ROV. The library is open source and does only (optionally) depend on the libssh[1] library.

The validation suite is independent from any protocol handler. Requests only go in one direction, i.e., from the protocol handler to the validation suite. It is thus purely agnostic in regards to who makes a request.

[1] https://www.libssh.org/

There are feature and non-feature requirements. Feature requirements, e.g., validation and signing procedures, are mandatory for AS path validation. Non-feature requirement, e.g., the choice of the programming language, are not mandatory for the implementation to work, but rather a choice of the developer.

The following non-feature requirements are not defined by any specification. They were chosen with the intent to make the validation suite as applicable to any parent software as possible.

Software License The goal for the validation suite is to be open source and free to use. The RTRlib is developed under the MIT license [55], which fits the requirements for the validation suite.

External Libraries Any library that the validation suite depends on implicitly adds another dependency to the parent software. When depending on an external library, its size, its own dependencies and its license are adopted.

Implementation Language C is an established, well-known and widely supported programming language on a broad variety of operating systems. It is therefore a good choice to implement the validation suite in. In case a user choses to make use of the validation suite but happens to use a different language, numerous Foreign Function Interface (FFI) exist, which can be used to call C functions via different programming languages[2]. This way, parent software written in other languages could potentially make use of the implementation as well.

With RTRlib as the library of choice, the following non-feature requirements support a smooth and sophisticated inclusion of the validation suite Application Programming Interface (API).

Lightweight The implementation shall only provide necessary features to provide AS path validation and signing. Additional features are not required.

Compatible The new API shall merely extend the current API of RTRlib. The current API shall not be altered to guarantee compatibility with other parent software that already makes use of RTRlib.

[2] A list of FFIs can be found here https://en.wikipedia.org/wiki/Foreign_function_interface.

Similar Setup The setup to use the new API shall be similar to the ROV API. Both the current and the new API shall depend on the same internal procedures of establishing a connection to a cache server. This helps both developers and users to interact with the new API.

Independent AS path validation shall be independent from any protocol handler. A complete BGPsec implementation shall not be necessary to use the validation suite.

Documented The current documentation for the RTRlib shall be extended by AS path validation. This includes all new public functions and code examples, e.g., a minimum working example.

At the end of this chapter, the requirements are revisited and checked, if they have been met. Next, the API functions that need to be provided for AS path validation are examined.

6.1.1 Features

The basic tasks of the validation suite are stripped down to two vital functionalities:

Validation Given all required parameters, the validation suite performs validation of an arbitrary amount of signatures. It returns the final result of the validation without intermediate signature results, as they are not relevant for the protocol handler further down the line [3]. Should a signature yield an invalid result, the procedure ends prematurely and the validation procedure returns ‘Not Valid’. Otherwise, it returns ‘Valid’.

Signing The validation suite can sign BGPsec paths. All required parameters are therefor passed to the validation suite. After successfully signing the path, the signature and its length are returned.

If at any time an error occurs, the validation suite returns an appropriate error code and logs the error message. Errors are not handled by the validation suite, but must instead be handled by the caller of the function. The validation suite comes with a variety of utility functions that are also part of the new AS path validation API.

[3] Intermediate results might be relevant for other software that uses the validation suite, but for now they are not considered essential.

6.1.2 Crypto Library

To validate and sign BGPsec paths, cryptographic operations are required. According to the specification, these operations must be executed with the Elliptic Curve Digital Signature Algorithm (ECDSA) [56]. The digest value, which serves as input for the ECDSA, is generated by the Secure Hash Algorithm 2 (256 bits) (SHA-256) algorithm [57].

RTRlib should depend on an existing, throughout tested library to provide cryptographic functionalities. The following list specifies the requirements to this library.

1. **Provides C API** Since RTRlib is implemented in C, the crypto-library of choice must be compatible with C.
2. **Open-Source And Free To Use** RTRlib is open-source and free, so must be all its components and dependencies.
3. **Supports Algorithms** All required cryptographic algorithms must be supported. To support future algorithm suites, the library should additionally support a range of modern algorithms.
4. **Long-Term Support** To ensure that bugs and security issues are fixed, the library should receive support for years to come.
5. **Widely Supported** Helps with compatibility on different devices and operating systems.
6. **Secure** Must not contain major known bugs that compromise BGPsec security.
7. **Follows Standards** Must follow specification standards of required algorithms.

FRR already depends on a crypto library, *OpenSSL*[4]. Since this work aims to provide AS path validation for FRR, it is reasonable to first check on this already existing dependency. Should it meet the criteria, it would be convenient to also use it for RTRlib. Having a different crypto library as a dependency for RTRlib would result in a redundant dependency for FRR.

A short summary of OpenSSL is given in the following paragraphs. Every time it meets one of the criteria, it is checked with the appropriate number of the previous list.

[4] https://www.openssl.org

OpenSSL is an open-source and free to use [58] (1) library written in C (2). It provides all necessary algorithms, namely ECDSA and SHA-256 hashing [59, 60] and also offers a range of other algorithms that are potentially required in the future (3). There are two concurrent versions of OpenSSL, version 1.0 and 1.1. Although they both offer the required algorithms, version 1.1 is not downwards compatible with 1.0. Even though version 1.0 is no longer supported as of end of 2019 [61], RTRlib aims to support both versions, since both versions are widely used. For version 1.1, no end of life date is specified yet (4)[5].

The library is available on most common operating systems, most importantly, Unix (5). OpenSSL can be installed from appropriate repositories or can be build from source [62].

Vulnerabilities do appear from time to time, but they are tracked and mitigated by frequent patches [63] (6).

OpenSSL provides an (incomplete) lists of protocol standards they implement here [64] (7).

In conclusion, OpenSSL is a fitting crypto library to provide operations that are required for BGPsec AS path validation. It meets with all the requirements and since it is already part of FRR, it does not add another dependency to the routing suite.

6.1.3 AS Path Validation API

The validation suite API possesses three kinds of functions. There are 1) (de-)allocator functions, 2) utility functions and 3) validation/signing functions.

The first group of functions handles allocation and deallocation of data structures. These data structures contain the BGPsec data that is validated or signed. The functions of the second group help to prepare and assemble this data. Functions of the third group perform the actual validation/signing. The data that is required for these operations is stored in `structs` provided by C. In Listing 6.1, all C structs that are required for AS path validation are displayed.

[5] As of 25.07.2021

```
1  struct rtr_secure_path_seg {
2      struct rtr_secure_path_seg* next;
3      uint8_t  pcount;
4      uint8_t  flags;
5      uint32_t asn;
6  };
7
8  struct rtr_signature_seg {
9      struct rtr_signature_seg* next;
10     uint8_t  ski[SKI_SIZE];
11     uint16_t sig_len;
12     uint8_t* signature;
13 };
14
15 struct rtr_bgpsec_nlri {
16     uint8_t prefix_len;
17     struct lrtr_ip_addr prefix;
18 };
19
20 struct rtr_bgpsec {
21     uint8_t  algo_id;
22     uint8_t  safi;
23     uint16_t afi;
24     uint32_t my_as;
25     uint32_t target_as;
26     uint16_t sigs_len;
27     uint8_t  path_len;
28     struct rtr_bgpsec_nlri*     nlri;
29     struct rtr_signature_seg*   sigs;
30     struct rtr_secure_path_seg* path;
31 };
```

Listing 6.1: Newly introduced C structs for AS path validation in RTRlib. The structs with their members roughly represent the BGPsec PATH attribute.

The Secure Path Segment (line 1) and Signature Segment (line 8) are singly linked lists. Each segment holds the reference to the next segment of the list. The rest of their fields are identical to those depicted in Figure 4.2. The struct in line 15 is a wrapper for an already existing RTRlib data structure that holds the NLRI, i.e., the prefix (line 17). The wrapper additionally holds the prefix length in bits (line 16). Wrapping the `lrtr_ip_addr` struct is necessary because changes to current data structs are to be avoided.

To have all information in one place, i.e., one data struct, the BGPsec data struct (line 20) holds references to both linked lists (line 29 and 30). Additionally to the segment lists, the struct holds the algorithm id, the SAFI, the AFI, the own AS, the target AS and the amount of Signature and Secure Path Segments (line 21 to 27). The amount of Signature and Secure Path Segments are meta information that are not required for validation or signing. They are primarily used for error checks and it is therefore discouraged to change them manually.

Figure 6.1 visualizes the linked lists and how the BGPsec data struct holds references to them.

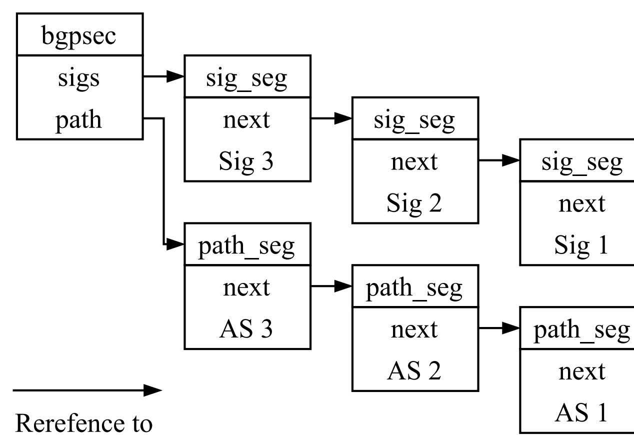


Figure 6.1: The bgpsec struct holds references to the signature and secure path lists. They always point to the head of it. The list elements themselves hold references to the next element of the list. In this example, the AS path is (3 2 1).

To prevent too much direct editing of the structs information, the allocator functions exist. The following functions allocate memory for the structs, initialize them and return a reference to them.

```
1  struct rtr_secure_path_seg*
2  bgpsec_new_secure_path_seg(uint8_t  pcount,
3                             uint8_t  flags,
4                             uint32_t asn);
5
6  struct rtr_signature_seg*
7  bgpsec_new_signature_seg(uint8_t* ski,
8                           uint16_t sig_len,
9                           uint8_t* signature);
10
11 struct rtr_bgpsec_nlri* bgpsec_nlri_new(void);
12
13 struct rtr_bgpsec* bgpsec_new(uint8_t  algo_id,
14                               uint8_t  safi,
15                               uint16_t afi,
16                               uint32_t my_as,
17                               uint32_t target_as,
18                               struct rtr_bgpsec_nlri nlri);
```

Listing 6.2: Allocator functions for the BGPsec structs.

The RTRlib structures are only required for validation and signature generation. As soon as the procedure is finished, they can be disposed of. To do so in a memory safe way, there are appropriate deallocator functions, as shown in Listing 6.3.

```
1 void bgpsec_secure_path_free(struct rtr_secure_path_seg* seg);
2
3 void bgpsec_signatures_free(struct rtr_signature_seg* seg);
4
5 void bgpsec_nlri_free(struct rtr_bgpsec_nlri* nlri);
6
7 void bgpsec_free(struct rtr_bgpsec* bgpsec);
```

Listing 6.3: Functions for deallocating BGPsec structs.

To deallocate the memory, a user passes a struct pointer to the appropriate function. Functions in line 1 and 3 take the singly linked segment lists as arguments and free them recursively. The deallocator function for the `rtr_bgpsec` struct in line 7 also frees the segment lists recursively. For the sake of completeness, the NLRI struct is freed via the function in line 5.

The second group of functions describe the utility functions that are required to further prepare the data before validation/signing it. The order of Signature and Secure Path Segments are crucial. New elements are *prepended* to the path, i.e., the new segment is attached to the head of the path. Prepending AS 3 to the path (2 1) would result in the path (3 2 1). This last in, first out (LIFO) order is expected by the validation and singing routines in RTRlib. To help the user assembling the data, utility functions allow appending to, prepending to, and stripping off segments from the `rtr_bgpsec` struct. Listing 6.4 shows these functions.

```
1  void bgpsec_append_sec_path_seg(
2                           struct rtr_bgpsec* bgpsec,
3                           struct rtr_secure_path_seg* new_seg);
4
5  int bgpsec_append_sig_seg(
6                           struct rtr_bgpsec* bgpsec,
7                           struct rtr_signature_seg* new_seg);
8
9  void bgpsec_prepend_sec_path_seg(
10                          struct rtr_bgpsec* bgpsec,
11                          struct rtr_secure_path_seg* new_seg);
12
13 int bgpsec_prepend_sig_seg(
14                          struct rtr_bgpsec* bgpsec,
15                          struct rtr_signature_seg* new_seg);
16
17 struct rtr_secure_path_seg*
18 bgpsec_pop_secure_path_seg(struct rtr_bgpsec* bgpsec);
19
20 struct rtr_signature_seg*
21 bgpsec_pop_signature_seg(struct rtr_bgpsec* bgpsec);
```

Listing 6.4: Utility functions that allow appending/prepending/stripping segments for Signature and Secure Path Segments.

All four append/prepend functions in line 1 to 15 take the `rtr_bgpsec` struct and the new segment as parameters. The functions then execute the desired action.

Prepending elements to the segment lists automatically updates the references of the `rtr_bgpsec`. Figure 6.2 illustrates what happens when a Secure Path Segment is prepended to an existing path.

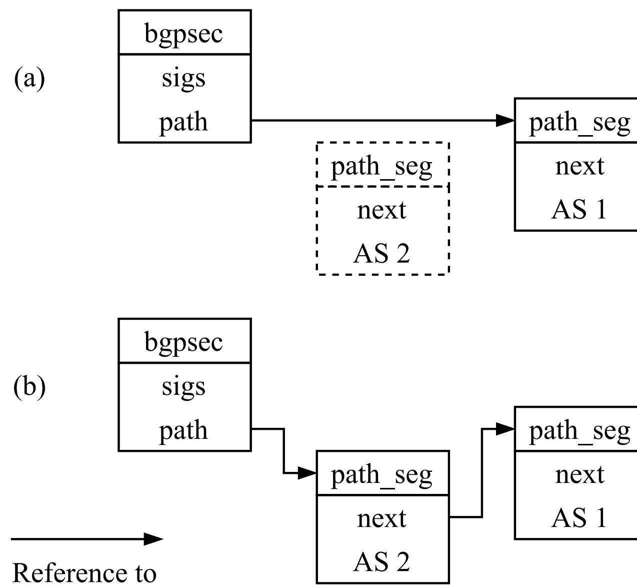


Figure 6.2: (a) shows the BGPsec struct that holds a reference to the head of the secure path list, AS 1. When prepending a new segment, AS 2, the references get updated accordingly (b). AS 2 is now the head of the list.

The pop functions in line 17 to 21 detach the first Secure Path or Signature Segment of that linked list, respectively, and return it.

These six functions enable the user to arrange the segments as they need.

After properly arranging the segments, the data as an entirety is then passed to either the validation or the signing procedure. Both functions can be found in Listing 6.5.

```
1 int bgpsec_validate_as_path(
2         const struct rtr_bgpsec* data,
3         struct rtr_mgr_config*   config);
4
5 int bgpsec_generate_signature(
6         const struct rtr_bgpsec*   data,
7         uint8_t*                   private_key,
8         struct rtr_signature_seg** new_signature);
```

Listing 6.5: One function for validating an AS path, another for generating a signature.

For validation (line 1 to 3), a user passes over the BGPsec `data` and the RTR `config` to the validation function. The RTR configuration holds a reference to the Subject Public Key Information (SPKI) table, which contains the router keys that were fetched from the RPKI. These public key information are necessary for verifying the signature that was generated with the appropriate private key. After the validation procedure finished, the caller is returned an integer that indicates a 'Valid' or 'Not Valid' path. Error codes are returned via the same integer. The caller is responsible for handling the validation result and eventual errors.

Signing (line 5 to 8) requires the user to already have attached its own Secure Path Segment to the `data` struct. The private key that is used for singing is also passed to the function. Last, an uninitialized Signature Segment is given as input. On return, `new_signature` contains the newly generated signature, together with its length. The Subject Key Identifier (SKI) needs to be added manually to `new_signature` by the user, as it is not contained within the information passed to the function.

Since handling the private key is a sensitive matter, reading it from the disk and handling its memory is left to the application that calls the function. The private key is passed to the signing function as raw bytes. The function itself does nothing with the key outside of the scope of signing.

RTRlib tries to catch various errors that occur during data preparation and signing/-validation. Although not all errors can be caught by RTRlib, it attempts to catch the following ones:

Error A generic error that can not be specified any further.

Load Private Key Error The private key could not be loaded. This error occurs if, e.g., the key was not found or is malformed.

Load Public Key Error The public key could not be loaded. Since the public keys are fetched via the RPKI, this error should only occur if a malformed public key was added manually to either the SPKI table or the RPKI cache.

Missing Data A missing data error occurs if either one of the following parameters is missing[6]: BGPsec data struct, Secure Path Segments, Signature Segments, SPKI table.

[6] Missing in this context means that the received parameter is `NULL`.

Signing Error Errors that occur during the signing procedure are signaled via this error code. The reason for such an error is not specified by OpenSSL and thus can not be further specified by RTRlib.

Unsupported Algorithm Suite Checks for unsupported algorithm suites prevent that RTRlib tries to verify or generate signatures with the wrong algorithms.

Unsupported AFI Only IPv4 and IPv6 AFI are allowed. Although the protocol handler is supposed to rule out such errors, RTRlib tries to provide some sort of safety for software that does not implement a protocol handler.

Wrong Segment Amount In case there is an uneven amount of Secure Path and Signature Segments, RTRlib returns this error. RTRlib has very limited ways of actually detecting this error. It therefore relies on the `sigs_len` and `paths_len` members of the BGPsec data structure. They contain the amount of segments respectively.

6.1.4 Project Structure

One requirement to the implementation is that the current API of RTRlib is merely extended and not altered in any way. To achieve better separation of the ROV and the AS path validation API, the new API is located in separate files. All AS path validation files are stored within the `rtrlib/bgpsec` directory. It contains five new files:

- `bgpsec.c`
- `bgpsec.h`
- `bgpsec_private.h`
- `bgpsec_utils.c`
- `bgpsec_utils_private.h`

`bgpsec.c` and `bgpsec_private.h` contain the declarations and definitions of the public API functions. `bgpsec.h` holds the data structures and error codes. `bgpsec_utils.c` and `bgpsec_utils_private.h` contain internal helper functions. They are not exposed to the public API.

The next section explains, how various features are implemented. Noteworthy parts of the implementation are highlighted with code excerpts.

6.2 Validation Suite Implementation

The following sections cover the implementation in detail. There are multiple components to the validation suite. An internal API for handling streams of raw bytes, BGPsec data alignment and the signing/validation routines.

6.2.1 Streams

Inspired by how FRR handles chunks of raw bytes, RTRlib adopts the functionality of streams. A stream in the context of RTRlib is a reference to an allocated memory region of custom size. It can not be accessed directly but must instead be accessed using appropriate functions. The advantage of this approach is that it prevents a developer from accidentally overwriting parts of the stream by having direct access to it. Changing a single bit of the data that is used for, e.g. hashing, would ultimately result in an invalid BGPsec update. Streams are encapsulated within an internal API of RTRlib that is not exposed to users of the RTRlib. It is intended for internal RTRlib developing purposes only. Its entire API can be seen in Listing 6.6.

```
1  struct stream {
2          uint16_t       size;
3          uint8_t*       stream;
4          const uint8_t* start;
5          uint16_t       w_head;
6          uint16_t       r_head;
7  };
8
9  struct stream* init_stream(uint16_t size);
10
11 struct stream* copy_stream(struct stream *s);
12
13 void free_stream(struct stream* s);
14
15 uint8_t* get_stream_start(struct stream* s);
16
17 uint16_t get_stream_size(struct stream* s);
18
19 void write_stream(struct stream* s, void* data, uint16_t len);
20
21 uint8_t read_stream(struct stream* s);
22
23 void read_n_bytes_stream(uint8_t* buff, struct stream* s,
24                          uint16_t len);
25
26 void read_stream_at(uint8_t* buff, struct stream* s, uint16_t start,
27                     uint16_t len);
```

Listing 6.6: Internal stream API of RTRlib. It offers functionalities for safely reading and writing raw chunks of bytes.

The size of a stream is stored within the `size` variable (line 2). The data of the steam, i.e., the raw bytes are pointed to by `stream` (line 3). To always have a reference to the beginning of the data, `start` (line 4) always points to the first byte of `stream`.

Each stream has a write and a read head (line 5 and 6). Reading from and writing data to a stream causes the heads to progress by the appropriate amount. When five bytes of a stream are read, the write head shifts five bytes along the stream data. The function `read_steam_at` (line 26) poses an exception, since the read position is determined by the user (for this call only). It does not affect the position of the heads. Using the technique of having read and write heads, a developer does not need to keep track of the current position when reading or writing.

To enable a developer to work safely with streams there are a variety of allocator-, helper- and getter-functions. The functions in line 9, 11 and 13 are used to initialize, copy and free a stream. Line 15 describes an getter function for the `start` variable. To get the total size of a stream, the function in line 17 returns the appropriate value.

Writing data to a stream works by passing arbitrary data together with a stream and a length value to the function in line 19. The routine writes `len` bytes of `data` to the stream `s`.

Reading a single byte from a stream can be done with the function in line 21. To read more than one byte, the function in line 23 is used. Since potentially more than one byte is returned, the result is written into a pre-allocated buffer. To read data from a stream at a certain position, the last function in line 26 can be used. A start position, from where on the data should be read, is passed to it. Otherwise, it behaves exactly like the read-function in line 23.

Multiple function calls of, e.g. `read_stream`, returns a different byte each time because the read head shifts with each call. An example can be seen in Listing 6.7.

```
1 struct stream* s = init_stream(3);
2
3 write_stream(s, 'a', 1);
4 write_stream(s, 'b', 1);
5 write_stream(s, 'c', 1);
6
7 read_stream(s); // returns 'a'
8 read_stream(s); // returns 'b'
9 read_stream(s); // returns 'c'
```

Listing 6.7: Example of the shifting read and write head mechanics of streams.

Here, the stream `s` is initialized with size 3. Then, the three bytes `'a'`, `'b'` and `'c'` are written to the stream (line 3 to 5). The write head shifted from position 0 to position 3. When reading three times (line 7 to 9), the three bytes are returned in the same order they were written. This time, the read head shifts from position 0 to 3.

The stream struct is found within the `bgpsec_utils.c` source file. It is made opaque by only forward declaring it in the header file `bgpsec_utils_private.h`. This way,

it cannot be initialized directly on the stack or heap using `malloc`. Instead, it must be initialized using the `init_stream` function, as demonstrated in line 2 of Listing 6.8.

```
1 struct stream* s = malloc(size); // Illegal
2 struct stream* s = init_stream(size); // Allowed
3
4 uint16_t s_size = s->size; // Illegal
5 uint16_t s_size = get_stream_size(s); // Allowed
6
7 uint8_t byte = s->stream[0]; // Illegal
8 uint8_t byte = read_stream(s); // Allowed
```

Listing 6.8: .

Any direct access on the struct is forbidden, as can be seen in line 4 and 7. For any kind of access, there are appropriate functions (line 5 and 8).

Utilizing streams is useful to enable safe read and write operations on raw chunks of bytes. The next section showcases where streams are used within the validation suite of RTRlib.

6.2.2 Data Hashing

Aligning the information that are hashed during the validation routine was optimized during the specification of BGPsec. Figure 4.6 from Section 4.2 shows how the data needs to be aligned in order to verify all signatures of an update without having to re-align it.

Starting from the beginning of the byte sequence, it is sufficient to calculate the offset from one Secure Path Segment to the next with each iteration. The offset is always the size of a Secure Path Segment plus the size of a Signature Segment. Because signatures are variable in length, the offset needs to be calculated with each iteration. In the implementation, this is handled with a for-loop, as shown in Listing 6.3.

```
1  s_size = get_stream_size(stream)
2
3  for (total_offset = 0, next_offset = 0;
4       total_offset <= s_size;
5       total_offset += next_offset):
6
7       bytes = read_stream_at(stream, total_offset,
8                              s_size - total_offset)
9       digest = hash_bytes(bytes)
10
11      /* Validate */
12
13      next_offset = sig_seg_len + sec_path_len
14 done
```

Figure 6.3: Truncated code of the validation for-loop. It generates a digest value with each iteration. With each iteration, the offset shifts to the next starting point.

The aligned data is already contained in `stream`. In the beginning, the size of the stream is stored in `s_size` in line 1. Next, in line 3, the two variables `total_offset` and `next_offset` are initialized to be 0. The first variable holds the offset of the entire byte sequence. The second variable holds the amount of bytes that `total_offset` needs to shift so `stream` is read from the correct position on in the next iteration. `next_offset` gets updated in line 13. Line 7 and 8 get the required bytes for hashing and line 9 hashes them. The entire process is visualized in Figure 6.4.

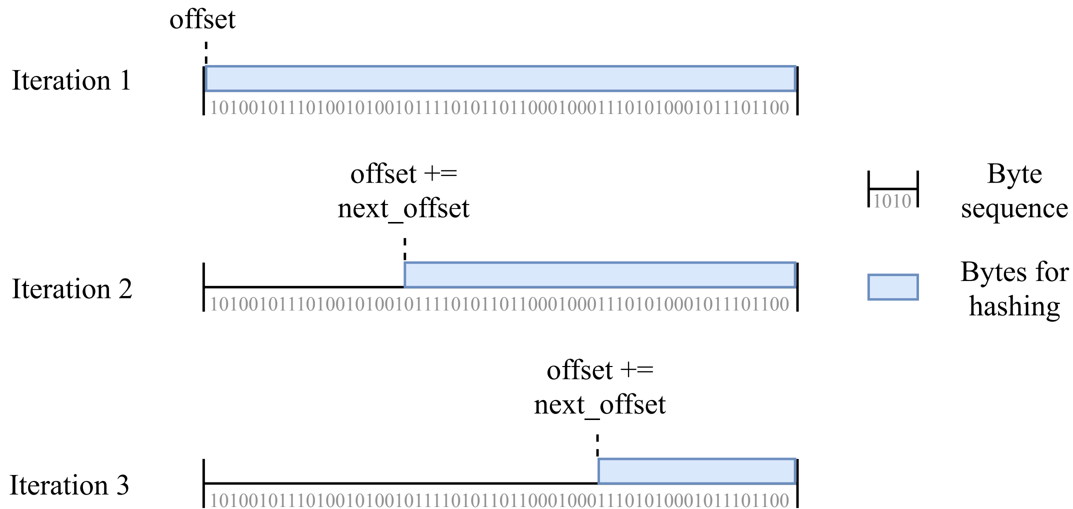


Figure 6.4: Visual representation of the offset shifting code in Listing 6.3. The bytes are aligned from left to right as a stream.

Each iteration, bytes to the right of the offset are hashed. The offset then shifts to the right by a variable amount of bytes and the next iteration begins. The amount of shifted bytes is the size of a Secure Path Segment plus the size of the next Signature Segment.

The validation procedure, which was merely symbolic in line 10 of Listing 6.3, is subject of the next section.

6.2.3 Signature Validation

As soon as a digest value is generated, the signature can be verified using the appropriate public router key. This key is stored within the SPKI table. It is fetched using the signers ASN and the keys SKI. If the required router key is not present during validation, the BGPsec RFC specifies to mark the Signature Block (and thus the entire BGPsec update as currently only one block is used) as 'Not Valid'. "Locate the public key [...]. If this check finds no such matching SKI value, then mark the entire Signature_Block as 'Not Valid' [...]" [65].

The OpenSSL validation function requires the SPKI to be encapsulated into an OpenSSL struct. After the conversion, the public key, the signature and the digest value are passed to the cryptographic validation function provided by OpenSSL. The result is

either ‘Valid’ or ‘Not Valid’. Listing 6.9 shows the same truncated loop that was already shown in Figure 6.3, this time focused on the validation part.

```
result = VALID

for (i = 0;
     i < sig_len OR result != NOT_VALID;
     i++):

     /* Calculate digest value */

     /* Get next signature, ASN and SKI */

     spki = spki_table_search_key(spki_table, asn, ski)

     pubkey = d2i_EC_PUBKEY(spki)

     result = ECDSA_verify(digest, signature, pubkey)

     /* Calculate next offset */
done

return result;
```

Listing 6.9: Loop that verifies all signatures. The focus here lies on the router key conversion and the validation itself.

The `result` variable in line 1 holds the validation result of each iteration. It is initialized to `VALID` so the for-loop condition in line 4 does not fail on the first iteration. Prior checks on `sig_len` assure that it is greater than 0 to have the loop run at least once. This needs to happen so `result` gets updated at least once before it is returned in line 20. The for-loop conditions in line 3 to 5 (compare line 3 to 5 of Listing 6.3) changed for the sake of brevity and focus on validation. The loop runs as long as there are signatures left unverified *or* the result of the previous iteration was ‘Not Valid’ (line 4). After the digest value has been calculated, the SPKI information is fetched from the SPKI table (line 11). Then, the SPKI is converted into a wholesome EC public key in line 13. Line 15 verifies the signature using the digest, the signature and the public key. Should the outcome be ‘Valid‘, the next iteration starts (implying there is a next signature to be verified). Otherwise, if the result is ‘Not Valid’, the loop breaks prematurely on the condition in line 4.

6.2.4 Signature Generation

Signing a BGPsec update is similar to validating one, with the difference that signing is not an iterative procedure. The byte sequence is aligned, hashed and then signed once. Listing 6.10 shows the defining parts of the implementation.

```
1 privkey = d2i_ECPrivateKey(byte_key)
2
3 bytes = align_byte_sequence(bgpsec_data)
4
5 hash = hash_byte_sequence(bytes)
6
7 new_signature = ECDSA_sign(hash, privkey)
```

Listing 6.10: Truncated code for generating a BGPsec signature. The process is not iterative and hence does not require a loop.

For signing, the private key of the router is loaded, as shown in line 1. Next, in line 3, the data is then aligned based on the input parameter `bgpsec_data`. It holds the data that is signed, including the data of the signing AS. In line 5, the byte sequence is hashed. Using the private key, the hash is signed in line 7. `new_signature` now contains the generated signature.

6.2.5 Revisiting Validation Suite Requirements

To finish this section, the requirements to the validation suite that were made in Section 6.1 are revisited. Each requirement is listed once more and checked, if it has been met.

Lightweight The validation suite has only two public features, namely validating and signing AS paths. To do so, OpenSSL is added as a dependency to RTRlib. Except for AS path validation, no other features were implemented. ✓

Compatible It was possible to add the validation suite to RTRlib without having to make changes to the current API. Software that currently depends on RTRlib would not break due to these changes. ✓

Similar Setup Router keys and prefixes are fetched from the cache server via the same connection. Changes to the setup were therefore not required. ✓

Independent Having a BGPsec protocol handler interacting with the validation suite is not required for signing or validating AS paths. All it needs is a connection to a cache server. ✓

Documented The documentation for the public API functions of AS path validation are included into the Doxygen[7] toolchain for RTRlib. When the documentation for RTRlib is generated, AS path validation is now part of it. ✓

All requirements have been met. This closes the validation suite implementation. The next section covers the implementation process of the protocol handler.

6.3 Protocol Handler Design

The protocol handlers duty is parsing and assembling BGPsec PATH attributes. It is build on top of BGP, which is already implemented by FRR. By utilizing the validation suite, the protocol handler signs and validates BGPsec updates.

6.3.1 Choice for FRR

The protocol handler is integrated into the FRR routing suite. One reason being that RTRlib is already used by FRR. RTRlib provides FRR with ROV and it is therefore only convenient to provide AS path validation through the same library. When FRR was forked from Quagga, the project was restructured and the development accelerated. New features were implemented rapidly. Between the years 2017 and 2019, six major releases were published. Development for Quagga is stale, its latest release, version 1.2.4, was published February 2018[8]

The BGP implementation of FRR is fully functional and provides all prerequisites for BGPsec such as 4-byte AS number support [66] and the multiprotocol extension [67].

Last, with BGP-SRx, Quagga already is provided with RPKI functionalities, making an additional BGPsec implementation in Quagga redundant. Bird also received a BGPsec implementation, although it was never officially released.

[7] https://www.doxygen.nl/index.html
[8] https://github.com/Quagga/quagga/releases/tag/quagga-1.2.4

6.3.2 Design

Just like with the validation suite, there are non-feature requirements to the protocol handler as well. These requirements are similar, although some of them differ in their meaning.

Lightweight The implementation shall only implement necessary features. If BGPsec is not enabled, it must not slow down the BGP protocol handler.

Exchangeable It shall be possible to remove the BGPsec protocol handler from FRR without breaking FRR or having to modify large portions of the BGP code.

Configurable To en- or disable BGPsec, it shall be configurable via FRR. Also, several recommendations by the RFC on *what* a user should be able to configure shall also be implemented.

Independent The protocol handler for BGPsec shall be compatible with other validation suites without the need to rewrite large portions of the FRR code base. Exchanging the function calls to the current validation suite with appropriate new ones shall be sufficient

Documented The usage of AS path validation shall to be documented with all its configurable options. The documentation shall also be integrated into the build chain of FRR.

At the end of this chapter, the above mentioned requirements are revisited and it will be checked, which of them have been met.

FRR can be extended with modules. A module is a piece of code that is build during compilation and loaded during run-time. Functions of the module are called via *hooks*. If hook-calling a function of a module that is not loaded, this call is ignored. This allows for a cleaner separation between the FRR code base and the protocol handler. However, separating BGPsec and BGP code is not always possible nor reasonable, as shown later in this section.

Since BGPsec is considered a protocol extension for BGP, there is no need to implement BGP functionalities from scratch. Instead, they can be used just the same, if the payload of BGPsec updates is prepared appropriately. This prevents a lot of code duplication that would otherwise be introduced with BGPsec. AS an example, FRR implements the

`aspath` struct. It is used to store AS path information, such as AS segments. Many functions take `aspath` as an input parameter. Instead of making all those functions compatible with BGPsec structures, the BGPsec PATH is translated into a `aspath`.

The following new features are essential to the protocol handler implementation:

Capability Negotiation Sending and receiving BGPsec updates requires two peers to know the capabilities of one another. Those are negotiated via the BGP open messages. The protocol handler needs to be able to create and parse the BGPsec capabilities of the open message.

Error Checking The semantics and syntax of BGPsec updates must be checked. Any inconsistencies must be handled according to the specification.

BGPsec PATH Parsing All contents of the new attribute must be parsed and processed appropriately.

BGPsec Update Construction To originate or forward a BGPsec update, the protocol handler needs to be able to generate a new BGPsec PATH attribute or append data to an existing one.

BGP Update Reconstruction In case it is not be possible to forward an announcement as a BGPsec update, it must be transformed into a plain BGP update.

6.3.3 The Module and Configuration

FRR already supports ROV, which is provided by RTRlib. This support is enabled via an RPKI module[9]. When configured accordingly, the BGP daemon performs ROV on incoming announcements. Prior to the validation, a cache server must be configured from where the RPKI information are fetched. RTRlib always fetches the prefix information as well as the SPKI data, if available. It is thus sufficient to also utilize the current cache configuration command for AS path validation. An example for cache configuration can be seen in Listing 6.11.

[9] https://github.com/FRRouting/frr/blob/master/bgpd/bgp_rpki.c

```
1 rpki
2     rpki cache 172.18.0.100 8383 preference 1
3     exit
```

Listing 6.11: Example configuration of an RPKI cache server in FRR. If the cache contains router keys, they are fetched together with the prefix information.

The command in line 1 enters the `rpki` environment. Line 2 reads as "connect to an `rpki cache` at the IP address `172.18.0.100` and port `8383` with `preference 1`". Line 3 `exit`s the `rpki` environment.

As for now, the approach of adding AS path validation to the RPKI module is convenient and appropriate. Because of the modular design of FRR, AS path validation could also be outsourced into a different module, if it should become necessary.

The following configuration commands are required for a minimal working BGPsec implementation.

1. **Cache Server Configuration** As mentioned above, we depend on the configuration command for setting up a cache server that is already used within the RPKI module. The cache server is required to fetch SPKI data.
2. **Private Key Location** The RPKI module needs to know the location of the private key to read it. Therefore, there needs to be a command to specify this location.
3. **Private Key SKI** Because the private key file does not necessarily contain the SKI, the user has to set it manually via a command.
4. **Capability Options** Section 4.2 defines the needed capabilities for BGPsec. To cover all cases, four options are necessary.
5. **Debugging** For debugging purposes, there is a command that logs BGPsec specific operations, e.g., whether or not an update validation was successful.

Commands 1 to 4 are mandatory for a minimum BGPsec implementation. Command 5 is optional but indispensable for the development process. Command 1 to 3 are implemented within the RPKI module. The capability and debugging commands need

to be implemented in the native BGP source code. They depend on existing commands and cannot be outsourced into the RPKI module.

The specific files that are updated are `bgp_vtysh.c` and `bgp_debug.c`. The first contains the capability commands, the latter the debug commands. The rest of the commands are located within the RPKI module file `bgp_rpki.c`.

The next section dives into the implementation details and elaborates, why certain design decisions have been made.

6.4 Protocol Handler Implementation

In this section, the implementation details of the protocol handler are inspected in detail. Whenever it is possible, pseudocode is provided. If the context of the relevant code is too vast, the implementation details are described without code as best as possible.

6.4.1 Commands

To give users control over BGPsec on the router, they need to have options like setting the cache server or the private key. A running BGP daemon can be configured directly via the *vtysh*[10] terminal. Alternatively, a configuration file can be passed to the daemon on startup. BGP commands have environments in which they can be applied. As can be seen in Listing 6.11, the `rpki` scope must be entered before the `cache` command can be used. The newly introduced BGPsec commands are applied to the following scopes:

1. **Router scope** In this scope, a BGP router is configured. It holds all information about its neighbors. It is intuitive to assign the *private key location* and *private key SKI* commands to it. Since capabilities for neighbors are also declared here, the BGPsec *capability options* are applied here as well.
2. **Global scope** Settings which apply globally, such as logging or debugging, are assigned to this scope. The BGPsec *debugging* command takes place here.

[10] http://docs.frrouting.org/en/latest/vtysh.html

Since there are no new commands added to the RPKI scope, it is omitted here. The following Listing 6.12 showcases the new commands in action.

```
debug rpki
debug bgp bgpsec

rpki
    rpki cache 172.18.0.100 8383 preference 1
    exit

router bgp 64496
    bgp router-id 10.10.10.10

    bgpsec privkey /path/to/privkey.der
    bgpsec privkey ski AB4D910F55CAE71A215EF3CAFE3ACC45B5EEC154

    neighbor 1.2.3.4 capability bgpsec send ipv4
    neighbor 1.2.3.4 capability bgpsec send ipv6

    neighbor 1.2.3.4 capability bgpsec recv ipv4
    neighbor 1.2.3.4 capability bgpsec recv ipv6

    neighbor 5.6.7.8 capability bgpsec both any
```

Listing 6.12: All commands that are relevant for configuring BGPsec. New commands are highlighted.

The first two lines enable debugging for RPKI and BGPsec. They work independently from each other. RPKI debug messages inform the user about the initialization process, connection establishment and ROV results. BGPsec debugging additionally informs the user about AS path validation results, announced and received BGPsec capabilities and errors related to BGPsec.

In line 8, the router scope is entered. Within, the private key is loaded from a file and the SKI is set with the appropriate commands (line 11 and 12). The private key must be loaded successfully prior to setting the SKI, thus the those commands must be executed in this order. If no key is set when setting the SKI, an error is returned.

Line 14 to 20 show, how the new capability command is working by configuring two neighbors. The current functionality of setting capabilities is extended with BGPsec. The syntax is: `neighbor <IP> capability bgpsec <send/recv/both> <ipv4/ipv6/any>`.

As seen in line 14 to 18, setting the capabilities to *send* and *receive* IPv4 and IPv6 would require four lines in total per neighbor. Line 20 demonstrates, how these four commands can be combined into a single line using the parameters *both* and *any*.

These are all the commands that are required to execute the performance tests in Chapter 7.

6.4.2 Capability Negotiation

BGPsec capabilities need to be added to the BGP open message based on the routers configuration. Received capabilities need to be parsed and applied to the appropriate peer. The FRR function `bgp_open_capability` is responsible for adding the appropriate capabilities to the open message. This is where the BGPsec code is added. Capabilities for a peer are set using flags, for each peer individually. Based on which flags are set for the peer, capabilities are added to the open message.

```
1  put_bgpsec_cap(open_msg, peer):
2
3      if peer->caps == SEND_IPV4:
4          put(open_msg, version)
5          put(open_msg, send)
6          put(open_msg, IPv4)
7
8      if peer->caps == SEND_IPV6:
9          put(open_msg, version)
10         put(open_msg, send)
11         put(open_msg, IPv6)
12
13     if peer->caps == RECEIVE_IPV4:
14         put(open_msg, version)
15         put(open_msg, receive)
16         put(open_msg, IPv4)
17
18     if peer->caps == RECEIVE_IPV6:
19         put(open_msg, version)
20         put(open_msg, receive)
21         put(open_msg, IPv6)
```

Listing 6.13: Pseudocode for adding BGPsec capabilities to the BGP open message of a specific peer. Each of the four if-statements checks for a direction-AFI combination. In case the capability for `peer` is set, the `version`, `direction` and `AFI` are added to the `open_msg` stream.

On the receiving end, parsing the open message needs to be extended with BGPsec capability parsing. Each capability has its own numeric code. In `bgp_capability_parse`, the BGPsec capability parsing routine is invoked if the appropriate capability code is encountered. If so, the encoded capability is passed to the RPKI module where it is processed. Various error checks make sure that the capability is not malformed and does not contain illegal values. Appropriate flags are then set for the peer from which the BGP open message was received.

```
1  bgpsec_capability_parse(cap, peer):
2
3      if cap->direction == send:
4          if cap->afi == IPv4:
5              set_flags(peer, SEND_IPV4)
6          else if cap->afi == IPv6:
7              set_flags(peer, SEND_IPV6)
8
9      else if cap->direction == receive:
10         if cap->afi == IPv4:
11             set_flags(peer, RECEIVE_IPV4)
12         else if cap->afi == IPv6:
13             set_flags(peer, RECEIVE_IPV6)
```

Listing 6.14: Pseudocode for parsing a received BGPsec capability and setting flags for that specific peer. The `direction` and `afi` of the received `cap` are checked in nested if-statements. If a condition is met, the flags for the peer are set appropriately. This function can only process one capability at a time.

Decisions, e.g. whether or not a BGPsec update is forwarded to this specific peer, are based on the flags and capabilities set by the functions shown in Listing 6.13 and 6.14.

6.4.3 Storing BGPsec Data

Data structures that need to be available throughout the BGP implementation, such as the new `bgpsec_aspath`, cannot be declared within a module. Therefore, functions to allocate, copy and free BGPsec data structures reside within the `bgpd/bgp_aspath.c` file, not the RPKI module. Functions that initialize the BGPsec PATH storage, compare BGPsec PATHs and manipulate Signature and Secure Paths Segments are also found there. In total, 24 new functions were added for these purposes.

There are a total of five new structures to store BGPsec data. With the exception of two, the FRR structs are almost identical to their RTRlib equivalent, which were described in Section 6.1.3. These two exceptions are the `bgpsec_sigblock` and the `private_key`. Listing 6.15 shows both new structs.

```
1  struct bgpsec_sigblock {
2      uint16_t                length;
3      uint8_t                 alg_id;
4      uint16_t                sig_count;
5      struct bgpsec_sigseg*   sigsegs;
6  };
7
8  struct private_key {
9      char*           filepath;
10     uint16_t        filepath_len;
11     uint8_t*        key_buffer;
12     uint16_t        key_len;
13     uint8_t         ski[SKI_SIZE];
14     unsigned char   ski_str[SKI_STR_SIZE];
15     bool            loaded;
16 };
```

Listing 6.15: Definition of the `private_key` and the `bgpsec_sigblock` struct.

`bgpsec_sigblock` is a wrapper for signature related data. Future BGPsec versions will define two separate Signature Blocks per BGPsec PATH. The wrapper allows separation on software level by defining two individual `bgpsec_sigblocks`. Line 2 holds the total length of the Signature Block, line 3 holds the algorithm ID. The amount of Signature Segments within the block is stored in line 4. The segments themselves form a singly linked list. Its head is referenced in line 5.

The `private_key` holds the private key information. Key data is loaded from disk during BGP daemon startup. Line 2 and 3 hold the file path location string and length of the private key. Its raw bytes and its length are stored in line 4 and 5. The corresponding SKI is stored as raw bytes in line 6. Because of its static length of 20 bytes, it is not necessary to have a field of dynamic length. For debugging purposes, the SKI is stored in human readable format in line 7. The field in line 8 tells whether or not the key was successfully loaded.

6.4.4 BGPsec Update Generation

Since the BGPsec path is only a new BGP attribute, the update generating routine does not have to be altered fundamentally. In fact, adding the attribute to the update message works similar to adding the capability to the open message. The only precaution that

needs to be made is that a BGPsec update must not contain both the BGPsec PATH and the AS PATH attributes at the same time.

In FRR, the function `bgp_packet_attribute` generates the attributes for a BGP update message. A function-hook to the RPKI module is added there. The shortened process of generating the BGPsec PATH is laid out as pseudocode in Listing 6.16.

```
1  build_bgpsec_path(update, peer, bgp, attr):
2      if send_bgpsec(peer, bgp) == false:
3          return false
4
5      my_segment = bgpsec_sps_new(65532, 1, 0)
6      new_path = bgpsec_aspath_new(bgp, attr, my_segment)
7
8      store_bgpsec_aspath(new_path)
9
10     gen_bgpsec_sig(new_path, bgp, attr)
11
12     write_bgpsec_aspath(update, new_path)
```

Listing 6.16: Simplified pseudocode for generating a BGPsec PATH and writing it to the update.

The function `build_bgpsec_path` handles attribute and signature generation and appends them to the update message. In the beginning in line 2, it is checked whether or not the peer and the own router are configured to speak BGPsec. If one of them is not, the process is aborted prematurely. Else, the Secure Path Segment of the constructing router and the BGPsec PATH struct are initialized in line 5 and 6. Prior to signing the path, it is stored in memory for later use using `store_bgpsec_aspath` (line 8). Then, in line 10, the signature is generated via `gen_bgpsec_sig` and is then appended to `new_path`. Finally in line 14, the entire BGPsec PATH attribute is appended to the update message. The update is now a BGPsec update and the usual update sending/forwarding routine of FRR continues.

6.4.5 BGPsec Update Parsing

Incoming BGPsec updates undergo various error checks, defined by the RFC. Some of these checks are performed as the BGPsec PATH attribute is parsed. A malformed update must be handled accordingly, usually with a *treat-as-withdraw* [68] approach.

Parsing a BGPsec update is not very different from parsing a plain BGP update. Instead of an AS PATH attribute, the update holds the BGPsec PATH attribute. The `bgp_update_parse` function, which processes incoming updates, invokes the `bgp_attr_parse` function. The latter iterates over all attributes within the update. If it encounters the BGPsec PATH attribute, a hook to the RPKI module is called, which parses the BGPsec PATH and adds a `bgpsec_aspath` to the `attr` struct of the update. The `bgpsec_aspath` is holds the information of the parsed BGPsec PATH attribute.

When processing a BGPsec path, the AS path information are redundantly stored in non-BGPsec structures. Those are required for, e.g., best path selection. This way, massive code duplication and altering existing functions can be avoided. Otherwise, all routines that work with AS paths would need to be altered to work with BGPsec structs. Hence, AS path information received from BGPsec updates are processed as if they would have been extracted from a plain BGP update. AS path validation does of course still occur.

Another advantage of redundant AS path data storing is quicker AS PATH reconstruction. Should a BGPsec update be forwarded without AS path protection, a plain BGP update needs to be constructed instead. Since the AS path information is already stored in a non-BGPsec way during BGPsec path processing, this additional step can be skipped. The implementation simply falls back on these plain BGP structures.

The BGPsec PATH contains *length* fields that tell, how many of the following bytes belong to which information. The attribute is parsed until no bytes are left. Should the byte stream end prematurely or should there be trailing bytes left, an error is returned and the update is deemed malformed. During the parsing processes, various error checks occur, e.g., if there is an equal amount of Secure Path and Signature Segments.

Validation of the update cannot occur until all attributes of the BGPsec update have been processed. To verify a signature, the announced NLRI must be present, which is contained within the multiprotocol attribute. Even though the BGPsec PATH attribute should be placed at the end of the update, this is not a mandatory requirement according to the specification. For this reason, the validation is postponed until the very end of the update processing.

To execute the validation itself, the `bgpsec_aspath` structs, which holds the BGPsec PATH data of the update, is passed to the `val_bgpsec_aspath` function within the RPKI module. The data is still stored in FRR data structures and must first be converted

into RTRlib structures. After the conversion, the data is passed to the RTRlib validation function. The result is given back to the module, which returns it to the FRR BGP code to make decisions based on the outcome.

6.4.6 Revisiting Protocol Handler Requirements

This section takes a look back on the design requirements placed on the protocol handler in Section 6.3.2.

Lightweight Redundancy that would be introduced with BGPsec is reduced by using as much of the current FRR BGP routines as possible. BGPsec related code is kept separate from BGP code as much as possible. ✓

Exchangeable Thanks to the hook-system that FRR implements, BGPsec can be disabled during compilation without breaking the software. BGPsec routines are only executed when available and are otherwise ignored. ✓

Configurable Right now, only commands that are crucial to configure BGPsec are available. As time passes and reviews are made, additional requirements will be made to the implementation. (✓)

Independent Calls to the RTRlib are reduced to allocating memory, copying data, as well as validating and signing BGPsec paths. A total of 19 BGPsec related calls to RTRlib are made throughout the RPKI module. Exchanging these calls with functions of another validation suite is all it takes to make the switch, given that equivalent functions are provided. ✓

Documented Although the BGPsec implementation comes along with code comments on what is happening, user documentation is still missing. This is partly because the user requirements to the implementation are still developing and are not yet final. ×

In conclusion, the most crucial requirements are met. The requirements that users impose on BGPsec still need to be identified throughout. Also, a user manual on how to use AS path protection with FRR is still incomplete.

Now that the BGPsec implementation is complete, it needs to be analyzed. Tests are covered by the next chapter.

7 Performance Analysis

In this chapter, the correctness of the BGPsec implementation is verified. Afterwards, the performance is measured and compared to the reference implementation.

7.1 General

In general, all correctness and performance tests are executed in a containerized environment using Docker[1] and Docker Compose[2]. Each application resides within its own container. They are located within a common network which is spawned using Docker. An example of a Docker Compose configuration used for one of these tests can be found in Appendix A.1.

Complex simulated networks are neither necessary nor feasible for the scenarios in question. The liveliness of the network is not relevant. Variance in AS path length can be achieved by chaining multiple router instances together. For some tests, artificially generated BGPsec updates are required. Those updates are created with tools specifically developed for this work.

All tests are executed on an machine with 500 GB of RAM and an AMD EPYC 7702 64-Core processor. FRR version 7.5.1 and BGP-SRx version 5.1.5 were used throughout all tests.

7.2 Tooling

In total, three tools are required throughout all tests. They were implemented for the course of this work and are introduced here. There are currently no other tools that

[1] https://www.docker.com/
[2] https://docs.docker.com/compose/

provide functionalities such as BGPsec update forwarding, router key generation or distribution.

7.2.1 Public/Private Key Generator

Without public-private-key pairs, BGPsec updates cannot be signed or validated. The public keys are fed into an SPKI cache so this cache can distribute them to BGPsec routers. The private keys are needed by those routers to sign updates. With the *KeyGen*-tool, arbitrary amounts of keys-pairs can be generated using a command like this:

```
$ ./keygen 1000 keys/
Generated 1000 router key(s) in directory keys/
```

This command generates 1000 key-pairs and puts them in the `keys` directory.

7.2.2 SPKI Cache

Currently, there are no cache servers out in the wild that serve SPKI data. A solution to this problem is having an own minimalistic SPKI cache. *SPKICache* is a script that mimics an RPKI cache that serves SPKI data. It reads public keys from a target location, extracts their SPKI data and SKI and creates router key PDUs with them. All PDUs are then stored in memory. When a router establishes a connection to SPKICache using the RTR protocol, the cache delivers all router key PDUs, followed by a End of Data PDU. The End of Data PDU sets maximum values for refresh, retry and expiration intervals to prevent the peering router from making subsequent connection attempts during testing. SPKICache then closes the connection. The router now has all the SPKI data stored in its SPKI table and can use it to validate BGPsec updates. To run SPKICache, the following command is executed:

```
$ ./cache.py 172.18.0.2 8383 --keypath keys/
Loading keys... done
Successfully loaded 1000 Router Keys
```

Using this command, SPKICache runs on the IP address `172.18.0.2` and port `8383`. It loads public keys from the `keys/` location. After loading all keys, it waits for incoming connections. SPKICache is not a complete implementation of the RTR protocol. It merely sends those messages that are required to establish a connection, send PDUs and terminate the connection. The tool also does not serve any IP prefix PDUs as they are not relevant to the test scenarios in question.

7.2.3 Dummy Router

FRR and BGP-SRx are not supposed to send erroneous or malformed messages. Hence, testing the processing of malformed BGPsec updates requires for a tool that sends such updates to the routing daemons. The *Dummy Router* is a tool that establishes a BGP session and sends BGPsec messages with arbitrary contents to its peer.

Aside from this basic behavior, the Dummy Router does not provide any other functionalities and does not implement a complete BGP finite state machine. Its only purpose is to send BGPsec messages to a router.

```
$ ./dummyrouter.py 172.18.0.200 8484 172.18.0.3 179 upd.bin
```

Starting the router with these parameters launches it on IP address `172.18.0.4` and port `8484`. It connects to `172.18.0.3` on the BGP port `179`. After successfully establishing a connection, Dummy Router sends BGPsec messages to its peer. They are stored in binary format in `upd.bin`.

7.3 Correctness of the Implementation

One way to assure a correct implementation of BGPsec is to let FRR peer with another BGPsec implementation. For all tests, the reference implementation, BGP-SRx, is chosen. FRR sends messages to and receives messages from BGP-SRx. Additionally, malformed and invalid BGPsec updates are sent to FRR via Dummy Router to check for a correct error handling of the implementation. The expected behavior is then compared with the actual results.

7.3.1 Open Message

Regarding the open message, a BGPsec implementation needs to make sure that the contents of the newly added BGPsec capability are properly handled. The new capability was introduced in Section 4.2. It contains the BGPsec version, a direction and an AFI.

The implementation needs to handle the following error cases for the BGPsec capability.

1. The BGPsec version is not supported
2. The AFI is not supported

In both cases, those capabilities are ignored, as specified by the RFC. The direction is not an issue as it is a single bit and both 0 and 1 are valid values. To make sure that error cases are handled properly and correct capabilities are applied to a running BGP instance, the following test scenario covers all cases.

Figure 7.1 depicts two test cases. In a), a peering between an FRR and a Dummy Router instance is established. The latter sends open messages to FRR that contain malformed BGPsec capabilities. All malformed BGPsec capabilities are expected to be ignored by FRR.

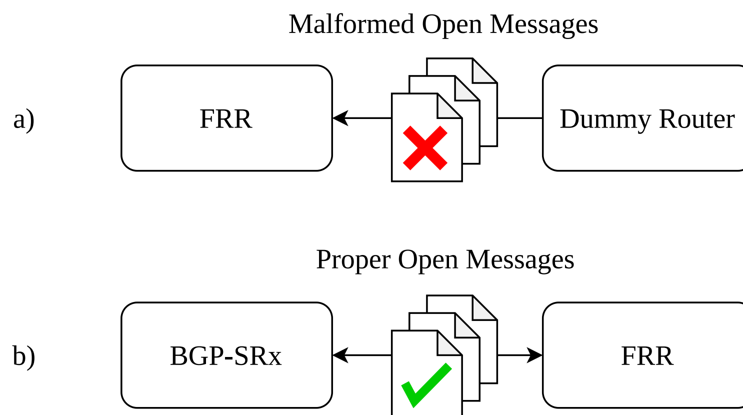


Figure 7.1: Testing open messages. In a), malformed BGPsec capabilities are sent from a Dummy Router instance to FRR. In b), proper open capabilities are exchanged between FRR and BGP-SRx. These capabilities contain various combinations of direction and AFI.

If the implementation is correct, it is further expected that FRR neither accepts nor sends any BGPsec updates during this session (since BGPsec update processing was not tested yet, it is merely tested *if* BGPsec updates are sent or received by FRR).

In b), an FRR and a BGP-SRx instance exchange open messages, this time with valid contents. Various cases for sending and receiving BGPsec updates are considered. If, e.g., BGP-SRx is configured to not accept BGPsec updates, FRR must not send any BGPsec updates to BGP-SRx (again, the mere process of sending and receiving of BGPsec updates is indicator for a working implementation). By configuring FRR and BGP-SRx accordingly, different cases are covered. It is expected that FRR only sends messages according to its own configuration and the one of BGP-SRx.

In scenario a), the BGP open messages sent by Dummy Router contain BGPsec capabilities with values that are not supported by the current BGPsec specification. For once, the version number is set to 1 instead of 0. Second, the AFI are set to 3 and 4 instead of 1 and 2. In both cases, FRR is expected to ignore these capabilities but still establishes a BGP session with Dummy Router. Later during the session, Dummy Router attempts to send a valid BGPsec update. Since FRR is supposed to ignore the capabilities sent by Dummy Router, it is expected to not accept the BGPsec update.

```
1 BGPsec Cap: Received unsupported BGPsec version 1
2 Ignoring BGPsec capability from peer 172.18.0.200
3 ...
4 172.18.0.200 sent BGPsec UPDATE, but capabilities are not set for AFI IPv4
5 172.18.0.200: Attribute BGPSEC_PATH, parse error
6 BGP UPDATE receipt failed for peer: 172.18.0.200
```

Listing 7.1: FRR log when receiving a BGPsec capability with BGPsec version 1, followed by a BGPsec update message.

Listing 7.1 shows the FRR log when receiving a BGPsec capability which contains an unsupported BGPsec version (line 1 and 2). FRR still establishes a session and accepts update messages. If Dummy Router (172.18.0.200) sends a BGPsec update though, FRR logs that no appropriate capabilities are set and rejects the update (line 4 to 6).

The same behavior is observed when receiving a BGPsec capability containing an unsupported AFI. Listing 7.2 shows the FRR log for this case.

```
Received invalid AFI 3 in BGPsec capability from peer 172.18.0.200
Ignoring BGPsec capability from peer 172.18.0.200
Received invalid AFI 4 in BGPsec capability from peer 172.18.0.200
Ignoring BGPsec capability from peer 172.18.0.200
...
172.18.0.200 sent BGPsec UPDATE, but capabilities are not set for AFI IPv4
172.18.0.200: Attribute BGPSEC_PATH, parse error
BGP UPDATE receipt failed for peer: 172.18.0.200
```

Listing 7.2: FRR log when receiving a BGPsec capabilities with AFI 3/4, followed by a BGPsec update message.

A warning is logged (line 1 to 4) but FRR still establishes a session and accepts BGP update messages, like it is supposed to. Incoming BGPsec updates are rejected, though (line 6 to 8).

In both cases, no BGPsec updates were processed and no paths were added to the BGP table. After connecting to the BGP daemon via *vtysh*, the `show ip bgp` command confirms an empty BGP table. Listing 7.3 shows that there are no prefixes to display, thus FRR successfully rejected the BGPsec updates sent by Dummy Router.

```
vtysh# show ip bgp
No BGP prefixes displayed, 0 exist
```

Listing 7.3: FRR instance that is not configured to accept BGPsec updates. When connecting to the BGP daemon via *vtysh* after receiving BGPsec updates, the `show ip bgp` command has no prefixes to display.

To further show that FRR itself did not sent any BGPsec updates, Wireshark[3] is used to capture the BGP session between FRR and Dummy Router. Figure 7.2 shows the package exchange after invalid BGPsec capabilities were sent[4]. FRR sends plain BGP updates to Dummy Router, indicated by the AS PATH attribute present within the path attributes.

[3]https://www.wireshark.org/
[4]Wireshark was extended with a BGPsec parser for this work

```
Source        Destination   Prot Leng Info
172.18.0.200  172.18.0.2    BGP  176 OPEN Message
172.18.0.2    172.18.0.200  BGP  176 OPEN Message
172.18.0.200  172.18.0.2    BGP   85 KEEPALIVE Message
172.18.0.2    172.18.0.200  BGP   85 KEEPALIVE Message
172.18.0.2    172.18.0.200  BGP  144 UPDATE Message, UPDATE Message
172.18.0.200  172.18.0.2    BGP   89 UPDATE Message
172.18.0.200  172.18.0.2    BGP   87 NOTIFICATION Message

Frame 12: 144 bytes on wire (1152 bits), 144 bytes captured (1152
Ethernet II, Src: 02:42:ac:12:00:02 (02:42:ac:12:00:02), Dst: 02:
Internet Protocol Version 4, Src: 172.18.0.2, Dst: 172.18.0.200
Transmission Control Protocol, Src Port: 179, Dst Port: 55536, Se
Border Gateway Protocol - UPDATE Message
  Marker: ffffffffffffffffffffffffffffffff
  Length: 55
  Type: UPDATE Message (2)
  Withdrawn Routes Length: 0
  Total Path Attribute Length: 28
  Path attributes
    Path Attribute - ORIGIN: IGP
    Path Attribute - AS_PATH: 64496
    Path Attribute - NEXT_HOP: 172.18.0.2
    Path Attribute - MULTI_EXIT_DISC: 0
  Network Layer Reachability Information (NLRI)
Border Gateway Protocol - UPDATE Message
```

Figure 7.2: Wireshark capture of a BGP session. The update include the AS PATH attribute, hence it must be a plain BGP update.

In Figure 7.3, valid BGPsec capabilities were sent by Dummy Router. Thus, FRR did send BGPsec updates, which is indicated by the BGPsec PATH attribute present within the path attributes of the update.

```
Source        Destination   Prot Leng Info
172.18.0.200  172.18.0.2    BGP  176 OPEN Message
172.18.0.2    172.18.0.200  BGP  176 OPEN Message
172.18.0.2    172.18.0.200  BGP   85 KEEPALIVE Message
172.18.0.200  172.18.0.2    BGP   85 KEEPALIVE Message
172.18.0.2    172.18.0.200  BGP  248 UPDATE Message, UPDATE Message
172.18.0.200  172.18.0.2    BGP   89 UPDATE Message
172.18.0.200  172.18.0.2    BGP   87 NOTIFICATION Message

Frame 15: 248 bytes on wire (1984 bits), 248 bytes captured (1984
Ethernet II, Src: 02:42:ac:12:00:02 (02:42:ac:12:00:02), Dst: 02:
Internet Protocol Version 4, Src: 172.18.0.2, Dst: 172.18.0.200
Transmission Control Protocol, Src Port: 179, Dst Port: 55522, Se
Border Gateway Protocol - UPDATE Message
  Marker: ffffffffffffffffffffffffffffffff
  Length: 159
  Type: UPDATE Message (2)
  Withdrawn Routes Length: 0
  Total Path Attribute Length: 136
  Path attributes
    Path Attribute - MP_REACH_NLRI
    Path Attribute - ORIGIN: IGP
    Path Attribute - MULTI_EXIT_DISC: 0
    Path Attribute - BGPsec_PATH
Border Gateway Protocol - UPDATE Message
```

Figure 7.3: Wireshark capture of a BGPsec session. The BGPsec PATH attribute is included within the update.

7.3.2 Malformed Update Message

To make sure that a BGPsec update is correct, the specification holds a list of checks that need to be made before the update can be deemed coherent. The summarized list is shown below. A BGPsec update is correct if:

- its last prepended AS is equal to the AS within the open message
- it has an even amount of Signature/Secure Path Segments
- it does not contain both an AS PATH and a BGPsec PATH attribute
- none of its pCount fields are equal to 0
- it contains no AS loops
- its algorithm ID is valid

In case one of the checks fails, the update is treated with a treat-as-withdraw approach. A special case occurs on an unsupported algorithm ID. If so, the update is converted into a plain BGP update, instead of going with the treat-as-withdraw approach. Note that an unsupported algorithm ID is different from an invalid algorithm ID. Invalid IDs are 0 and 255, while unsupported IDs have values ranging from 2 to 254. Currently, the only supported ID is 1 [69].

To test the various errors cases, forged BGPsec updates are generated, one for each error that needs to be checked. Each of those updates are constructed in a way that they trigger one of the errors. As shown in Figure 7.4, Dummy Router sends those forged updates to FRR. The expected result is that FRR would always respond with a withdraw to every malformed BGPsec update. This can be verified by looking at the debugging output of FRR.

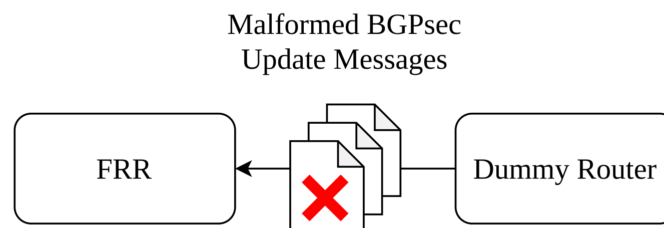


Figure 7.4: Dummy Router sends malformed BGPsec updates to FRR.

By logging the reason for an error it is possible to demonstrate the correctness of the implementation. Listing 7.4 shows all errors that were produced by sending malformed BGPsec updates to FRR.

```
172.18.0.200 sent invalid BGPsec UPDATE (last AS (666) does not match
             peer AS (123))
172.18.0.200: Attribute BGPSEC_PATH, parse error
...
172.18.0.200 sent invalid BGPsec UPDATE (uneven amount of segments)
172.18.0.200: Attribute BGPSEC_PATH, parse error
...
172.18.0.200 sent invalid BGPsec UPDATE (contains AS_PATH and BGPsec_PATH)
172.18.0.200: Attribute BGPSEC_PATH, parse error
...
172.18.0.200 sent invalid BGPsec UPDATE (encountered pCount value 0)
172.18.0.200: Attribute BGPSEC_PATH, parse error
...
1.2.3.0/24 IPv4 unicast -- DENIED due to: as-path contains our own AS;
...
172.18.0.200 sent invalid BGPsec UPDATE (invalid algorithm ID 0)
172.18.0.200: Attribute BGPSEC_PATH, parse error
```

Listing 7.4: FRR reacts to every error and logs the reason.

The check for AS loops was already implemented by FRR and could be reused for the BGPsec implementation. To further prove that no updates were accepted, the BGP table is inspected in Listing 7.5.

```
vtysh# show ip bgp
No BGP prefixes displayed, 0 exist
```

Listing 7.5: After receiving multiple malformed BGPsec updates, the BGP table is still empty.

With no prefixes to display, none of the received BGPsec updates were processed after the errors occurred. The result meets the expectations.

7.3.3 Parsing Update Message

To show that FRR validates BGPsec updates according to the specification, FRR needs to validate BGPsec updates which it received from BGP-SRx. To further show that the signing routine works as intended, BGP-SRx needs to successfully validate BGPsec updates that were signed by FRR. Figure 7.5 shows a scenario in which BGPsec update processing (generation and parsing) can be verified over multiple hops.

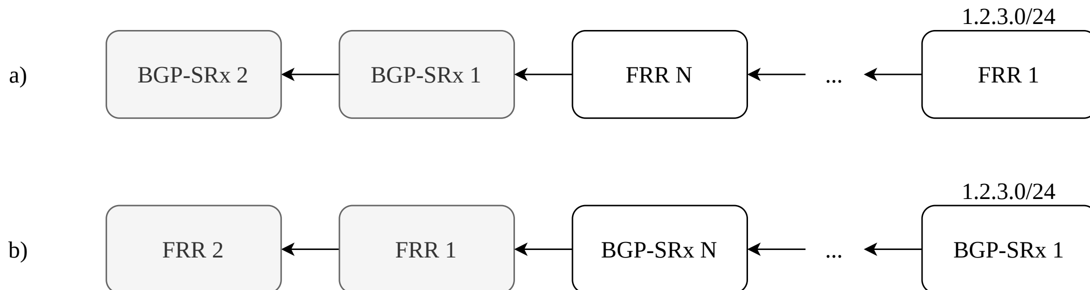


Figure 7.5: BGPsec update correctness test. In a), the signing routine is evaluated. BGP-SRx 1 and 2 must successfully validate the BGPsec update forwarded by FRR N. Scenario b) evaluates validation. A BGPsec update forwarded by BGP-SRx N must be successfully validated, this time by FRR 1 and 2.

An arbitrary amount of AS hops is achieved by chaining multiple router instances together. In a), the signing routine of FRR is verified. FRR 1 originates a BGPsec update with the prefix *1.2.3.0/24*. It is forwarded until it reaches FRR N. At this point, the BGPsec update has a path length of N-1. FRR N then forwards the BGPsec update to BGP-SRx 1. If it is successfully validated there, it is forwarded to BGP-SRx 2. This ensures that the update successfully passes the validation on hop farther. If BGP-SRx 2 successfully validates the update, the test is complete.

In b), the validation routine of FRR is verified. This time, BGP-SRx and FRR instances are switched. BGP-SRx does the forwarding from BGP-SRx 1 to N. FRR 1 and 2 then validate the BGPsec update. If both FRR 1 and 2 successfully validate the BGPsec update, the validation routine in FRR works as intended.

After executing a), BGP-SRx 2 must hold a validated path to the prefix announced by FRR 1. This can be verified by accessing the BGP daemon and invoking the *show ip bgp* command. It outputs the following lines (truncated):

```
bgpsrx# show ip bgp
   Ident      SRxVal   [...]  Network     [...]  Path
*> 41045A66   i(n,v)          1.2.3.0/24          7 6 5 4 3 2

Total number of prefixes 1
```

Listing 7.6: *show ip bgp* command invoked on an BGP-SRx instance after validating a BGPsec update originated by an FRR instance. The *v* in the highlighted part in line 3 tells that the BGPsec path is valid.

The highlighted part in Listing 7.6 shows the validation result. The inner values of the braces signal the validation result of the ROV (left) and the AS path validation (right). Since origin validation is not performed, it defaults to *n* - not found. To its right, *v* - valid - signals that the AS path validation was successful. Combining both values, BGP-SRx determines that the overall result is invalid, indicated by the *i*. However, this has no implications on the functionality of the BGPsec implementation.

Vice versa, FRR needs to be able to validate BGPsec updates that were originated by BGP-SRx. FRR has currently no other way of displaying the validity status of updates other than checking the debugging output. In combination with the BGP table output, it is possible to verify the correctness of update processing. Listing 7.7 shows the debugging output, together with the *show ip bgp* command.

```
---- BGP daemon
[...]
BGPSEC: Validation result of signature: valid
BGPSEC: Validation result of signature: valid
BGPSEC: Validation result of signature: valid
BGPSEC: Validation result of signature: valid
BGPSEC: Validation result of signature: valid
BGPSEC: Validation result of signature: valid
BGPSEC: Validation result for the whole BGPsec_PATH: valid
BGPSEC: 172.18.0.6 All signatures are valid.
[...]

---- vtysh
frr# show ip bgp
   Network     [...]  Path
*> 1.2.3.0/24         6 7 8 9 10 11

Displayed  1 routes and 1 total paths
```

Listing 7.7: *show ip bgp* command invoked on an FRR instance after validating a BGPsec update originated by a BGP-SRx instance.

Line 1 to 11 show the output that is logged by the BGP daemon. All signatures of the BGPsec update are valid. Line 16 shows the BGP table which contains the expected entry.

This test has shown that FRR is able to generate and send/forward valid BGPsec updates. It is also capable of correctly processing incoming BGPsec updates.

7.3.4 Update Reconstruction

In some cases, a BGPsec update may be received, yet forwarding it as a BGPsec update is not possible. This is most likely because no other BGPsec speaking peers are available. If so, the BGPsec update needs be transformed into a plain BGP update before it can

be forwarded. A test scenario to ensure this behavior works as intended is constructed in Figure 7.6.

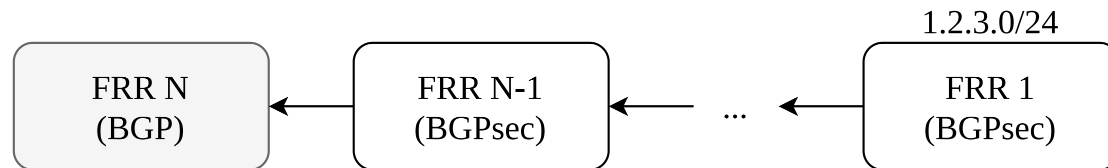


Figure 7.6: Test scenario for BGPsec update reconstruction. Because FRR N does not speak BGPsec, FRR N-1 needs to reconstruct a plain BGP update from the BGPsec update.

To make sure that reconstruction works for BGPsec paths with a length greater than 1, N-1 BGPsec routers are chained together prior to the reconstruction. FRR N is not speaking BGPsec, thus FRR N-1 strips the BGPsec PATH attribute from the update and inserts the AS PATH instead. To verify that this procedure is working as intended, the BGP table of FRR N needs to contain the prefix and path announced by FRR N-1. Listing 7.8 shows the BGP table of FRR N.

```
1  frr# show ip bgp
2     Network     [...]  Path
3  *> 1.2.3.0/24         5 4 3 2
4
5  Displayed  1 routes and 1 total paths
```

Listing 7.8: *show ip bgp* command invoked on a plain BGP router instance. It received the announcement from a BGPsec peer. That peer reconstructed a BGPsec update into a plain BGP update prior to forwarding it.

Seeing that the BGP table holds the correct prefix and path shows that the update reconstruction was successful.

7.4 Memory Usage Performance

BGPsec adds a significant performance overhead due to cryptographic operations. To show this overhead, different tests measure the performance of FRR and BGP-SRx.

When comparing the performance of FRR and BGP-SRx, or rather Quagga, it must be kept in mind that FRR is being actively developed on, while Quagga is hardly getting any updates. To keep the comparison as fair as possible, only routines that are affected by BGPsec are compared against each other.

BGPsec comes with two major kinds of memory consumptions. First, there are router keys. They are fetched from an SPKI cache and stored in memory to be quickly available to the application. Second, there are BGPsec paths. They must be stored in memory, so when a BGPsec announcement is forwarded, it is ready to be signed.

Three measurements are undertaken: pure memory consumption of router keys, of BGPsec paths and the total amount of memory the routing suites occupy using BGPsec. Memory consumption of router keys and BGPsec paths is estimated prior to executing the tests. The estimates are compared to the test results.

For router keys, a starting sample size of 100 keys is taken. With each iteration, the amount increases by another 100 keys, finishing at 1,000 keys. For BGPsec paths, 500 paths are taken, increasing the amount by 500 with each iteration, stopping at 5,000. For each iteration, a path length up to 5 hops is measured. Last, the total memory consumption of FRR and BGP-SRx are compared, storing 100 router keys and up to 5,000 BGPsec updates of fixed length.

The memory in use is gathered utilizing the internal memory tracking of the routing suites. As soon as the test is executed, e.g., storing 100 router keys, a connection via a command line interface to the BGP daemon is established. The *show memory* command displays memory information of the running daemon. The total amount of memory is written down. The amount of memory storing no router keys is subtracted from the amount when storing 100 keys. This yields the total amount of memory the 100 router keys take.

$$\text{Daemon memory storing 100 keys} - \text{Daemon memory without keys} = \text{Memory of 100 keys}$$

When processing BGPsec updates, their contents are translated to plain BGP internally. Thus, each BGPsec update additionally allocates memory for plain BGP structures. To get the amount of memory that is allocated for BGPsec only, the memory required for plain BGP updates must first be measured and must then be subtracted from the amount of memory the BGPsec updates occupy. The following formula summarizes the approach:

$$\text{BGPsec total} - \text{BGP total} - \text{router keys} = \text{BGPsec paths}$$

- BGPsec total: total daemon memory storing a fixed amount of BGPsec paths.
- BGP total: total daemon memory storing the same amount of BGP AS paths.
- Router Keys: memory router keys occupy.
- BGPsec paths: memory that was allocated purely for BGPsec paths.

This process is repeated for different path lengths. Subtracting the memory for path length 1 from the memory for path length 2 results in the memory that is allocated per additional AS hop.

This rather simple approach provides results that can be used to estimate the memory usage for an arbitrary amount of router keys or BGPsec paths with great precision.

7.4.1 Router Keys

Router keys have a fixed size of memory and are defined by three information: the SKI (20 bytes), the ASN (4 bytes) and the SPKI (91 bytes), adding up to 115 bytes. Additional memory for storing these information, e.g. linked lists or tables, is required. On top of that, meta data or debugging information such as "stringified" values of the keys may be useful to the developer.

In case of RTRlib, which handles the SPKI storage for FRR, the TCP socket from where a router key PDU was received from is tracked for each key. The keys are then stored inside a linked list. Counting in all this additional memory yields that a router key in RTRlib requires 215 bytes of memory. Using this value, an estimate for an arbitrary amount of router keys can therefore be calculated using the following formula:

$$keys \cdot 215$$

Where *keys* is the amount of router keys.

BGP-SRx occupies 1,859 bytes per router key, which is roughly 8 times the amount compared to RTRlib. BGP-SRx stores router keys in a sophisticated key storage. The storage holds an element for each loaded router key. The elements form a linked list, each one holding key data such as redundant copies of ASN and SKI, various flags and an OpenSSL formatted copy of the key. Storing the keys in OpenSSL structures comes with a high memory overhead but yields serious performance advantages when it comes to validating and signing BGPsec updates, as Section 7.6 will show.

Calculating the estimated amount of required memory for BGP-SRx router keys can be achieved by multiplying the memory per key with the amount of keys.

$$keys \cdot 1,859$$

Table 7.1 compares RTRlib and BGP-SRx router keys and displays the overhead compared to the minimum required key space of 115 bytes.

	RTRlib	**BGP-SRx**
Router Key	215 bytes	1,859 bytes
Overhead	100 bytes	1,744 bytes

Table 7.1: Memory required for storing a single router key in RTRlib and BGP-SRx. The minimum is 115 bytes. RTRlib allocates an additional 100 bytes per key, while BGP-SRx requires 1,744 bytes of extra memory per key.

Using the formulas from this section, estimates can be calculated on how the memory growth should behave. They are directly compared to the actual measured values in Figure 7.7.

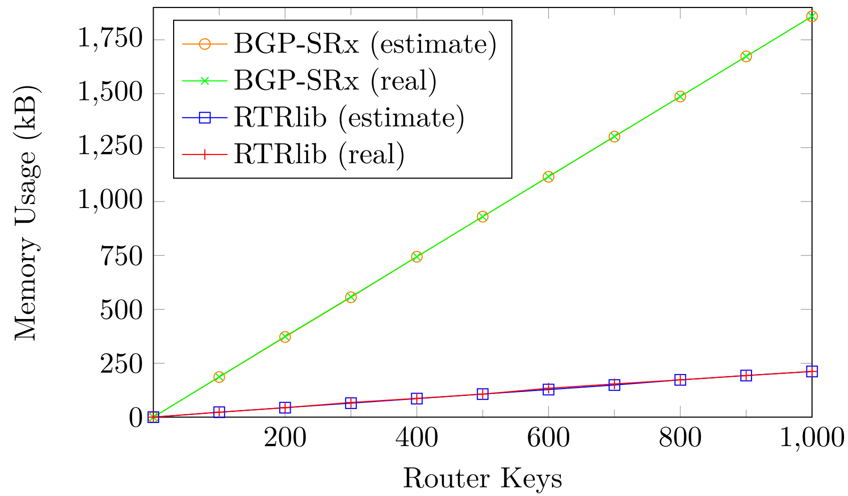


Figure 7.7: Memory usage of router keys for RTRlib and BGP-SRx. Between 100 and 1,000 router keys have been examined, all are of same size.

The estimates for both RTRlib and BGP-SRx are almost identical to the actual measured values. A linear behavior for storing router keys can be observed for both implementations. As expected, the memory growth for BGP-SRx is approximately 8x higher compared to RTRlib.

Figure 7.8 shows the percentage of the total memory usage of FRR/BGP-SRx in relation to the space 1,000 router keys take. In this example, routers were not peering and thus did not exchange any messages to ensure no additional memory was allocated. FRR requires a base amount of 10.77 MB of memory. Adding 1,000 router keys consumes another 0.23 MB of memory. The base amount of memory for BGP-SRx is 1.97 MB. The 1,000 router keys add another 1.86 MB, almost doubling it.

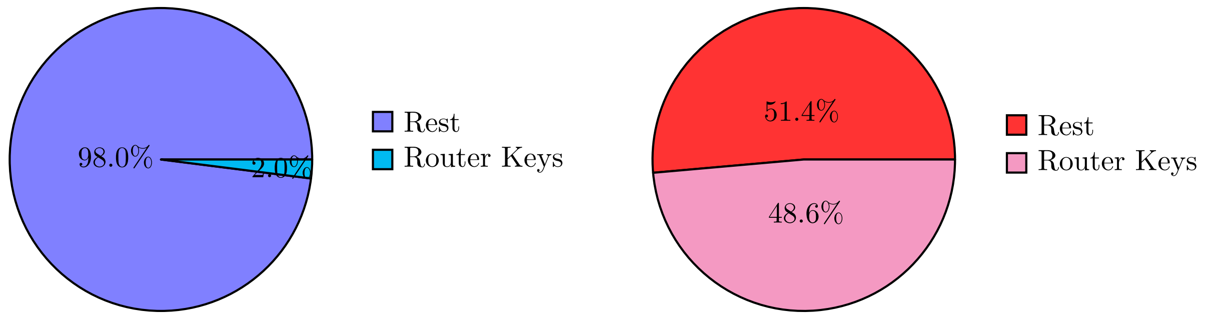


Figure 7.8: Left: total memory usage of FRR with 1,000 stored router keys. Router keys occupy 2% of it. Right: same scenario using BGP-SRx. 48.6% of the total memory is occupied by the 1,000 router keys.

FRR requires almost 5x more base memory than BGP-SRx does. This is mainly due to the lively development process on FRR. New features and dependencies are added constantly, resulting in a steady growth of the code base. With new features and capabilities, the increase of memory comes naturally. Compared to the vast amounts of available memory on modern hardware, however, 10 MB are insignificant.

In conclusion, router keys in BGP-SRx require 8x the memory per key compared to FRR. The difference is explained with different approaches of storing router keys. BGP-SRx wraps structures around router keys and transforms them into OpenSSL `EC_KEY`s. FRR takes a different approach and tries to keep the overhead as minimal as possible. The memory growth is linear for both routing suites. The exact size of required space can be reliably calculated using simple formulas.

7.4.2 BGPsec Paths

According to the BGPsec specifications, a BGPsec PATH of length 1 is between 107 and 109 bytes in size [2, 42]. The variance is due to the variable signature length. Each hop adds an additional 98 to 100 bytes. Additional hops take less memory because initial data structures do not need to be allocated again. The data of additional hops, i.e., Signature and Secure Path Segments are appended to existing BGPsec PATH attributes. Using these values, a formula to calculate the maximum size of a BGPsec PATH attribute for any amount of updates and hops can be constructed.

$$paths \cdot (109 + (100 \cdot (pathlen - 1)))$$

As an example, the size of 5,000 BGPsec updates of length 5 are calculated below.

$$\begin{aligned} 5,000 \cdot (109 + (100 \cdot (5 - 1))) &= 2{,}545{,}000\,\text{bytes} \\ &= 2{,}545\,\text{kB} \end{aligned}$$

This formula helps as a reference for how much overhead the BGPsec implementations add on top of the minimal amount of bytes that need to be stored. Just like with router keys, overhead is inevitable, as during the course of processing BGPsec updates, additional data is generated and stored. For example, each BGPsec PATH attribute in FRR is stored within a data structure that holds information such as the hop count, the prefix and the "stringified" AS path. These information are not part of the BGPsec PATH attribute as standardized by the IETF.

To apply the formula to FRR and BGP-SRx, two parameters must be known: the base amount of memory for a BGPsec path and the amount of memory per additional hop.

The approach from Section 7.4 to figure out the exact amount of memory for a single BGPsec path is applied to FRR. The results show that there are 305 bytes of required memory per BGPsec path of length 1. Each additional hop increases the size by 160 bytes. The formula used to calculate the estimated memory consumption is:

$$paths \cdot (305 + (160 \cdot (pathlen - 1)))$$

Applying the same parameters from the previous calculation yields:

$$\begin{aligned} 5,000 \cdot (305 + (160 \cdot (5 - 1))) &= 4{,}725{,}000\,\text{bytes} \\ &= 4{,}725\,\text{kB} \end{aligned}$$

For FRR, the results show 1,128 bytes per BGPsec path of length 1 and an extra 527 bytes per additional hop. To calculate the estimated memory usage, the following formula can be used:

$$paths \cdot (1,128 + (527 \cdot (pathlen - 1)))$$

To calculate the required memory to store 5,000 BGPsec paths of length 5:

$$\begin{aligned} 5,000 \cdot (1,128 + (527 \cdot (5 - 1))) &= 16{,}180{,}000\,\text{bytes} \\ &= 16{,}180\,\text{kB} \end{aligned}$$

The difference in memory between both implementations is explained with the way BGP-SRx stores BGPsec paths. BGP-SRx wraps BGPsec paths around dedicated structures that allow quick access to path information. The paths are stored in a linked list and additionally in a hash table. Additionally, every digest value that is generated during the validation and signing routines is saved. Having digest values stored allows for quick access of the digest without having to re-hash parts of the BGPsec path, but comes with the price of high memory overhead, as a single hash is 256 bytes alone.

FRR does not wrap BGPsec paths, which potentially slows down processes that need to quickly access certain information. FRR also does not store digest values which saves a lot of memory but requires re-hashing whenever needed.

Memory values for a single BGPsec path and its overhead compared to the minimum required space of 109 bytes is listed in Table 7.2.

	FRR	**BGP-SRx**
BGPsec Path	305 bytes	1,128 bytes
Overhead	196 bytes	1,019 bytes
Additional Hop	160 bytes	527 bytes
Overhead	60 bytes	427 bytes

Table 7.2: Memory overhead for BGPsec paths of FRR and BGPsec. For reference, RFC values for BGPsec path size of 109 bytes base value and 100 bytes per hop are used [3].

Testing memory usage in FRR was executed with a setup described in Figure 7.9. The sample sizes for these test are 500 to 5,000 updates. The same test is repeated with BGP-SRx instances. Both results are then compared against each other.

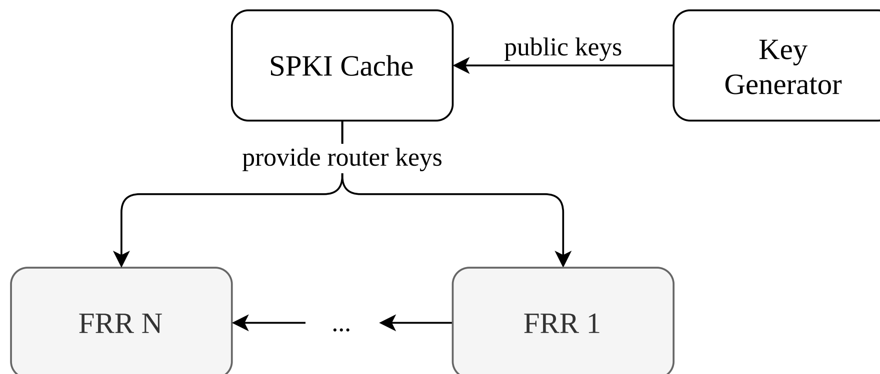
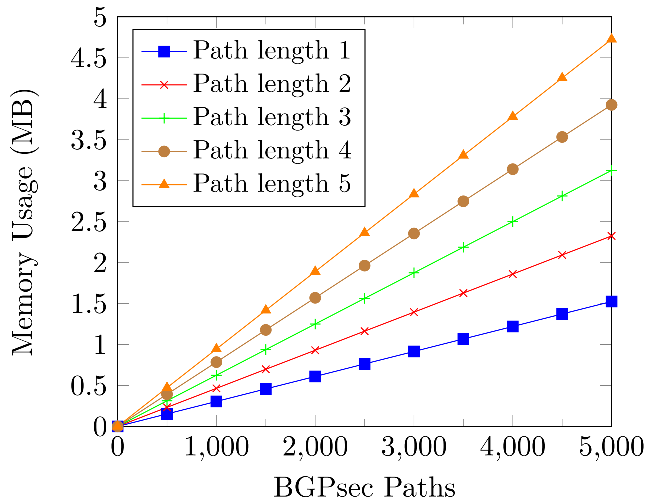
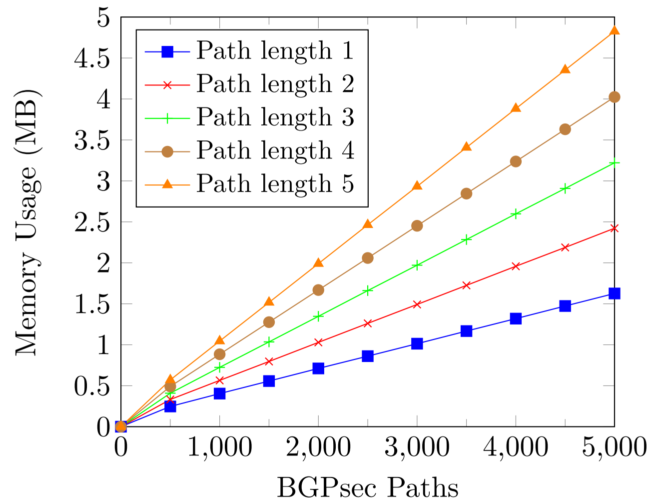


Figure 7.9: Setup for testing memory usage of FRR. To test memory usage for AS paths of length N, there need to be N+1 FRR instances chained together. After validating all updates, the memory usage on every instance is checked via the *show memory* command on the appropriate daemon.

Figure 7.10 compares the calculated estimate to the actual measured memory consumption of BGPsec paths. It shows, how the consumption behaves with increasing amounts of paths and length.



(a) FRR: estimated memory usage of BGPsec paths.

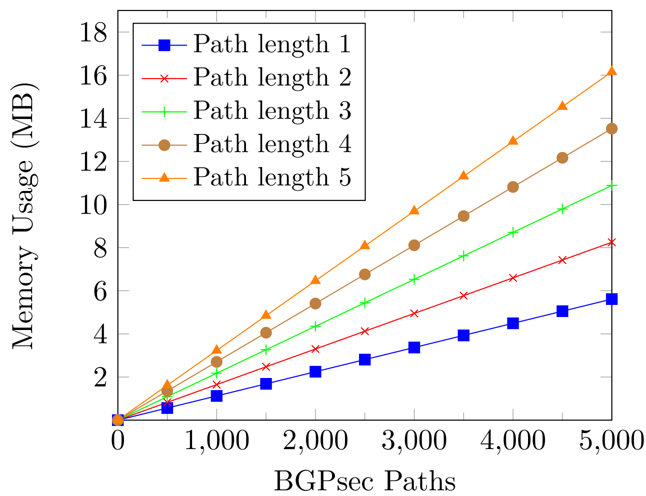


(b) FRR: real memory usage of BGPsec paths.

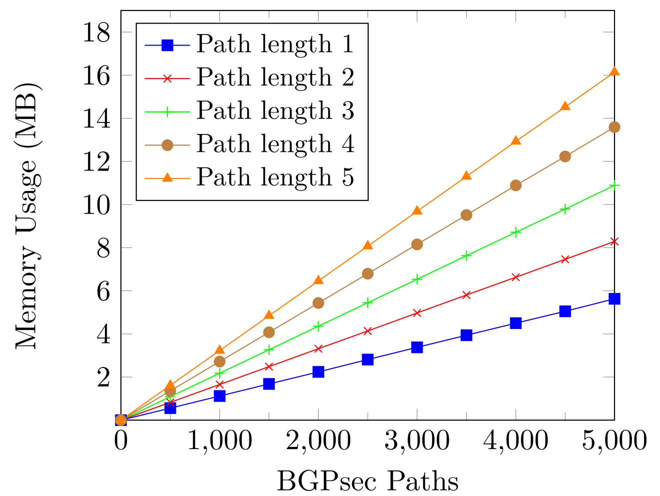
Figure 7.10: Estimated and real memory usage of stored BGPsec paths in FRR. Between 500 and 5,000 BGPsec paths of length 1 to 5 were processed. Estimate (a) and measurements (b) are almost identical. The initial spike in (b) is attributed to BGPsec initialization.

The estimate (a) predicts that BGPsec path length behaves linear for all path lengths. To verify the calculated estimate, the results of the measurements are presented in (b). Both graphs (a) and (b) in Figure 7.10 are close together. In (b), there is a minor initial increase in memory due to BGPsec initialization. This graph confirms the expected linear behavior of memory growth.

The test is repeated with BGP-SRx. As Figure 7.11 shows, the estimates (a) and real results (b) are almost identical.



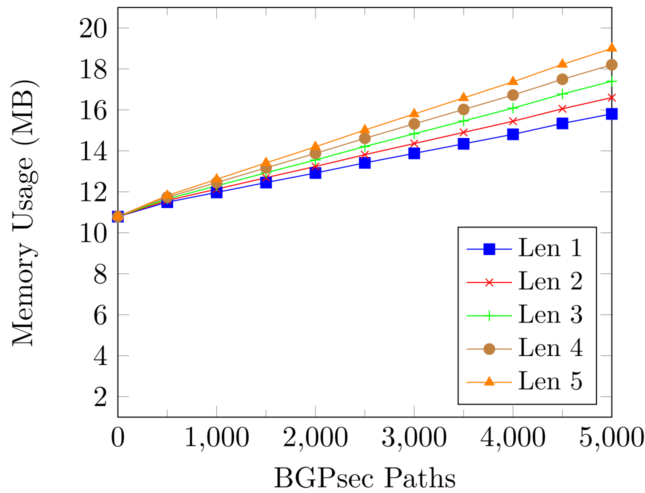
(a) Estimated memory usage.



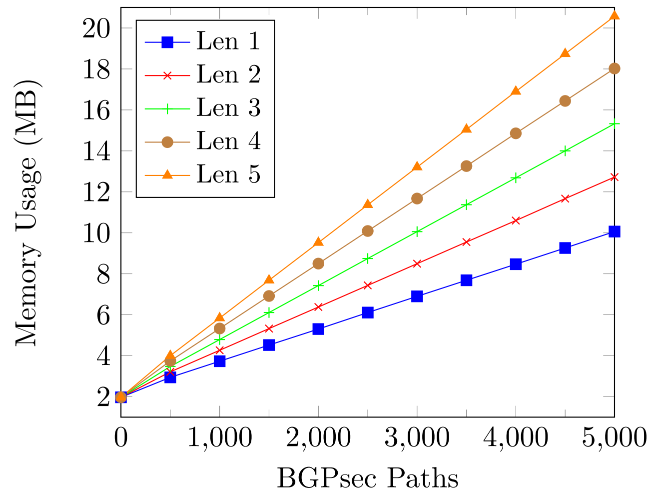
(b) Real memory usage.

Figure 7.11: Estimated and real memory usage of stored BGPsec paths in BGP-SRx. Between 500 and 5,000 BGPsec paths of various lengths were processed. Estimate (a) is almost identical to test result (b).

BGPsec paths in BGP-SRx, although they perform linear, occupy more memory compared to FRR. 5,000 paths of length 5 peak at 16 MB of memory for BGP-SRx, for FRR it is approximately 4.8 MB. Looking at the total memory usage in Figure 7.12, it becomes apparent how the memory growth impacts both routing suites.



(a) Total memory usage of FRR, including router keys.



(b) Total memory usage of BGP-SRx, including router keys.

Figure 7.12: Comparison of memory values for storing between 500 and 5,000 BGPsec paths of length 1 to 5. Although FRR has a higher base value of occupied memory, its memory consumption per path is lower compared to BGP-SRx .

Starting at a base value of approximately 2 MB of memory, BGP-SRx memory usage grows much quicker compared to FRR. BGP-SRx surpasses FRR when storing 4,500 BGPsec paths of length 5. Comparing the peak values of allocated memory, FRR and BGP-SRx differ by approximately 2 MB. Figure 7.13 puts the memory usage of BGPsec paths in relation to the total memory usage.

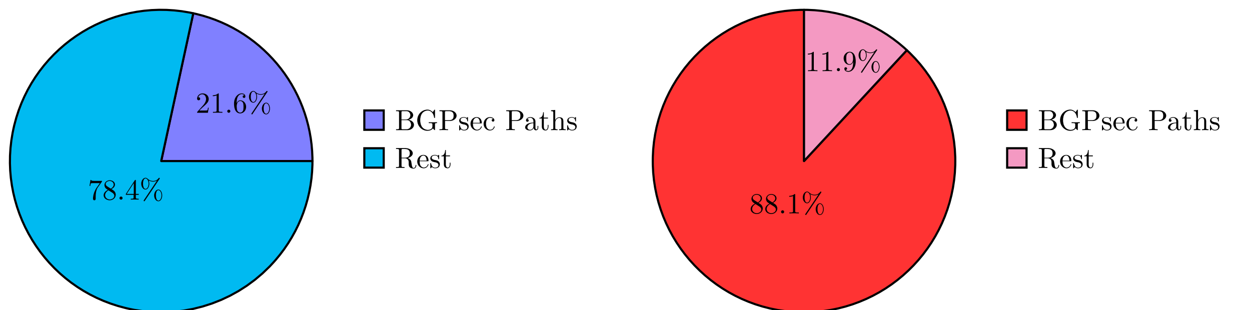
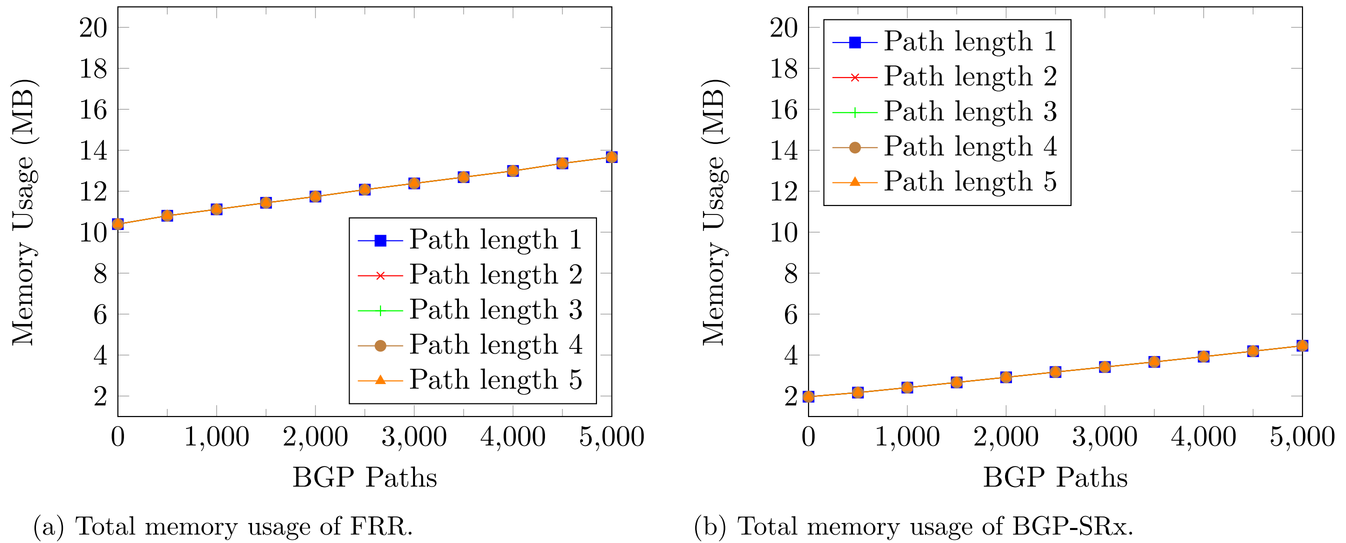


Figure 7.13: Left: memory usage of FRR storing 5,000 BGPsec paths of length 5. The BGPsec paths occupy 21.6% of the total memory, including router keys. Right: same scenario with BGP-SRx. Here, the BGPsec paths occupy 88.1% of the total memory in use.

Due to the high amount of base memory for FRR, the percentage of BGPsec paths do not carry as much weight compared to BGP-SRx. For the latter, most of the occupied memory is attributed to BGPsec path structures. Nevertheless, the memory usage for both routing suites is still relatively low, when compared to modern architecture.

Finally, to so how much BGPsec performs when compared to BGP, Figure 7.14 displays the total amount memory for FRR and BGP-SRx when storing up to 5,000 plain BGP paths.



(a) Total memory usage of FRR.

(b) Total memory usage of BGP-SRx.

Figure 7.14: Total memory usage of FRR and BGP-SRx running plain BGP. Between 500 and 5,000 BGP paths of various lengths were processed. Path length in this test has to visible impact on the memory consumption.

Although plain BGP is also gradually increasing the BGP daemons memory, it scales much better with growing path length. The test shows that there is in fact no difference in memory consumption between a path length of 1 and a path length of 5.

Summarizing this section, memory growth of BGPsec paths can be predicted with high accuracy for both implementations. Both routing suites show a linear memory growth for BGPsec paths across all amounts and lengths, as well as linear growth for router keys. BGP-SRx requires less base memory than FRR. However, memory consumption per router key and BGPsec path is higher compared to FRR. To put the results in perspective, the memory usage of both implementations is far from critical. Memory on

modern architecture is vast. Looking at the server where these tests were executed on, the BGP daemons occupied less than 0.01% of the available memory.

The following sections benchmark the run time of various subroutines that are affected by BGPsec.

7.5 Crypto Microbenchmarks

Prior to measuring the processing time of the BGPsec implementation, microbenchmarks show how much processing time the cryptographic operations occupy. The operations in question are the validation and signing routines of OpenSSL: `ECDSA_verify` and `ECDSA_sign`. Since FRR and BGP-SRx depend on different versions of OpenSSL, tests are repeated for different versions.

The CPU time, i.e., only the time the process is actively running on the CPU, of both OpenSSL functions is measured. This gives a reference to how long the update processing routines in FRR take compared to the cryptographic part. Since only the processing time of the OpenSSL functions are of interest, they can be benchmarked in a separate context. Therefore, a test program was written. The essential parts of the program are shown as pseudocode in Listing 7.9. The entire code can be found in Appendix A.2.

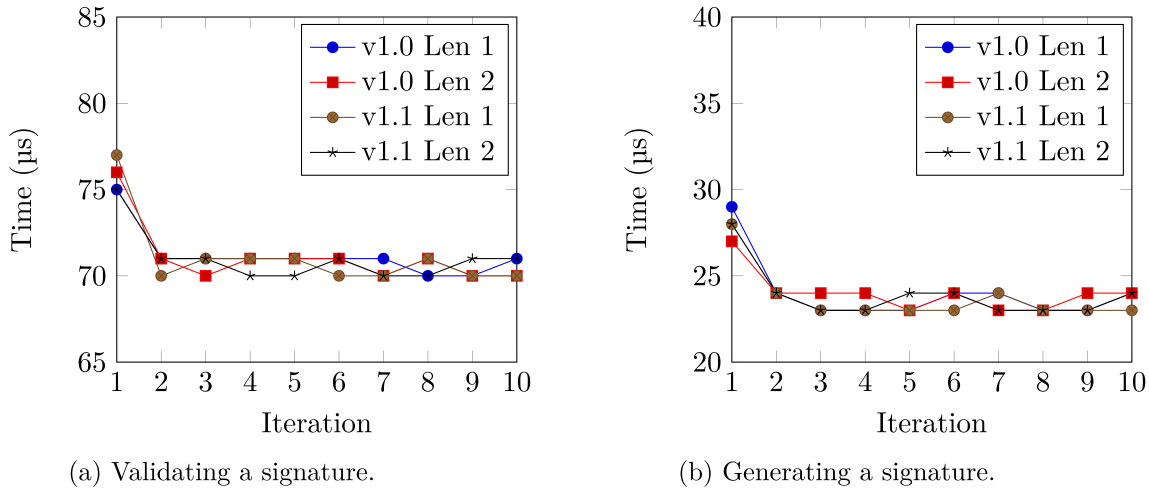
To perform the microbenchmarks, valid signature-hash-router key triples were generated and stored as raw bytes. These three information are passed to `ECDSA_verify` and `ECDSA_sign` respectively.

```
1 signature[] = {0xAB, 0xCD, <...>}
2 hash[] = {0x01, 0x23, <...>}
3 router_key[] = {0x1A, 0x2B, <...>}
4
5 count = 0
6 cpu_time = 0
7 start, end = NULL
8
9 while (count < 10):
10     getrusage(start)
11     ECDSA_verify(signature, hash, router_key)
12     getrusage(end)
13
14     count += 1
15     cpu_time += end - start
16     avg_cpu_time = cpu_time / count
17 done
```

Listing 7.9: The `getrusage` function fetches the CPU time used by this process only. Subtracting `end` from `start` in line 15 is simplified here. It calculates the time it took for `ECDSA_verify` to return. `avg_cpu_time` stores the average duration of the measured function.

By replacing the OpenSSL validation function in Line 12 with `ECDSA_sign`, the signing routine can be measured as well. In this case, `router_key` in line 3 is replaced with a private key.

The routines are executed 10 times in a row, consecutively validating the same signature using the same input. To make sure that the BGPsec path length has no impact on the validation time, the hash message in line 2 was generated from BGPsec paths of length 1 and length 2. They and are exchanged between tests. The test is repeated for signature generation under the same conditions. The results are presented in Figure 7.15.



(a) Validating a signature.

(b) Generating a signature.

Figure 7.15: Microbenchmark of the OpenSSL validation (a) and signing (b) routines. The two OpenSSL version 1.0 and 1.1 were examined. Both routines were executed 10 times in succession with a hash representing a path length of 1 (Len 1) and a path length of 2 (Len 2). The length of the BGPsec path has no impact on the duration since the path is hashed to a fixed length of 256 bytes. The initial spike comes from OpenSSL initialization. The version has to visible impact on the execution time.

The first execution of the validation and signing routines spike due to OpenSSL initialization. These spikes are only occurring once and do not have an impact on the performance of the implementation.

From iteration 2 onwards, both graphs show a constant behavior on average, independent of version and BGPsec path length. They diverge by 1 µs at maximum. The average duration for a signature validation is 71 µs, not counting in the initial spike. Signing takes 24 µs on average. The length of the BGPsec path that was hashed prior to the validation/signing routines does not have an impact on execution time, since the hash has a constant length of 256 bytes.

When looking at the average duration of update generation for FRR and BGP-SRx in Figure 7.16, the duration does not increase significantly with increasing path length. Signatures are generated once for a single update, while validation is an iterative process.

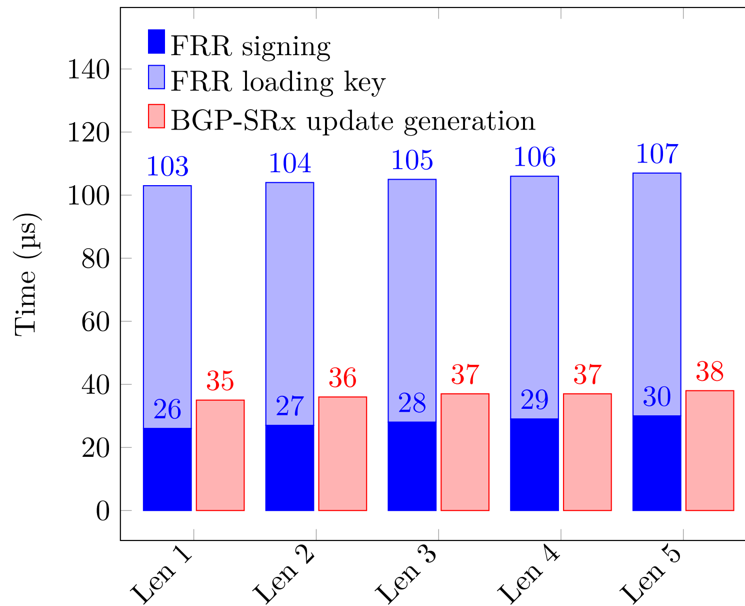


Figure 7.16: Average duration for generating BGPsec updates, 5,000 in total. The average slightly increases with growing path length. FRR measurements are split in signing and key loading. Due to repetitive loading of router keys in FRR, a constant overhead of 77 µs is added on top.

BGP-SRx outperforms FRR when looking at the total time. When ignoring the key loading overhead, FRR manages to perform slightly better than BGP-SRx. The large performance discrepancy comes from loading router keys. The reason is explained further down below.

Due to the iterative process when validating BGPsec updates, a constant growth in processing time is observed with increasing path length. Figure 7.17 shows the average duration when processing updates of length 1 to 5.

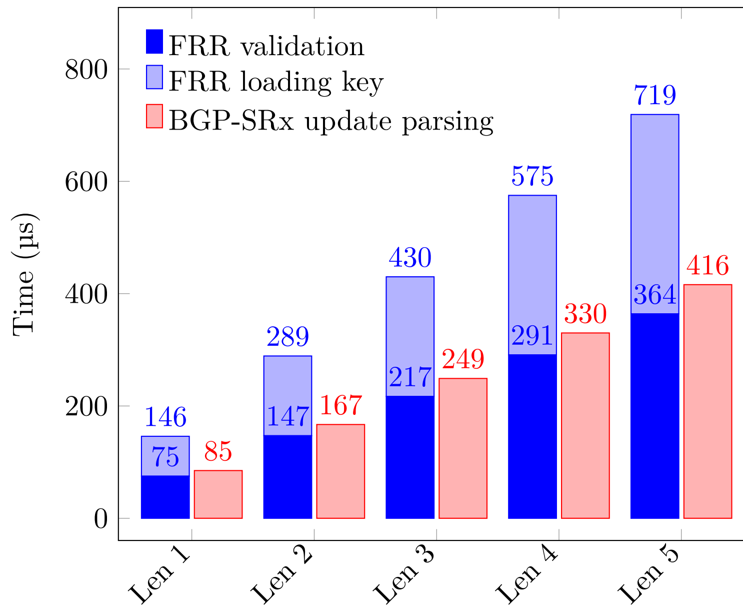


Figure 7.17: Average duration for validating a single BGPsec update. 5,000 BGPsec updates per path length were processed. Results for FRR are split into the validation and key loading part to visualize the overhead.

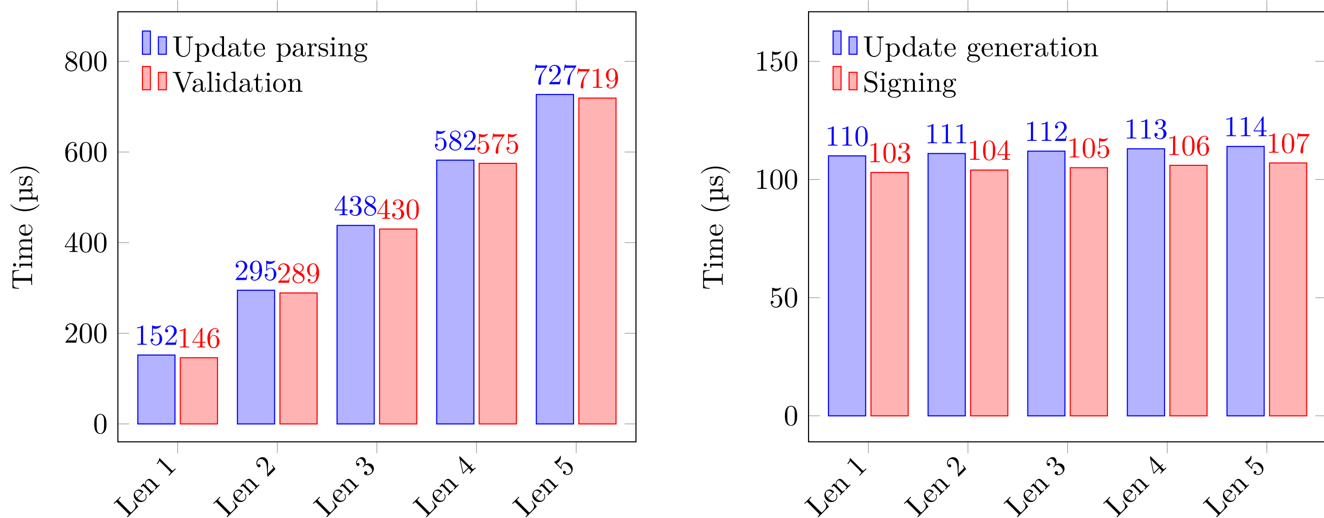
When disregarding the key loading part, FRR performs again slightly better than BGP-SRx. However, the key loading process doubles the processing time for FRR when looking at total values.

Both Figure 7.16 and Figure 7.17 show large performance differences between FRR and BGP-SRx in favor of BGP-SRx. The OpenSSL functions `ECDSA_sign` and `ECDSA_verify` require the router key argument to be an OpenSSL `EC_KEY` structure. OpenSSL provides functions to convert raw byte keys into such a structure. Microbenchmarks reveal that loading keys is a very performance critical task, averaging to 71 µs per public key and 77 µs per private key. This performance issue is amplified with increasing path lengths during validation, as for each signature within a BGPsec PATH, a different key must be loaded.

BGP-SRx already stores public and private router keys in `EC_KEY` structures during BGPsec initialization. Loading the private key with each signature that is generated is therefore not necessary. FRR does not follow the same approach and creates and frees

the EC_KEY each time. This results in a large performance overhead for FRR, tripling the time it takes to generate a BGPsec update compared to BGP-SRx.

Figure 7.18 puts the time it takes for FRR to parse and generate BGPsec updates in relation to the cryptographic part. It turns out that between 94 and 99% of the time is spent performing cryptographic operations.



(a) FRR validation routine in relation to the entire update parsing process.

(b) FRR signing routine in relation to the entire update generation process.

Figure 7.18: Left: FRR validation time in comparison to the whole update parsing routine. 99% of the time parsing the BGPsec update is spent verifying the signatures. Right: Comparing the generation of an entire BGPsec update to the time it takes to generate a signature. 94% of the time is spent generating the signature.

With growing path length, the validation time increases and thus the entire update parsing duration. The time for non-cryptographic tasks does not increase significantly for update generation and parsing.

7.6 Processing Time Performance

The following tests compare BGPsec update processing time of FRR and BGP-SRx. Additionally, processing times for plain BGP updates are also measured to show the

overhead that BGPsec imposes on BGP. The setup from Section 7.4.1 can be reused for this scenario, it is depicted in Figure 7.19.

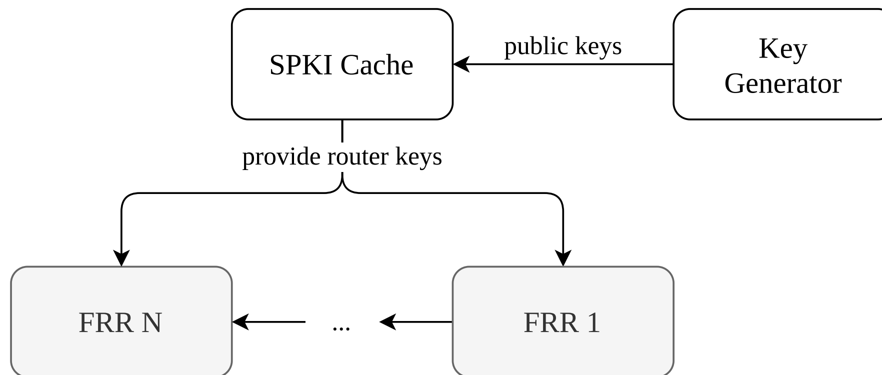


Figure 7.19: Setup for measuring validation and signing processing time. It is identical to the one for memory usage.

The processing time is measured in CPU time. This avoids the scheduler distorting the results. It is expected that signing yields a close to constant performance behavior for variable path length, since signing occurs only once per update, independent of the length of the path. Validation time is expected to grow with increasing path length. However, the performance growth is expected to still behave linear.

Different routines are tracked separately, e.g. the attribute parsing or the validation routine. The processing time of routines that are not touched by the BGPsec implementation, such as best-path-selection, are not regarded in these tests.

The code for measuring the run time of an arbitrary function is displayed in Listing 7.10.

```
1 RUSAGE_T before, after;
2 _Atomic unsigned long total;
3
4 GETRUSAGE(&before);
5 bgp_update_receive(<...>);
6 GETRUSAGE(&after);
7 total += thread_consumed_time(&after, &before);
```

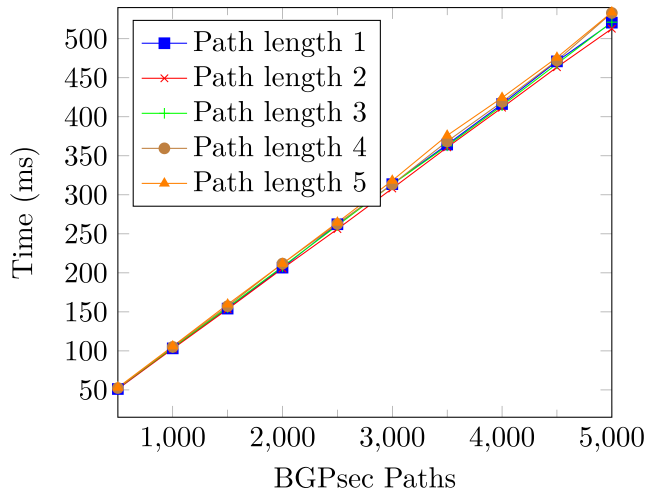
Listing 7.10: Code for measuring the incoming update processing routine. The GETRUSAGE macro is used to get the current CPU time. total is a variable that holds the elapsed CPU time of all bgp_update_receive invocations.

FRR has a macro called GETRUSAGE for getting the CPU time. In line 4 and 6, the macro gets the CPU time before and after the measured function, here it is bgp_update_receive. The code is basically the same as in Listing 7.9, but here, portions of it are wrapped and a helper function exist. The helper function thread_consumed_time calculates system and user time together and returns the difference between before and after (Listing 7.9 simplifies this process). This result is added to total each time the function was invoked.

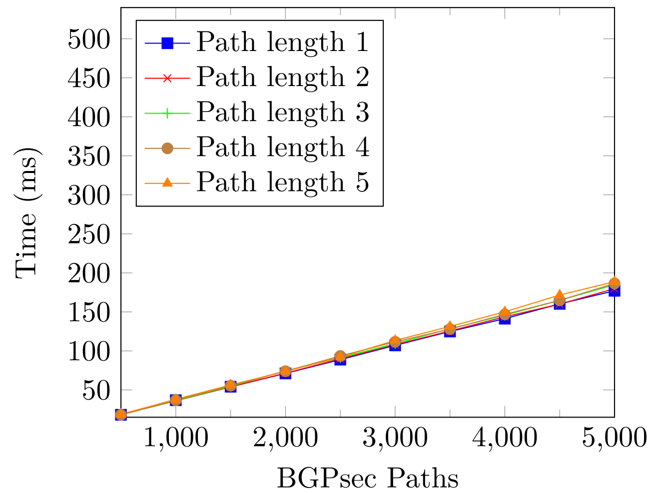
All other FRR and BGP-SRx functions are measured using the same technique, as GETRUSAGE and thread_consumed_time are available to both implementation.

7.6.1 Outgoing Updates

Both routing suites generate BGP(sec) updates via the bgp_packet_attribute function. The following Figure 7.20 compares the results of FRR and BGP-SRx generating BGPsec updates of different length.



(a) Update generation routine of FRR.

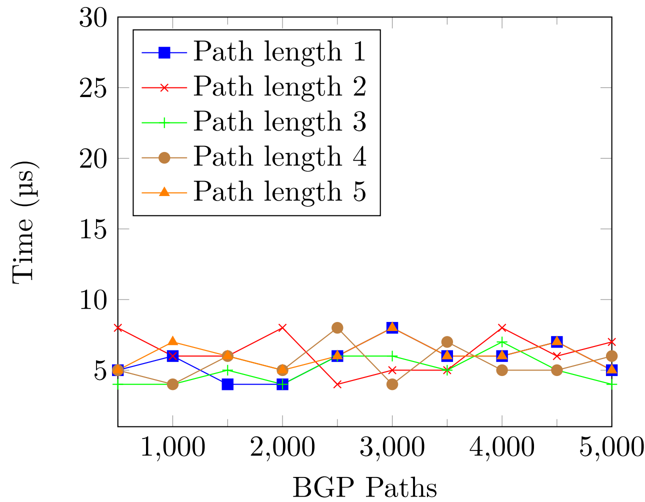


(b) Update generation routine of BGP-SRx.

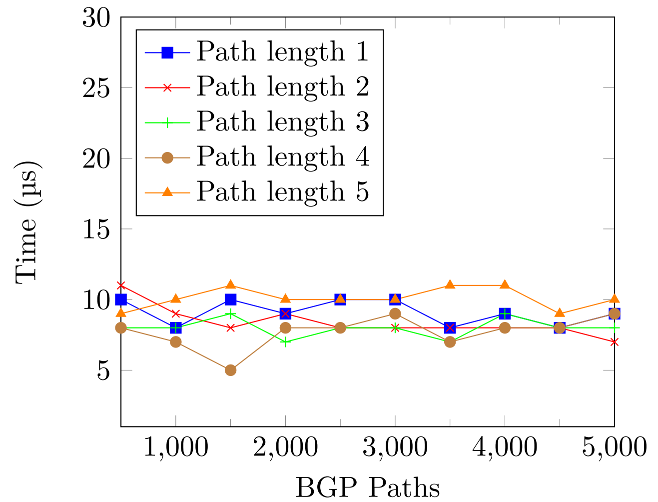
Figure 7.20: Measurement of BGPsec update generation. Processing time slightly increases with growing path length, as microbenchmarks have shown.

Although minor deviations are visible for increasing path length, both implementations show the expected linear behavior. FRR can not compete with BGP-SRx due to the repetitive key loading process each time a signature is generated. BGP-SRx generates BGPsec updates approximately 3x faster than FRR.

Next, the test is repeated with plain BGP updates. Generating updates for plain BGP does not underlie the restrictions of BGPsec. This means that a single BGP update can contain multiple prefixes. It is therefore not necessary to produce 5,000 individual announcements for 5,000 prefixes. They can instead be combined into a single BGP update. The test scenario is identical to the previous one, but instead of speaking BGPsec, routers are configured to only speak plain BGP. The following Figure 7.21 shows the test results.



(a) BGP update generation routine of FRR.



(b) BGP update generation routine of BGP-SRx.

Figure 7.21: Update generation of plain BGP updates. Only one update is generated per hop as BGP updates can contain multiple prefixes at once.

When looking at plain BGP update generation in Figure 7.21, it is noticeable that the times do only vary by a few microseconds. There is no visible growth with increasing path length nor amounts of paths. During the test it was indeed observed that only a single BGP update containing all prefixes was generated.

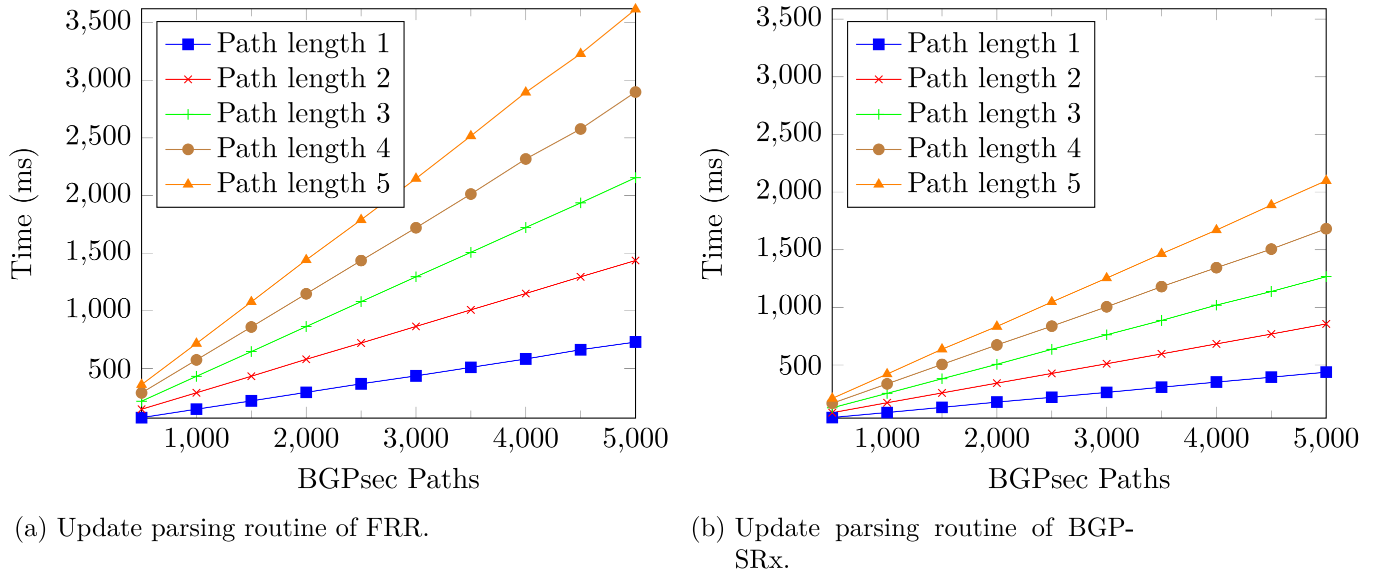
When comparing these results to BGPsec, the massive performance overhead becomes visible. Generating 5,000 updates of length 1 takes both implementations a mere 5 to 10 µs. The same amount of BGPsec updates take BGP-SRx up to 177 ms, FRR even 520 ms.

The following test compares processing time of incoming BGPsec updates. Further, incoming BGP updates are also measured and compared to BGPsec results. The setup is identical to the previous scenario.

7.6.2 Incoming Updates

For both implementations, the function `bgp_attr_parse` parses incoming BGP(sec) updates. Between 500 and 5,000 BGPsec updates are received and validated by FRR

and BGP-SRx instances. Each update is forwarded up to a path length of 5. The results are displayed in Figure 7.22.



(a) Update parsing routine of FRR.

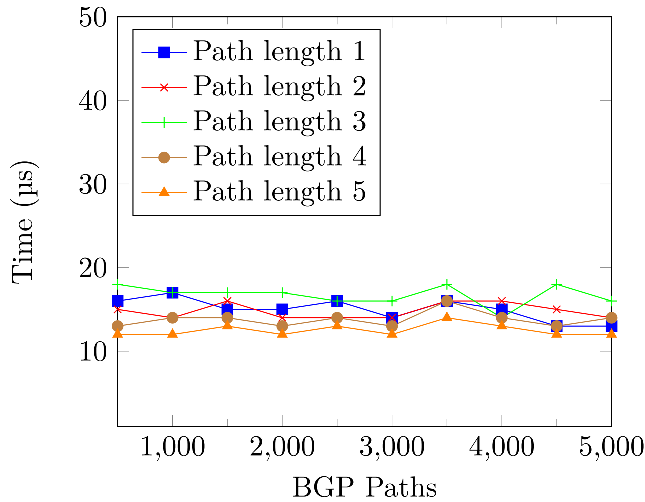
(b) Update parsing routine of BGP-SRx.

Figure 7.22: Parsing and validating up to 5,000 BGPsec updates.

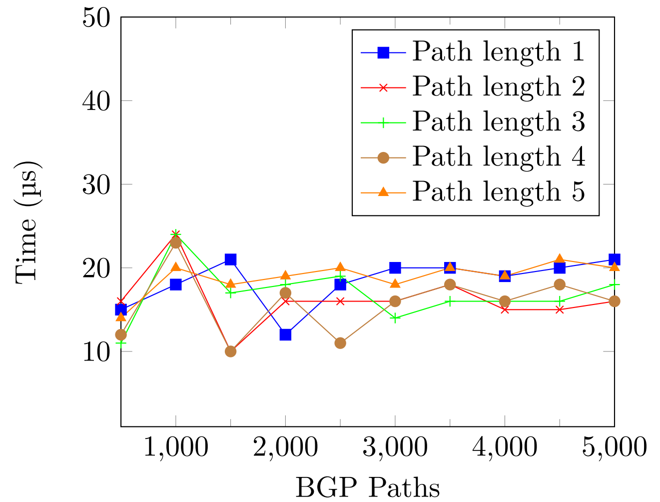
Update processing behaves linear for both implementations. FRR, however, takes much longer to process a single BGPsec update because of key loading. BGP-SRx does not suffer from this issue.

The duration for processing a single BGPsec paths grows with its path length. All graphs diverge by a constant value for each point on the x-axis. This is observed for both implementations. Next, the test is repeated with plain BGP.

Plain BGP updates can contain multiple prefixes. So just like with the test for update generation, only one BGP update is received and processed in the following scenario. Figure 7.23 shows the results.



(a) BGP update parsing routine of FRR.



(b) BGP update parsing routine of BGP-SRx.

Figure 7.23: Update parsing of plain BGP updates. Despite the growing path length and amount of prefixes, only a single update is generated. FRR and BGP-SRx show no observable difference in performance for varying path length.

Measuring the parsing routine for plain BGP updates yields a processing time with little variance. It takes between 10 and 20 µs to finish processing the update for FRR and between 10 and 25 µs for BGP-SRx. Since only one update is received and processed, path length and prefix count do not factor noticeably into the update parsing procedure.

In conclusion, BGP-SRx does outperform FRR in terms of BGPsec update generation as well as validation. In terms of plain BGP, both routing suites perform very similar to each other. Comparing BGPsec and BGP directly shows that AS path validation puts a tremendous overhead on the router. This is not only because of the cryptographic operations, but also because of the increasing amounts of individual updates that need to be generated and parsed. Throughout all tests, no exponential behavior was observed.

8 Conclusion and Outlook

This work presented a complete BGPsec implementation in the open-source library RTRlib and the open-source routing suite FRR. Alongside the implementation, its performance has been analyzed. The implementation was divided into two parts, the validation suite and the protocol handler. The former handles AS path signing and validation and was implemented with the RTRlib library. The latter manages protocol handling such as parsing the BGPsec PATH attribute. It was included into the FRR routing suite.

To ensure that the implementation is correct, BGPsec sessions were established between FRR and the reference implementation of BGPsec, BGP-SRx, in different constellations. FRR passed all tests, i.e., capability negotiation, correct update generation, update processing and update reconstruction while peering with BGP-SRx as well as other FRR instances. Error cases and malformed messages were properly handled according to the specification.

To measure the performance of the FRR BGPsec implementation, memory usage and run time was inspected and compared to BGP-SRx. The results showed that router keys in FRR occupy 8x less memory than keys in BGP-SRx. BGPsec PATH structures in FRR take up 3x less space. BGP-SRx performs 2x faster in terms of validating BGPsec updates and 3x faster when signing them. All expectations regarding the growth behavior of memory consumption and run time were met for both implementations. Memory usage scales linearly, just like run time for validation. Signing a single BGPsec update yielded a constant behavior independent of path length.

The large performance difference between FRR and BGP-SRx is explained with FRR repeatedly loading and freeing OpenSSL keys instead of keeping them in memory, like BGP-SRx does.

The run time of the cryptographic operations are not the main problem with BGPsec. A far greater issue is the way BGPsec distributes announcements. BGPsec needs to generate each update individually for each peer and each prefix. In contrast, plain BGP

can forward a single update containing multiple prefixes to multiple peers. The overhead that BGPsec puts on top of plain BGP is amplified by this circumstance.

Further work is required to improve performance and usability of FRR BGPsec. By keeping OpenSSL transformed router keys stored in memory, run times of the validation and signing routines can be lowered significantly. Right now, BGPsec in FRR is limited in its versatility. To address this, additional configuration options need to be implemented. What exactly operators want and need is still open for discussion.

Whether or not BGPsec will see deployment throughout the Internet is still uncertain. The fact that alternative AS path protection mechanisms [70, 71] are being standardized before BGPsec is even deployed on a small scale predicts a grim future for BGPsec. Without partial deployment, BGPsec would need to be accepted and used by multiple large clusters of ASes, ultimately merging together. Smaller clusters or individual ASes would then be able to join and benefit from AS path security [46]. Since this scenario is unlikely, many ASes will probably wait for alternatives to thrive.

In conclusion, although some work is still open in regards to the implementation, BGPsec receives another implementation for an up to date routing suite. This takes down another hurdle and pushes BGPsec towards deployment.

Bibliography

[1] R. Bush and R. Austein, "The Resource Public Key Infrastructure (RPKI) to Router Protocol, Version 1," IETF, RFC 8210, September 2017.

[2] M. Lepinski and K. Sriram, "BGPsec Protocol Specification," IETF, RFC 8205, September 2017.

[3] S. Turner and O. Borchert, "BGPsec Algorithms, Key Formats, and Signature Formats Appendix," IETF, RFC 8208, September 2017. [Online]. Available: https://tools.ietf.org/html/rfc8208#appendix-A (Accessed 25-02-2021).

[4] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4)," IETF, RFC 4271, January 2006.

[5] K. Butler, T. Farley, P. McDaniel, and J. Rexford, "A Survey of BGP Security Issues and Solutions," *Proc. of the IEEE*, vol. 98, no. 1, pp. 100–122, January 2010.

[6] M. Lepinski and S. Kent, "An Infrastructure to Support Secure Internet Routing," IETF, RFC 6480, February 2012.

[7] P. Mohapatra, J. Scudder, D. Ward, R. Bush, and R. Austein, "BGP Prefix Origin Validation," IETF, RFC 6811, January 2013.

[8] M. Wählisch, F. Holler, T. C. Schmidt, and J. H. Schiller, "RTRlib: An Open-Source Library in C for RPKI-based Prefix Origin Validation," in *Proc. of USENIX Security Workshop CSET'13*. Berkeley, CA, USA: USENIX Assoc., 2013. [Online]. Available: https://www.usenix.org/conference/cset13/rtrlib-open-source-library-c-rpki-based-prefix-origin-validation

[9] IANA, "Number Resources." [Online]. Available: https://www.iana.org/numbers (Accessed 02-01-2018).

[10] IANA, "Autonomous System (AS) Numbers." [Online]. Available: https://www.iana.org/assignments/as-numbers/as-numbers.xhtml (Accessed 20-02-2018).

[11] G. Huston, "Autonomous System (AS) Number Reservation for Documentation Use," IETF, RFC 5398, December 2008.

[12] A. Tanenbaum and D. Wetherall, *"Computernetzwerke"*. Pearson Deutschland GmbH, 2012.

[13] I. van Beijnum, *"BGP"*. O'Reilly Media, Inc., 2002.

[14] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4) Section 3.2 Routing Information Base," January 2006. [Online]. Available: https://tools.ietf.org/html/rfc4271#section-3.2 (Accessed 29-09-2020).

[15] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4) Section 9.1 Decision Process," January 2006. [Online]. Available: https://tools.ietf.org/html/rfc4271#section-9.1 (Accessed 27-08-2020).

[16] J. Karlin, S. Forrest, and J. Rexford, "Autonomous Security for Autonomous Systems," *Computer Networks*, vol. 52, no. 15, pp. 2908–2923, 2008.

[17] N. Spring, R. Mahajan, and T. Anderson, "The Causes of Path Inflation," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, 2003, pp. 113–124.

[18] G. Huston, M. Rossi, and G. Armitage, "Securing BGP - A literature survey," *IEEE Communications Surveys & Tutorials*, vol. 13, no. 2, pp. 199–222, 2011.

[19] M. Wählisch, O. Maennel, and T. C. Schmidt, "Towards Detecting BGP Route Hijacking using the RPKI," in *Proc. of ACM SIGCOMM, Poster Session*. New York: ACM, August 2012, pp. 103–104. [Online]. Available: http://conferences.sigcomm.org/sigcomm/2012/paper/sigcomm/p103.pdf

[20] C. Zheng, L. Ji, D. Pei, J. Wang, and P. Francis, "A Light-Weight Distributed Scheme for Detecting IP Prefix Hijacks in Real-Time," in *Proc. of SIGCOMM '07*. New York, NY, USA: ACM, 2007, pp. 277–288.

[21] M. Handley and E. Rescorla, "Internet Denial-of-Service Considerations," IETF, RFC 4732, December 2006.

[22] BGPmon, "Chinese ISP hijacks the Internet." [Online]. Available: https://bgpmon.net/chinese-isp-hijacked-10-of-the-internet/ (Accessed 12-03-2018).

[23] "YouTube Hijacking: A RIPE NCC RIS case study," http://www.ripe.net/internet-coordination/news/industry-developments/youtube-hijacking-a-ripe-ncc-ris-case-study, March 2008, retrieved 2013-08-16.

[24] A. Toonk, "BGPstream and The Curious Case of AS12388," 4 2017. [Online]. Available: https://bgpmon.net/bgpstream-and-the-curious-case-of-as12389 (Accessed 07-01-2018).

[25] A. Chadd, "The "AS7007 Incident"," 8 2006. [Online]. Available: http://lists.ucc.gu.uwa.edu.au/pipermail/lore/2006-August/000040.html (Accessed 06-01-2018).

[26] T. Wan and P. C. Van Oorschot, "Analysis of BGP prefix origins during Google's May 2005 outage," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International.* IEEE, 2006, pp. 8–pp.

[27] M. Lepinski, S. Kent, and D. Kong, "A Profile for Route Origin Authorizations (ROAs)," IETF, RFC 6482, February 2012.

[28] R. NCC, "BGP Origin Validation," 9 2016. [Online]. Available: http://www.ripe.net/manage-ips-and-asns/resource-management/certification/bgp-origin-validation (Accessed 12-01-2018).

[29] NLnetLabs, "RPKI Software Projects," Jun 2020. [Online]. Available: https://rpki.readthedocs.io/en/latest/tools.html (Accessed 12-08-2020).

[30] R. Bush and R. Austein, "The Resource Public Key Infrastructure (RPKI) to Router Protocol," IETF, RFC 6810, January 2013.

[31] M. Wählisch, R. Schmidt, T. C. Schmidt, O. Maennel, S. Uhlig, and G. Tyson, "RiPKI: The Tragic Story of RPKI Deployment in the Web Ecosystem," in *Proc. of 14th ACM Workshop on Hot Topics in Networks (HotNets).* New York: ACM, Nov. 2015, pp. 11:1–11:7. [Online]. Available: http://dx.doi.org/10.1145/2834050.2834102

[32] D. Iamartino, C. Pelsser, and R. Bush, "Measuring BGP route origin registration validation," in *Proc. of PAM*, ser. LNCS. Berlin: Springer, 2015, pp. 28–40.

[33] NIST, "Global Prefix/Origin Validation using RPKI." [Online]. Available: https://rpki-monitor.antd.nist.gov/ (Accessed 03-11-2020).

[34] R. NCC, "RIPE Certification Statistics." [Online]. Available: https://certification-stats.ripe.net/ (Accessed 03-11-2020).

[35] T. Chung, E. Aben, T. Bruijnzeels, B. Chandrasekaran, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, R. van Rijswijk-Deij, J. Rula, and N. Sullivan, "RPKI is Coming of Age: A Longitudinal Study of RPKI Deployment and Invalid Route Origins," in *Proc. of ACM IMC*. New York, NY, USA: ACM, 2019, pp. 406–419.

[36] S. Goldberg, "Why is It Taking So Long to Secure Internet Routing?" *Commun. ACM*, vol. 57, no. 10, pp. 56–63, September 2014.

[37] Y. Gilad, A. Cohen, A. Herzberg, M. Schapira, and H. Shulman, "Are We There Yet? On RPKI's Deployment and Security," in *Proc. of NDSS*. Reston, USA: ISOC, 2017.

[38] A. Reuter, R. Bush, I. Cunha, E. Katz-Bassett, T. C. Schmidt, and M. Wählisch, "Towards a Rigorous Methodology for Measuring Adoption of RPKI Route Validation and Filtering," *ACM Sigcomm Computer Communication Review*, vol. 48, no. 1, pp. 19–27, January 2018. [Online]. Available: https://doi.org/10.1145/3211852.3211856

[39] J. Kristoff, R. Bush, C. Kanich, G. Michaelson, A. Phokeer, T. C. Schmidt, and M. Wählisch, "On Measuring RPKI Relying Parties," in *Proc. of ACM Internet Measurement Conference (IMC)*. New York: ACM, 2020, pp. 484–491. [Online]. Available: https://doi.org/10.1145/3419394.3423622

[40] S. Kent, C. Lynn, and K. Seo, "Secure Border Gateway Protocol (S-BGP)," *Selected Areas in Communications, IEEE Journal on*, vol. 18, no. 4, pp. 582–592, 2000.

[41] IEEE, "IEEE Standard Specifications for Public-Key Cryptography," *IEEE Std 1363-2000*, pp. 1–228, Aug 2000.

[42] S. Turner and O. Borchert, "BGPsec Algorithms, Key Formats, and Signature Formats," IETF, RFC 8208, September 2017.

[43] M. Lepinski and K. Sriram, "BGPsec Protocol Specification Section 4.1 General Guidance," IETF, RFC 8205, September 2017. [Online]. Available: https://tools.ietf.org/html/rfc8205#section-4.1 (Accessed 15-10-2020).

[44] M. Lepinski and K. Sriram, "BGPsec Protocol Specification Section 5.1 Overview of BGPsec Validation," IETF, RFC 8205, September 2017. [Online]. Available: https://tools.ietf.org/html/rfc8205#section-5.1 (Accessed 15-10-2020).

[45] K. Sriram, "BGPsec Design Choices and Summary of Supporting Discussions Section 8.2 Signing and Forwarding Updates when Signatures Failed Validation," IETF, RFC 8374, April 2018. [Online]. Available: https://tools.ietf.org/html/rfc8374#section-8.2 (Accessed 15-10-2020).

[46] M. Lepinski and K. Sriram, "BGPsec Protocol Specification Section 9.7 Incremental/Partial Deployment Considerations," IETF, RFC 8205, September 2017. [Online]. Available: https://tools.ietf.org/html/rfc8205#section-9.7 (Accessed 21-10-2020).

[47] J. Ng, "Extensions to BGP to Support Secure Origin BGP (soBGP)," IETF, Internet-Draft – work in progress 02, April 2004.

[48] R. Lychev, S. Goldberg, and M. Schapira, "BGP Security in Partial Deployment: Is the Juice Worth the Squeeze?" in *Proc. of ACM SIGCOMM*. New York, NY, USA: ACM, 2013, pp. 171–182.

[49] P. Gill, M. Schapira, and S. Goldberg, "Let the Market Drive Deployment: A Strategy for Transitioning to BGP Security," in *Proc. of ACM SIGCOMM*. New York, NY, USA: ACM, 2011, pp. 14–25.

[50] A. Cohen, Y. Gilad, A. Herzberg, and M. Schapira, "One Hop for RPKI, One Giant Leap for BGP Security," in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XIV. New York, NY, USA: Association for Computing Machinery, 2015. doi: 10.1145/2834050.2834078. ISBN 9781450340472. [Online]. Available: https://doi.org/10.1145/2834050.2834078

[51] S. T. Kent, C. Lynn, J. Mikkelson, and K. Seo, "Secure Border Gateway Protocol (S-BGP)-Real World Performance and Deployment Issues." in *NDSS*, 2000.

[52] S. Goldberg, M. Schapira, P. Hummon, and J. Rexford, "How Secure are Secure Interdomain Routing Protocols," in *Proc. of ACM SIGCOMM*. New York, NY, USA: ACM, 2010, pp. 87–98.

[53] NIST, "BGP Secure Routing Extension (BGP-SRx) Prototype," 9 2017. [Online]. Available: https://www.nist.gov/services-resources/software/bgp-secure-routing-extension-bgp-srx-prototype (Accessed 16-05-2018).

[54] D. Conry-Murray, "Free Range Routing Project Forks Quagga," Apr 2017. [Online]. Available: https://packetpushers.net/free-range-routing-project-forks-quagga (Accessed 12-08-2020).

[55] Opensource.org, "The MIT License." [Online]. Available: https://opensource.org/licenses/MIT (Accessed 24-09-2020).

[56] NIST, "Digital Signature Standard," National Institute of Standards & Technology, Gaithersburg, MD, US, Federal Information Processing Standard 186–4, July 2013.

[57] NIST, "Secure Hash Standard," National Institute of Standards & Technology, Gaithersburg, MD, US, Federal Information Processing Standards 180–3, October 2008.

[58] OpenSSL, "OpenSSL License." [Online]. Available: https://www.openssl.org/source/license.html (Accessed 20-08-2020).

[59] OpenSSL, "OpenSSL 1.0 Documentation." [Online]. Available: https://www.openssl.org/docs/man1.0.2/apps/openssl.html (Accessed 20-08-2020).

[60] OpenSSL, "OpenSSL 1.1 Documentation." [Online]. Available: https://www.openssl.org/docs/man1.1.1/man1/openssl.html (Accessed 20-08-2020).

[61] OpenSSL, "OpenSSL Versions." [Online]. Available: https://www.openssl.org/news/vulnerabilities-1.0.2.html (Accessed 20-08-2020).

[62] OpenSSL, "OpenSSL GitHub Install." [Online]. Available: https://github.com/openssl/openssl#build-and-install (Accessed 20-08-2020).

[63] OpenSSL, "OpenSSL Vulnerabilities." [Online]. Available: https://www.openssl.org/news/vulnerabilities-1.1.1.html (Accessed 20-08-2020).

[64] OpenSSL, "OpenSSL Standards." [Online]. Available: https://beta.openssl.org/docs/standards.html (Accessed 25-11-2020).

[65] M. Lepinski and K. Sriram, "BGPsec Protocol Specification Section 5.2 Validation Algorithm," IETF, RFC 8205, September 2017. [Online]. Available: https://tools.ietf.org/html/rfc8205#section-5.2 (Accessed 29-09-2020).

[66] Q. Vohra and E. Chen, "BGP Support for Four-Octet Autonomous System (AS) Number Space," IETF, RFC 6793, December 2012.

[67] T. Bates, R. Chandra, D. Katz, and Y. Rekhter, "Multiprotocol Extensions for BGP-4," IETF, RFC 4760, January 2007.

[68] E. Chen, J. Scudder, P. Mohapatra, and K. Patel, "Revised Error Handling for BGP UPDATE Messages," IETF, RFC 7606, August 2015.

[69] S. Turner and O. Borchert, "BGPsec Algorithms, Key Formats, and Signature Formats," IETF, RFC 8608, June 2019.

[70] A. Azimov, E. Uskov, R. Bush, K. Patel, J. Snijders, and R. Housley, "A Profile for Autonomous System Provider Authorization," IETF, Internet-Draft – work in progress 04, November 2020.

[71] A. Azimov, E. Bogomazov, R. Bush, K. Patel, and J. Snijders, "Verification of AS_-PATH Using the Resource Certificate Public Key Infrastructure and Autonomous System Provider Authorization," IETF, Internet-Draft – work in progress 06, November 2020.

A Appendix

```yaml
version: "2.3"
services:
    spki-cache:
        image: spki-cache-server:latest
        volumes:
            - ./keys:/keys
        privileged: true
        networks:
            netz_a:
                ipv4_address: 172.18.0.100
                ipv6_address: 2001:3200:3200::100
        stdin_open: true
        tty: true
    frr1:
        image: frr-bgpsec
        volumes:
            - ./conf/bgpd/bgpd1.conf:/etc/frr/bgpd.conf
            - ./conf/zebra/zebra1.conf:/etc/frr/zebra.conf
            - ./frr:/frr
            - ./conf/setup.sh:/setup.sh
            - ./privkeys/privkey1.der:/frr/privkey.der
        privileged: true
        networks:
            netz_a:
                ipv4_address: 172.18.0.2
                ipv6_address: 2001:3200:3200::2
        stdin_open: true
        tty: true
```

```yaml
    frr2:
        image: frr-bgpsec
        volumes:
            - ./conf/bgpd/bgpd2.conf:/etc/frr/bgpd.conf
            - ./conf/zebra/zebra2.conf:/etc/frr/zebra.conf
            - ./frr:/frr
            - ./conf/setup.sh:/setup.sh
            - ./privkeys/privkey2.der:/frr/privkey.der
        privileged: true
        networks:
            netz_a:
                ipv4_address: 172.18.0.3
                ipv6_address: 2001:3200:3200::3
        stdin_open: true
        tty: true
networks:
    netz_a:
        enable_ipv6: true
        driver: bridge
        ipam:
            driver: default
            config:
                - subnet: 172.18.0.0/24
                - subnet: 2001:3200:3200::/64
```

Listing A.1: Truncated Docker-Compose configuration to set up a test scenario. The full configuration includes multiple FRR and/or BGP-SRx containers that are connected in a chain-like constellation.

```
#include <openssl/x509.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <sys/resource.h>

// HASH
static uint8_t hash[] = {
        0x01, 0x4F, 0x24, 0xDA, 0xE2, 0xA5, 0x21, 0x90, 0xB0, 0x80,
        0x5C, 0x60, 0x5D, 0xB0, 0x63, 0x54, 0x22, 0x3E, 0x93, 0xBA,
        0x41, 0x1D, 0x3D, 0x82, 0xA3, 0xEC, 0x26, 0x36, 0x52, 0x0C,
        0x5F, 0x84};

// SIG
static uint8_t sig[] = {
        0x30, 0x46, 0x02, 0x21, 0x00, 0xEF, 0xD4, 0x8B, 0x2A, 0xAC,
        0xB6, 0xA8, 0xFD, 0x11, 0x40, 0xDD, 0x9C, 0xD4, 0x5E, 0x81,
        0xD6, 0x9D, 0x2C, 0x87, 0x7B, 0x56, 0xAA, 0xF9, 0x91, 0xC3,
        0x4D, 0x0E, 0xA8, 0x4E, 0xAF, 0x37, 0x16, 0x02, 0x21, 0x00,
        0x90, 0xF2, 0xC1, 0x29, 0xAB, 0xB2, 0xF3, 0x9B, 0x6A, 0x07,
        0x96, 0x3B, 0xD5, 0x55, 0xA8, 0x7A, 0xB2, 0xB7, 0x33, 0x3B,
        0x7B, 0x91, 0xF1, 0x66, 0x8F, 0xD8, 0x61, 0x8C, 0x83, 0xFA,
        0xC3, 0xF1};

// KEY
static uint8_t spki[] = {
        0x30, 0x59, 0x30, 0x13, 0x06, 0x07, 0x2A, 0x86, 0x48, 0xCE,
        0x3D, 0x02, 0x01, 0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D,
        0x03, 0x01, 0x07, 0x03, 0x42, 0x00, 0x04, 0x28, 0xFC, 0x5F,
        0xE9, 0xAF, 0xCF, 0x5F, 0x4C, 0xAB, 0x3F, 0x5F, 0x85, 0xCB,
        0x21, 0x2F, 0xC1, 0xE9, 0xD0, 0xE0, 0xDB, 0xEA, 0xEE, 0x42,
        0x5B, 0xD2, 0xF0, 0xD3, 0x17, 0x5A, 0xA0, 0xE9, 0x89, 0xEA,
        0x9B, 0x60, 0x3E, 0x38, 0xF3, 0x5F, 0xB3, 0x29, 0xDF, 0x49,
        0x56, 0x41, 0xF2, 0xBA, 0x04, 0x0F, 0x1C, 0x3A, 0xC6, 0x13,
        0x83, 0x07, 0xF2, 0x57, 0xCB, 0xA6, 0xB8, 0xB5, 0x88, 0xF4,
        0x1F};

int main() {
    EC_KEY *pub_key = NULL;
    char *p = NULL;
    struct rusage before, after;
    unsigned long cpu_time_used = 0;
    unsigned long total = 0;

    p = (char *)spki;
    pub_key = d2i_EC_PUBKEY(NULL, (const unsigned char **)&p,
                (long)91);

    for (int i = 0; i < 1000; i++) {
```

```
49          getrusage(RUSAGE_SELF, &before);
50          ECDSA_verify(
51                  0,
52                  hash,
53                  SHA256_DIGEST_LENGTH,
54                  sig,
55                  72,
56                  pub_key);
57          getrusage(RUSAGE_SELF, &after);
58
59          cpu_time_used =
60              (after.ru_utime.tv_sec - before.ru_utime.tv_sec) * \
61              CLOCKS_PER_SEC + \
62              (after.ru_utime.tv_usec - before.ru_utime.tv_usec);
63          cpu_time_used +=
64              (after.ru_stime.tv_sec - before.ru_stime.tv_sec) * \
65              CLOCKS_PER_SEC + \
66              (after.ru_stime.tv_usec - before.ru_stime.tv_usec);
67          total += cpu_time_used;
68
69          printf("Result: %luus\n", cpu_time_used);
70      }
71      printf("Result average: %luus\n", total/1000);
72
73      EC_KEY_free(pub_key);
74
75      return 0;
76  }
```

Listing A.2: Program for benchmarking the OpenSSL validation function `ECDSA_verify`. With a few adjustments, the same program can be used to benchmark the signing function `ECDSA_sign`.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „- bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] - ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen.“

Quelle: §16 Abs. 5 APSO-TI-BM bzw. §15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: ______________________________

Vorname: ______________________________

dass ich die vorliegende Masterarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Implementation and Evaluation of BGPsec for the FRRouting Suite

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

____________	____________	______________________
Ort	Datum	Unterschrift im Original