Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Implementing and Integrating BGPsec AS Path Validation into RTRlib

Colin Sames

Hauptprojekt

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Colin Sames

Implementing and Integrating BGPsec AS Path Validation into RTRlib

Implementing and Integrating BGPsec AS Path Validation into RTRlib eingereicht im Rahmen des Hauptprojekts

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

Eingereicht am: March 14, 2019

Contents

1 Introduction

In the early days of the Internet, many exploits and attacks that are targeted at the Internet traffic did not yet exist. Because of that, protocols back then were not designed with appropriate protection mechanisms. One of these protocols is the Border Gateway Protocol (BGP) [1], a protocol Autonomous Systems (ASes) use to exchange routing information. The contents of BGP messages can be manipulated to cause wrong routing behavior between ASes. To prevent such incidents, a protection mechanism has been developed that enables ASes to validate crucial information of BGP messages. These mechanisms are called *route origin validation (ROV)* and *AS path validation*. They are both utilized via the Resource Public Key Infrastructure (RPKI) [2].

The RPKI protects two sensible information within an BGP message: the *origin AS* of the message and the *AS path* that the message took. The first information is protected by ROV. Each announced IP address range (prefix) is linked to a certain AS. This linkage can be checked by ASes that receive a BGP message. If a message announces an AS-prefix-tuple for which no such linkage exists, the message may be rejected.

The second information, the AS path, can be protected by the BGP protocol extension BGPsec [3]. It adds cryptographic validation to the contents of a BGP message that can be validated by receiving ASes. If the contents of such a message are changed along the AS path, a validation check will no longer yield a valid result.

The information that are required to perform both ROV as well as AS path validation is gathered from the RPKI using the RPKI To Router (RTR) [4] protocol. Implementations for RTR and ROV exist while BGPsec implementations are sparse, so it is important to develop software that enables AS path validation for BGP router operators. Without this software, the deployment of BGPsec is constrained.

In this work it is shown, how the library RTRlib [5][1], which already offers ROV, will be extended with BGPsec AS path validation. A prior prepared implementation concept will serve as the foundation. The new feature will allow a user to *validate* and *sign* the AS path of BGPsec updates using the library.

[1] http://rtrlib.realmv6.org/

Currently, only few BGPsec implementations exist, like the BGP-SRx [6], developed by the National Institute of Standards and Technology (NIST), or the BIRD routing suite[2]. The goal of integrating BGPsec AS path validation into RTRlib is its independence from any software routers, so that any application may integrate it and make use of its RPKI features.

The remainder of this report is structured as follows. At first, in Section 2, a brief overlook over the implementation concept is given and requirements to the implementation are made. Next, Section 3 will present the crypto library that is going to be used. The implementation part is described in Section 4. To proof the functionality, Section 5 compares RTRlib AS path validation to BGP-SRx validation. Basic performance measurements are presented in Section 6. Section 7 concludes this work.

[2] http://bird.network.cz

2 Concept and Requirements

The RFC 8205 [3] specifies all of the BGPsec functionalities, but only validating and signing AS paths is relevant to RTRlib. The rest of the mainly operational BGPsec features are left to the software that includes RTRlib. The underlying concept for the implementation of these features was developed in a separate work. To give an overview of what this concept looks like, its central points are presented here:

- *The implementation only provides necessary features to provide AS path validation.* RTRlib aims to remain lightweight.

- *BGPsec AS path validation merely extends current RPKI features of RTRlib.* Current functionalities of the library are not altered in any way.

- *The implementation is similar to the current API of ROV.* Since it is the same library both functionalities are embedded in, their usage should be similar to each other.

- *The library remains independent.* It should be usable by any application and not be specifically designed for one piece of software only.

- *The implementation offers an API to make use of AS path validation.* There are several functionalities that must be implemented:
  - Get the supported BGPsec version
  - Check support for the propagated algorithm suite
  - Validate a BGPsec AS path
  - Sign a BGPsec AS path

- *The usage of the new features are documented.* The current documentation for the RTRlib will be extended to describe validation and signing routines. There should also be code examples that show a minimal working example.

The goal is to satisfy all of these requirements equally, yet trade-offs are made if necessary. If, e.g., the new API design needs to differ from the current one to make the usage of AS path validation more convenient, this trade-off is made.

The reasoning for only providing AS path validation is that RTRlib tries to stay as basic as possible, providing only RTR/RPKI specific features. For instance, adding a features for assembling the BGPsec_PATH attribute would meddle with data structures that are specific to routing suites or other software. BIRD and FRRouting (FRR)[1] both have their own way to represent internal structures. Making use of these structures would go against the goal of keeping RTRlib independent. Therefore, any data that is passed to RTRlib must first be stored in a format or structure that RTRlib itself provides.

The next section is a quick look at the crypto library that RTRlib is going to depend on.

[1] https://frrouting.org

3 Crypto Library Considerations

To validate and sign a BGPsec path, cryptographic operations are required. These operations are *validating* and *generating* signatures with the *Elliptic Curve Digital Signature Algorithm (ECDSA)* [7]. The input for this algorithm is a byte sequence which is digested by the *SHA-256* algorithm [8]. These two algorithms form the *algorithm suite 1* [9]. An algorithm suite defines a set of cryptographic algorithms. In the context of BGPsec, the protocol must support all cryptographic algorithms specified by the algorithm suite. In case an algorithm suite 2 is added to the protocol specification, the new set of algorithms must also be supported.

RTRlib will depend on an existing, throughout tested library to provide cryptographic functionalities. There are a few requirements to these libraries.

- Provide a C compatible API
- Provide all necessary and potential future algorithms
- Long term support
- Open source and free to use
- Well-known and widely supported
- Secure and no known major bugs

Since RTRlib is implemented in C, the crypto-library that is included must be compatible with C. The library must also support all algorithms specified by algorithm suite 1. Additionally, the library should support a broad variety of other modern algorithms so RTRlib does not need to switch libraries every time a new algorithm suite is released.

Long term support is just as important. A discontinued library is dangerous because undiscovered bugs are unlikely to be fixed short-term, if they are to be fixed at all. A continuously developed crypto-library is hence indispensable.

RTRlib is open source and free, so all components of the software should be as well, including the crypto-library. In addition, using a piece of software that is well-known to the Internet is more convenient than relying on lesser known software. Potential users of the RTRlib might trust the crypto-library more, if it is known to them. Including crypto software into RTRlib that is unheard of to most of the users could potentially raise skepticism about the reliability

of RTRlib. Another reason is that wildly known software is more likely to be supported by the underlaying operating system. Last, the crypto-library of choice should be secure and should not contain security critical bugs that are known of.

One library that meets all these criteria is *OpenSSL*[1], an open-source and free to use C library. OpenSSL splits into two libraries, *libssl* and *libcrypto*, yet only the latter is required for our use case. Since libcrypto is part of OpenSSL, we will refer to OpenSSL when speaking about its crypto functionalities.

OpenSSL provides all necessary algorithms, namely elliptic curve cryptography and SHA-256 hashing [10] and also offers a range of other algorithms that are potentially required in the future. The library offers support for version 1.0 until end of 2019 and long-term support for version 1.1. OpenSSL is often considered the standard library when it comes to crypto operations, since it is well-established and well-known. Even the reference implementation of BGPsec, BGP-SRx, uses OpenSSL (version 1.0). RTRlib aims to provide support for both versions, 1.0 and 1.1, in case the system where RTRlib is installed prefers one version over the other.

In conclusion in can be said that using OpenSSL seems to be the most convenient way of handling the crypto operations that are required for BGPsec AS path validation.

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

4 Implementation

This section will be a view on the C implementation of BGPsec AS path validation for RTRlib. Not every line of code will be discussed, though. Instead, the most important and most interesting parts will be highlighted. The code is mostly reduced to pseudo code and slightly adjusted to make it comprehensible, even without much context.

The implementation consists of multiple additions to the current code base of RTRlib. The first addition are two new headers, *bgpsec.h* and *bgpsec_private.h*. The former contains public structs and enums, while the latter contains function declarations.

The second file is the source file, *bgpsec.c*, which implements the API functions. Last, tests which are defined in *test_bgpsec.c* cover the new functionalities offered by the AS path validation to validate the implementation.

The whole implementation is hosted on GitHub as a fork of RTRlib[1]. The header and source files can be found in `/rtrlib/bgpsec`, the test file is located in the `/tests` directory.

4.1 The Header Files

The header files contain all data structures and function definitions that are necessary for the public API. In the following, both the new structs as well as the new functions and their tasks are introduced.

As the names suggest, the private header file does not expose its functions to the user. To make them available, RTRlib exposes these functions via the *rtr_mgr*. In there, wrapper functions are defined that invoke the appropriate BGPsec function. This way, the BGPsec API is centralized with the rest of the RTRlib API.

The validation and signing functions require the input data to be stored in certain structures, so that it can be processed properly by the functions. So prior to invoking the functions, the user needs to fill these data structures with all the necessary information. Below in Listing 4.1, all three required RTRlib C structs are presented.

[1] https://github.com/colinbs/rtrlib/tree/bgpsec

Listing 4.1: The structs with their members roughly represent the BGPsec_PATH elements.

```
1  struct rtr_secure_path_seg {
2    uint8_t pcount;
3    uint8_t conf_seg;
4    uint32_t asn;
5  };
6
7  struct rtr_signature_seg {
8    uint8_t *ski;
9    uint16_t sig_len;
10   uint8_t *signature;
11 };
12
13 struct rtr_bgpsec_data {
14   uint8_t alg_suite;
15   uint8_t safi;
16   uint16_t afi;
17   uint32_t asn;
18   uint8_t *nlri;
19   uint8_t nlri_len;
20 };
```

The `rtr_secure_path_seg` with its three members represents a *Secure Path Segment* as specified by the BGPsec RFC.

The second struct is the representation of a *Signature Segment*. It also has three members, two of which are pointers. The `ski` is at a constant length of 20 bytes, while the length of the `signature` is variable. This is what the `sig_len` field is for.

The third and last struct is the `rtr_bgpsec_data` struct. It holds the rest of the data that is relevant for AS path validation. The `asn` field in line 17 holds the ASN of the AS that currently processes the BGPsec update and performs AS path validation. The `nlri` field holds the IP address in binary form. The *nlri_len* field holds the length of the IP address in bits. As specified by the RFC, only one IP prefix per BGPsec update is allowed.

Aside from the structures, there are a total of five new public functions available with the RTRlib API. The following functions that are presented are the wrapper functions which are defined in the *rtr_mgr*. With the exception of the validation function, which dereferences one of its input parameters before passing it down, all functions simply invoke the private BGPsec function with the received parameters.

rtr_mgr_bgpsec_validate_as_path validates a given BGPsec path. It takes the above mentioned structures as input parameters.

rtr_mgr_bgpsec_create_signature generates a signature for a given BGPsec path. This function additionally requires the private key of the signing application or device, such as a router.

rtr_mgr_bgpsec_get_version returns the latest supported BGPsec version of RTRlib. This function is required to make sure that the application only passes compatible BGPsec paths to RTRlib.

rtr_mgr_bgpsec_check_algorithm_suite checks, if a given algorithm suite is supported by RTRlib. Another useful function that lets the application check, if the propagated algorithm suite within a BGPsec update is compatible with RTRlib. This check is optional though, as both the validation and signing functions perform this check as a safety measure.

rtr_mgr_bgpsec_get_algorithm_suites_arr returns a pointer to a static list containing all algorithm suites supported by RTRlib. In case the user prefers to check against a static list instead of invoking `rtr_mgr_bgpsec_check_algorithm_suite` each time a BGPsec update arrives.

The return values of these functions depend on the outcome. Usually, these values are RTR_BGPSEC_SUCCESS or RTR_BGPSEC_ERROR. The latter return value may be more specific to let the user know, why an error occurred:

- RTR_BGPSEC_LOAD_PUB_KEY_ERROR if the public key could not be loaded
- RTR_BGPSEC_LOAD_PRIV_KEY_ERROR if the private key could not be loaded
- RTR_BGPSEC_ROUTER_KEY_NOT_FOUND if a router key was not found within the SPKI table
- RTR_BGPSEC_SIGNING_ERROR if an error occurred during the signing process
- RTR_BGPSEC_UNSUPPORTED_ALGORITHM_SUITE if the propagated algorithm suite is not supported

If a user is interested in why an action did not terminate successfully, it is up to them to check for these error codes. Regardless of why an error occurred, the action was not successful and should be repeated.

There are two exceptions in regards to successful return values. The first exception is the validation function, as it has its own two values on a successful return.

- RTR_BGPSEC_VALID if the BGPsec update is valid or
- RTR_BGPSEC_NOT_VALID if the BGPsec update is not valid

Note that RTR_BGPSEC_NOT_VALID is not the same as RTR_BGPSEC_ERROR. The validation function may return with RTR_BGPSEC_ERROR, yet this does not determine the status of the validation. This only indicates that the path validation could not complete successfully, thus the validity of the AS path remains unknown.

The other exception is the signing function which will, instead of RTR_BGPSEC_SUCCESS, return the length of the created signature.

4.2 The Source File

The implementation of the API functions will be explained in the following. There are also a few helper functions that are worth mentioning.

These helper functions are private and handle different parts of the validation/signing process. Some of them are only required for debugging purposes such as pretty-printing. The underscore prefix (_) signals that these functions are private and should be used carefully.

_align_val_byte_sequence aligns the sequence of bytes used for digestion from the BGPsec path segments, as the RFC specifies. A more detailed explanation on why the bytes need to be aligned like this can be found here [11].

_align_gen_byte_sequence does the same as `align_val_byte_sequence`, but for signing. Both procedures slightly differ from each other, which is why there need to be two functions.

_hash_byte_sequence takes a byte sequence and hashes it.

_validate_signature takes all necessary and prepared parameters and performs the cryptographic validation of a signature.

_get_sig_segs_size calculates the size in bytes of all signature segments. Because signature segments are of dynamic size, this function is necessary in order to allocate memory.

_load_public_key loads a public key from the SPKI table into an OpenSSL struct.

_load_private_key loads a private key into an OpenSSL struct.

_byte_sequence_to_str returns a human-readable string of a given byte sequence.

_bgpsec_segment_to_str returns a human-readable string of the contents of a given BGPsec path segment.

_ski_to_char converts the byte representation of a given SKI into human readable form.

None of these helper functions are publicly exposed to the RTRlib API. They alone provide little benefits for the user and can cause unexpected behavior and errors when used out of context.

These helper functions are required during the validation and signing processes. The following listings show how the validation process is implemented. The code is reduced to pseudo code and some portions have been removed to ease readability (e.g., error handling, memory allocation, most comments).

At the beginning of the validation function in line 1 of Listing 4.2, a check is made whether or not RTRlib supports the propagated algorithm suite. If not, the function returns an appropriate error code. This needs to be done in case the user did not check the supported algorithm suites beforehand.

Listing 4.2: Check, if the algorithm suite is supported.

```
1 if _bgpsec_check_algorithm_suite(alg_suite) == RTR_BGPSEC_ERROR:
2     return RTR_BGPSEC_UNSUPPORTED_ALGORITHM_SUITE
```

If the function supports the algorithm suite, RTRlib makes sure that all required router keys are present in the SPKI table, as seen in Listing 4.3. Doing this before proceeding any further potentially saves time because it prevents abortion of the process due to a missing router key halfway through the validation. If the router key is missing during this check, the validation function returns with an error code. The function in line 2 that searches for a specific SKI is part of the SPKI API of RTRlib.

Listing 4.3: To validate the AS path, all router keys must be present.

```
3 for i = 0, i < as_hops, i++:
4     router_key = spki_table_search_by_ski(skis[i])
5     if router_key == NULL:
6         return RTR_BGPSEC_ROUTER_KEY_NOT_FOUND
```

```
7  end for
```

Now that everything is set up and present, the data needs to be aligned as a specific byte sequence [11]. This is done via the `_align_val_byte_sequence` function. The `bytes` pointer is passed to the function, as seen in Listing 4.4. On a successful return, the pointer points to the whole byte sequence.

Listing 4.4: Assemble the data and store the byte sequence in `bytes`.

```
8  bytes = _align_val_byte_sequence(data,
9                                   sig_segs,
10                                  sec_paths)
```

With the byte sequence assembled, the hashing and validation itself can begin. Listing 4.5 shows, how a byte sequence and how its digest could look like.

Listing 4.5: At the top is the assembled byte sequence from the BGPsec_PATH data. Below is the resulting hash.

```
Byte Sequence:
00 01 00 01 AB 4D 91 0F 55 CA E7 1A 21 5E F3 CA
FE 3A CC 45 B5 EE C1 54 00 48 30 46 02 21 00 EF
D4 8B 2A AC B6 A8 FD 11 40 DD 9C D4 5E 81 D6 9D
2C 87 7B 56 AA F9 91 C3 4D 0E A8 4E AF 37 16 02
21 00 8E 21 F6 0E 44 C6 06 6C 8B 8A 95 A3 C0 9D
3A D4 37 95 85 A2 D7 28 EE AD 07 A1 7E D7 AA 05
5E CA 01 00 00 01 00 00 01 00 00 00 FB F0 01 00
01 01 18 C0 00 02

Digest:
01 4F 24 DA E2 A5 21 90 B0 80 5C 60 5D B0 63 54
22 3E 93 BA 41 1D 3D 82 A3 EC 26 36 52 0C 5F 84
```

Hashing is done inside a for-loop since there can be multiple AS path elements which need to be processed one by one, each element being hashed individually. The for-loop over all segments runs as long as the validation result of each iteration is valid. If the result is not valid, the whole BGPsec path is considered not valid and the validation process stops. Continuing would be an unnecessary waste of processing power. Line 13 of Listing 4.6 shows the for-loop.

Listing 4.6: Main loop for validation.

```
12 offset = 0
13 val_result
14 for i = 0, i < as_hops, i++:
15     hash_result = _hash_byte_sequence(bytes, offset)
16     router_keys = spki_table_search_by_ski(skis[i])
17
18     for j = 0, j < router_keys_length, j++:
19         val_result = _validate_signature(hash_result,
20                                          signature[i],
21                                          router_keys[j])
22         if val_result == BGPSEC_VALID:
23             break
24     end for
25
26     if val_result == BGPSEC_NOT_VALID:
27         break
28
29     offset = sig_length + ski_length + sec_path_size
30 end for
31
32 return val_result
```

The `offset` variable is used to determine, from which position on the byte sequence needs to be hashed. Figure 4.1 illustrates how the offset is shifted throughout the iterations. At the beginning there is no need for an offset on the byte sequence, thus `offset` is set to 0 in line 12. The variable `val_result` in line 13 holds the current status of the validation. It does not need to be initialized. It is continuously updated by the outer for-loop that starts in line 14. Next, in line 15, the byte sequence is hashed from the starting position on via SHA-256. The result is stored in `hash_result`. Then, all router keys for the current SKI are collected in line 16. In case there is more than one key bound to a SKI, another for-loop in line 18 iterates through all keys and tries to validate the current signature with each of them (line 19). Only one of the keys needs to be able to produce a valid result when validating the signature. If this is the case, the inner loop breaks in line 23.

Before continuing with the next iteration of the outer for-loop, the byte sequence offset is set to the point where the next BGPsec path segment starts and the process repeats (line 29).

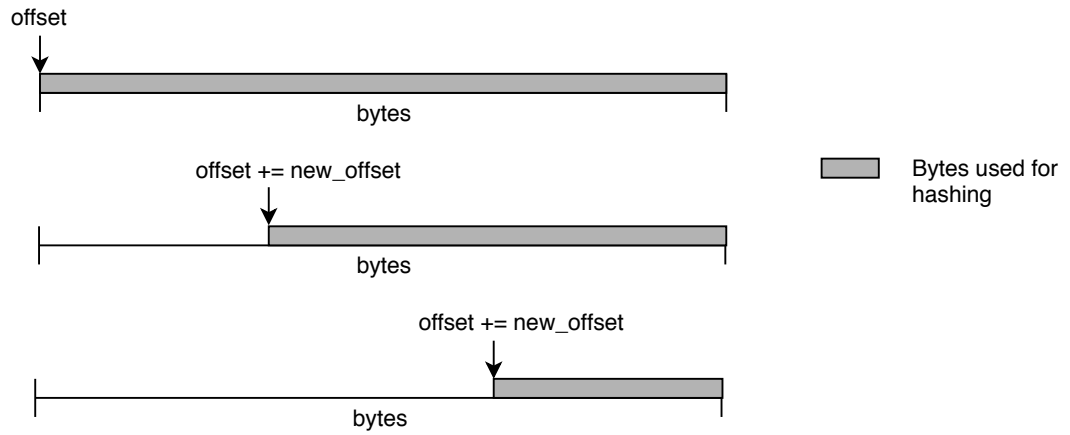

Figure 4.1: Consecutively processing of bytes. At the beginning the offset is 0, so all bytes are hashed. In the next iteration the offset moves forward by new_offset bytes. This continues until all bytes are processed.

If all signatures within the BGPsec path were successfully validated, the for-loop finishes and returns the result to the application that called the function. If one of the signatures was determined not valid, the process stops prematurely and the BGPsec path and eventually the whole BGPsec update is considered *not valid*.

When a user executes the validation function and is returned a positive result, they continue with appending the own AS information to the BGPsec path of the update. Prior to that, these information must be signed. To do so, the user calls the `rtr_mgr_bgpsec_generate_signature` function of RTRlib. The functions internals work very similar to those of the validation function, so the signing procedure is only discussed briefly. Notable differences to validation are pointed out.

Since during signing there is no need for public router keys, all SPKI checks are not necessary. This significantly shortens the function. Checks for algorithm suites are still required, though. The whole simplified singing procedure can be viewed in Listing 4.7.

Listing 4.7: Main loop for validation.

```
1 priv_key = _load_private_key(raw_key)
2
3 byte_sequence = _align_gen_byte_sequence(data,
4                                          sig_segs,
5                                          sec_paths,
```

```
6                                                   own_sec_path,
7                                                   target_as)
8
9  hash_result = _hash_byte_sequence(bytes, offset)
10
11 sig_len = ECDSA_sign(signature, hash_result, priv_key);
12
13 if sig_len < 1:
14     return RTR_BGPSEC_SIGNING_ERROR
15
16 return sig_len
```

At first, the private key, which is passed as raw bytes, must be loaded into an OpenSSL key structure in line 1. If successful, the byte sequence over all information is then constructed in line 3 and afterwards hashed in line 9. Then in line 11 The resulting hash is, together with the private key, used to create the signature. The variable `signature` in line 11 is a pointer passed to the function by the user. If `sig_len` is less than 1, signing was not successful and an error code is returned (line 13-14). Otherwise, signing was successful. The signature is stored in `signature` and its length is returned to the user.

4.3 The Test File

To provide a small set of test coverage of the new API functions, tests exist that execute these functions. These tests cover the following functionalities:

- Check for version and algorithm suites
  - Check return values on (not) supported algorithm suites
  - Check the length of the algorithm suites array
- Validate a BGPsec path with...
  - ...everything set up correctly
  - ...a wrong signature
  - ...a wrong public key within the SPKI table
  - ...a public key not present within the SPKI table
  - ...multiple different public keys with the same SKI within the SPKI table
  - ...an unsupported algorithm suite
- Generate a BGPsec signature from/without existing BGPsec path

For signing there are less test scenarios because this functionality does not rely on the SPKI table. Hence, error sources like missing or malformed public keys are not an issue.

The test cases are executed with static input data, meaning that all SKI, router keys and signatures are already present as byte sequences. These information are valid examples which are used in the RFC 8208 [9].

The byte sequences are used when assembling the BGPsec path elements, i.e., the secure path segments and the signature segments, for testing purposes. The rest of the BGPsec data, as shown in Listing 4.1 in Section 4.1, needs to be assembled as well. Listing 4.8 shows the process of assigning the data.

Listing 4.8: The BGPsec data necessary for AS path validation is assembled here.

```
1  struct signature_seg ss[2]
2  struct secure_path_seg sps[2]
3  struct bgpsec_data data
4
5  int as_hops = 2
6
7  nlri = { 0xC0,0x00,0x02 } // 192.0.2.0
8
9  ss[0]->ski          = ski1
10 ss[0]->sig_len      = 72
11 ss[0]->signature    = signature1
12
13 sps[0]->pcount      = 1
14 sps[0]->conf_seg    = 0
15 sps[0]->asn         = 65536
16
17 ss[1]->ski          = ski2
18 ss[1]->sig_len      = 72
19 ss[1]->signature    = signature2
20
21 sps[1]->pcount      = 1
22 sps[1]->conf_seg    = 0
23 sps[1]->asn         = 64496
24
25 bg->alg_suite       = 1
26 bg->afi             = 1
```

```
27 bg->safi           = 1
28 bg->asn            = 65537
29 bg->nlri_len       = 24
30 bg->nlri           = nlri
```

Using the `_bgpsec_segment_to_str` function, a BGPsec path segment can be printed out. Passing a segment that is filled with the data from Listing 4.8 will produce the following output.

Listing 4.9: A printed BGPsec path segment.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Signature Segment:
  SKI:
    47 F2 3B F1 AB 2F 8A 9D 26 86 4E BB D8 DF 27 11
    C7 44 06 EC

  Length: 72
  Signature:
    30 46 02 21 00 EF D4 8B 2A AC B6 A8 FD 11 40 DD
    9C D4 5E 81 D6 9D 2C 87 7B 56 AA F9 91 C3 4D 0E
    A8 4E AF 37 16 02 21 00 90 F2 C1 29 AB B2 F3 9B
    6A 07 96 3B D5 55 A8 7A B2 B7 33 3B 7B 91 F1 66
    8F D8 61 8C 83 FA C3 F1

----------------------------------------------------------------
Secure_Path Segment:
        pCount: 1
        Flags: 0
        AS number: 65536
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

4.4 Review

After a first review of the implementation, a few improvements have been made. While some of the changes affected the BGPsec API, others were performance or code improvements. The current state of this document already reflects the following changes.

An API change that has been made is the renaming of the `structs` and `enums`. All structs and enums that are publicly exposed to the API have been prefixed with `rtr_` to reduce the risk of collision with other software applications and libraries. Variables that are used during the build process of the RTRlib have also been prefixed for the same reason.

To make a clearer distinction between public API and private functionalities, RTRlib only exposes these symbols that are relevant for using the library. There is now a header file that only contains symbols like structs and enums, while the other, private header file contains the function definitions. To keep the functions accessible, the BGPsec implementation uses wrapper functions which are defined and exposed via the *rtr_mgr.c* file. These wrapper functions simply invoke the appropriate BGPsec function with the given parameters, e.g., `rtr_mgr_bgpsec_validate_as_path` invokes `rtr_bgpsec_validate_as_path`. The prefix changed from `rtr_` to `rtr_mgr_`.

The wrapper for the validation function received a change in parameters. Instead of the SPKI table it takes the RTR manager configuration. When invoking the actual validation function, the table is dereferenced and passed to the function, as shown in listing 4.10.

Listing 4.10: The truncated wrapper of the validation function.

```
1  int rtr_mgr_bgpsec_validate_as_path(...,
2                                      struct rtr_mgr_config *config,
3                                      ...)
4  {
5      int retval = rtr_bgpsec_validate_as_path(...,
6                                               config->spki_table,
7                                               ...);
8
9      return retval;
10 }
```

This allows the user to use the higher level `config` structure. Functionality tests on the other hand only require the SPKI table, so it is not necessary to have a whole configuration set up. By having the higher level function for the user and the lower level function for internal testing, both needs are satisfied.

Also for privacy reasons, constant BGPsec variables have been moved from the header to the source file. This prevents that they are unnecessarily exposed. If required, they can be obtained via the appropriate getter functions. Additionally, the moved constants

have been replaced with preprocessor macros or have been replaced entirely. For example, `SECURE_PATH_SEGMENT_SIZE` has been replaced with **`sizeof`** instructions on the secure path struct.

Stability has been improved by adding more NULL pointer checks on input parameters. If a user of the API accidentally passes a required parameter with value NULL to the validation or signing function, the function now returns a `RTR_BGPSEC_ERROR`. Previously, such cases would have led to a SEGFAULT during runtime.

Should BGPsec receive a new algorithm suite, the implementation has to be adjusted to meet the new requirements. The code has been slightly modified to make adding a new algorithm suite smoother and more convenient.

The following changes mainly focus on readability of the code and consistency with the rest of RTRlib. The public functions to check and get the algorithm suites have been renamed to for the sake of readability.

The previously used C++ single line comment style has been replaced with C block comment style. In addition, the documentation was expanded in a few places to support any potential users.

5 Verification

To make sure that the results produced by RTRlib are valid, they have to be verified by another BGPsec implementation. Therefore, the BGPsec reference implementation BGP-SRx is used. The BGP-SRx implementation will validate a signature that was generated by RTRlib and vice versa. In detail, the procedure will look like this:

1. Generate a signature A with RTRlib
2. Generate a signature B with BGP-SRx
3. Create a RTRlib test case with signature B
4. Create a BGP-SRx test case with signature A
5. Run both tests

If both implementations are correct, both test results should be valid. This section will cover the test cases in detail.

At first, we will need a signature generated by both implementations. To make sure that these signatures are created over the same information, the BGPsec path for RTRlib and BGP-SRx must be identical. It consists of

- Origin AS 65536
- Prefix 192.0.2.0/24
- Algorithm Suite 1
- AFI 1
- SAFI 1
- pCount 1
- Flags 0
- Target AS 65537

After constructing the BGPsec path, the RTRlib `rtr_mgr_bgpsec_generate_signature` function is executed. The result can be seen in Listing 5.1.

Listing 5.1: This test prints the byte sequence, the calculated digest and the generated signature for the constructed BGPsec path.

```
> ./tests/test_bgpsec
Byte Sequence:
00 01 00 01 01 00 00 01 00 00 01 00 01 01 18 C0
00 02

Digest:
2B 1C EE E2 72 FA D5 A1 C5 D6 7B 21 A8 92 EE 24
72 24 6A AA B0 0F 55 5A E0 1C C7 62 8F 76 E2 BD

Signature:
30 45 02 21 00 8A CC 91 63 A6 13 51 BF 89 E4 AC
97 32 F2 43 A9 E7 DA 35 5C B6 1D D8 87 42 19 5A
D6 26 C9 09 41 02 20 06 9D B4 AB DE 36 5C AB B2
31 5C 6A 54 8F 9B 1B 36 F3 5A ED 75 0B C6 5A E1
1C 3A BC 7B 81 B5 AD
```

The signature is generated over the digest, using the same private key that is available to BGP-SRx. Now that we have generated signature A, we will continue to generate signature B with BGP-SRx. BGP-SRx offers functionalities to validate and generate whole BGPsec updates and BGPsec paths. To produce a signature with the software, a whole BGPsec path is generated using the same path values as with RTRlib. Instead of writing a whole test, BGP-SRx allows to generate updates via input parameters, as can be seen in Listing 5.2.

Listing 5.2: BGP-SRx is called with various parameters to create a BGPsec path. The resulting path is dumped into a binary file.

```
> ./bgpsecio --mode GEN-C --asn 65536 --peer_asn 65537 --update "
    192.0.2.0/24" --out ~/bgpsecpath -f bgpsecio.conf
Starting bgpsecio 0.2.0.8...
Done.
```

The whole chain of parameters can be read as "*Generate* a BGPsec path from *AS 65536* to *AS 65537* with the *prefix 192.0.2.0/24* and dump the result to *~/bgpsecpath*". The `-f` parameter points to an additional config file that contains the rest of the required information.

Since the resulting file is binary encoded, it is loaded into a hex editor to make its contents visible. Figure 5.3 shows the hex representation of the generated BGPsec path.

Listing 5.3: Parts of the binary BGPsec path when dumped as hex. The highlighted bytes form the signatures. Top: the signature B (line 5-9) generated by BGP-SRx. Bottom: signature B was replaced with signature A (line 17-21) from RTRlib.

```
1  03 02 12 00 6D 00 00 00 01 00 76 00 01 00 00 00
2  ...
3  F2 3B F1 AB 2F 8A 9D 26 86 4E BB D8 DF 27 11 C7
4  44 06 EC 00 47
5  30 45 02 20 7E CA 01 0A EE 77 FF 14 46 87 92 CF
6  E2 DB 37 99 A5 08 61 F0 3C BC B4 2F 6A 73 7F A3
7  9A 64 E0 26 02 21 00 B4 29 97 67 36 7B 88 C3 66
8  EC 2F C4 98 A7 B1 8D A7 BB FA 22 78 BB D3 BC 4B
9  21 F5 71 34 55 B2 66
10
11 ----------------------------------------------
12
13 03 02 12 00 6D 00 00 00 01 00 76 00 01 00 00 00
14 ...
15 F2 3B F1 AB 2F 8A 9D 26 86 4E BB D8 DF 27 11 C7
16 44 06 EC 00 47
17 30 45 02 21 00 8A CC 91 63 A6 13 51 BF 89 E4 AC
18 97 32 F2 43 A9 E7 DA 35 5C B6 1D D8 87 42 19 5A
19 D6 26 C9 09 41 02 20 06 9D B4 AB DE 36 5C AB B2
20 31 5C 6A 54 8F 9B 1B 36 F3 5A ED 75 0B C6 5A E1
21 1C 3A BC 7B 81 B5 AD
```

The generated Signature B (top of Listing 5.3) is extracted and saved and will be later used for an RTRlib test. The signature is then replaced with signature A.

Both implementations now validate each others signatures. BGP-SRx has the ability to read in a BGPsec path in binary form and validate it. No other information except the signature has been modified. Listing 5.4 shows the validation result of BGP-SRx using signature A, which was created by RTRlib. Listing 5.5 shows the validation result of RTRlib using Signature B, which was created by BGP-SRx.

Listing 5.4: The validation output of BGP-SRx after successfully validating the signature that was generated by RTRlib.

```
Hash(validate):
00 01 00 01 01 00 00 01 00 00 01 00 01 01 18 C0
00 02

Digest(validate):
2b 1c ee e2 72 fa d5 a1 c5 d6 7b 21 a8 92 ee 24
72 24 6a aa b0 0f 55 5a e0 1c c7 62 8f 76 e2 bd

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


Statistics Valid:
====================
  1 updates (1 segments) in 395769 ns processed
  - average time per update:  395769 ns
  - average time per segment: 395769 ns
  - average number of segments per update: 1.00
  - segments per second: 2526

Done.
```

Listing 5.5: RTRlib output after successfully validating signature B, generated by BGP-SRx.

```
Byte Sequence:
00 01 00 01 01 00 00 01 00 00 01 00 01 01 18 C0
00 02

Digest:
2B 1C EE E2 72 FA D5 A1 C5 D6 7B 21 A8 92 EE 24
72 24 6A AA B0 0F 55 5A E0 1C C7 62 8F 76 E2 BD
```

```
(2018/08/24 11:20:29:484858): BGPSEC: Validation result of
   signature: valid
(2018/08/24 11:20:29:484908): BGPSEC: Validation result for the
   whole BGPsec_PATH: valid
Test successful
```

Note, that both byte sequences (top row) and digests (second row from the top) from BGP-SRx and RTRlib are identical. With both implementations successfully validating each others signatures it is safe to say that signing and validating works for RTRlib.

6 Performance

First measurements show the performance of both the RTRlib as well es BGP-SRx implementation. For the tests, each implementation will validate and sign AS paths with a length ranging from 1 up to 5. The time measured is the processing time for a single validation/signing procedure. An average time over 500 executions is then calculated. It is important to mention that only the duration of the validation/signing function is measured, no initialization or setup is included.

Figure 6.1 shows the performance when validating an AS path. The x-axis shows the path length, the y-axis shows the processing time in microseconds. BGP-SRx performs faster than the current implementation of RTRlib by a factor ≈ 1.7. Both implementations scale linearly with the increasing path length of an AS path.

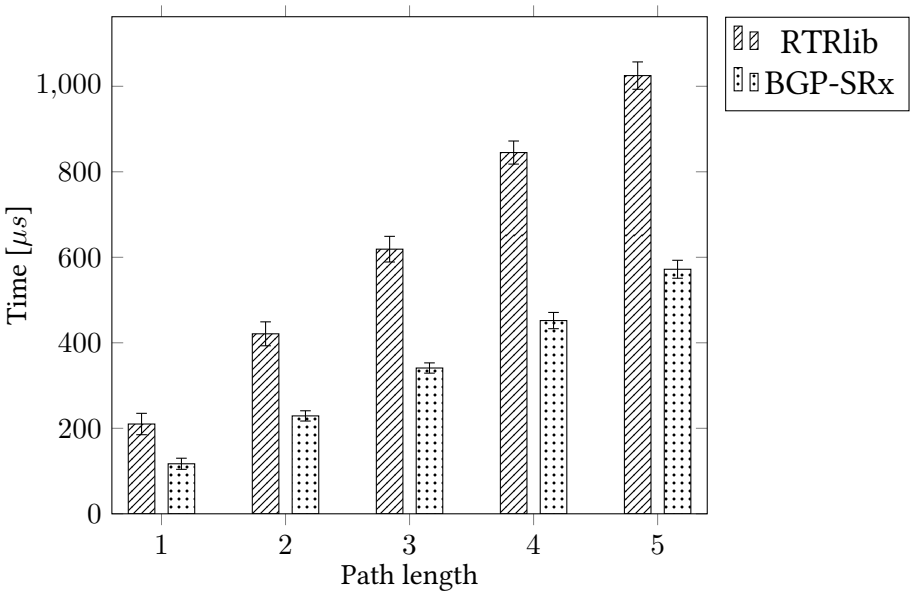

Figure 6.1: The bars show the mean execution time of a single AS path *validation* for different path length. A total of 500 iterations were measured. The error bars show the standard deviation.

When looking at the signing procedure in Figure 6.2, BGP-SRx outperforms RTRlib, being faster by a factor of ≈ 2.4. The path length does not have an impact on the signing operation due to the BGPsec signing design. Signing is not an iterative process like validation, because the hashing and signing only occurs once over the whole path, rendering the length of the path neglectable.

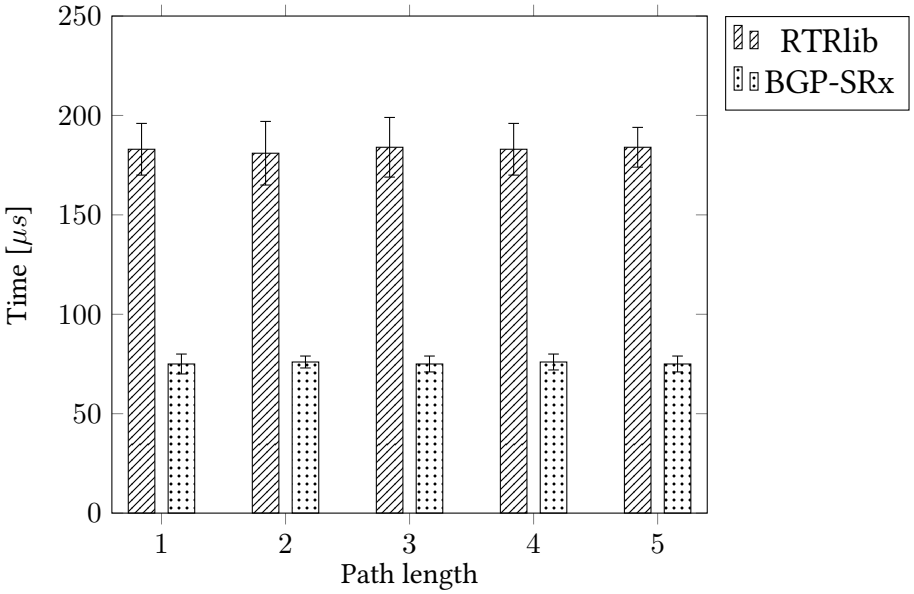

Figure 6.2: The bars show the mean execution time of a single AS path *signing* for different path length. Again, a total of 500 iterations were measured. The error bars show standard deviation.

The results show that the behavior of both implementations is the same. For validation, the duration scales linearly with the increasing path length. Signing has a constant processing duration, independent from the path length.

The implementation of AS path validation for RTRlib can and should be improved. The reasons for the performance discrepancies between both implementations are subject to a more in-depth analysis. Saving processing time is crucial for BGPsec implementations as BGP routers are under heavy load. Performing cryptographic operations increases this load even more.

Further to performance evaluations, the tests should be expanded to cover more scenarios, preferably in a dedicated environment.

7 Conclusion and Outlook

This work has presented a working implementation of BGPsec AS path validation for RTRlib. Before an integration into the library happens, further code reviews and more detailed documentation have to take place. The main requirements that were made in Section 2 are mostly met. These were:

1. Lightweight, only provide validation & signing of BGPsec paths (met)
2. Independent from other applications (met)
3. Intuitive API, similar to current one (met)
4. No changes to the current implementation (met)
5. Detailed documentation (WIP)

Requirement 1. is met since the implementation provides no more functionalities than validating and signing BGPsec paths.

Next, 2. is also met since no dependencies other than the OpenSSL crypto library exist. RTRlib AS path validation is not designed to fit any specific software but allows anyone to integrate it into their application. This is achieved by RTRlib providing all required data structures and specifying a certain format to handle the data.

Point 3. is more of a subjective matter, but it can be argued that the new API is very similar to the existing one. The workflow of the initial setup for RTRlib does also not require additional steps.

The 4. requirement has also been successfully met, as there are no changes to the current code base for AS path validation. In fact, there are no interferences or dependencies between AS path validation and route origin validation.

As for the last requirement 5., the documentation is still work in progress and will be expanded as more code reviews happen in the near future. The current documentation, which is generated with Doxygen, gives details about the function names, their parameters and return values. More in depths information to the usage of the functions and the workflow is required, though.

The actual functionality of the implementation was demonstrated in Section 5 and should suffice for the moment. It is for the future, however, indispensable to establish a testing

environment to constantly test the implementation throughout. As the tests from the previous chapter have shown, there are still improvements to be made to the implementation. The next steps are to check, which operations can be improved, and how. Also, tests are required that measure more parameters, such as memory or disk space usage.

In conclusion, this implementation is extending RTRlib by more RPKI functionalities, namely BGPsec AS path validation and signing.

Bibliography

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

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

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

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

[5] 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

[6] 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).

[7] NIST, "Digital Signature Standard," Federal Information Processing Standards 186–4, July 2013.

[8] National Institute of Standards and Technology, "FIPS 180–3, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-3," http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf, Department of Commerce, Gaithersburg, MD, US, Tech. Rep., October 2008.

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

[10] OpenSSL, "OpenSSL Documentation." [Online]. Available: https://www.openssl.org/docs/man1.0.2/apps/openssl.html (Accessed 03-08-2018).

[11] O. Borchert and M. Baer, "Subject: Modification request: draft-ietf-sidr-bgpsec-protocol-14," Feb 2016. [Online]. Available: https://mailarchive.ietf.org/arch/msg/sidr/8B_e4CNxQCUKeZ_AUzsdnn2f5Mu (Accessed 18-10-2018).