Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Projektbericht 2

Martin Landsmann

Komponenten zum sicheren Routing und Forwarding mit RIOT

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Martin Landsmann

Komponenten zum sicheren Routing und Forwarding mit RIOT

Eingereicht am: 13. Juni 2016

Inhaltsverzeichnis

1 Netz-Einträge in der FIB

Jede Instanz eines Routing-Protokolls verwaltet immer nur ein eindeutiges Präfix. Die Implementierung eines Routing-Protokolls muss dafür Sorge tragen, dass sich Präfixe mehrerer Instanzen nicht überschneiden. Insbesondere muss das auch gewährleistet sein wenn unterschiedliche Routing-Protokolle auf einem Knoten zum Einsatz kommen. In jedem verwalteten Präfix können Netze definiert sein, hinter denen sich Host-Adressen sowie weitere Sub-Netze befinden.

1.1 Problemstellung

Die FIB kann keine Netz-Einträge von Host-Einträgen unterscheiden. Damit kann eine Zieladresse nur erreicht werden, wenn explizit ein Eintrag für sie in der FIB steht. Hosts, die in vom Knoten aus erreichbaren Netzen liegen, müssen explizit mit ihrer Zieladresse in die FIB eingetragen werden. Aggregieren von Zieladressen in eine Netz-Route wie es durch das *Classless Inter-Domain Routing* (CIDR)[1] für IPv4 sowie für IPv6[2] eingesetzt wird ist nicht möglich. Daraus ergeben sich zwei zentrale Probleme:

- Die FIB-Tabelle wird mit aggregierbaren Einträgen befüllt. Damit wird der zumeist knappe Speicher eines Sensorknotens mit redundanter Information belegt.
- Die Suche nach einem next-hop in der FIB ist angesichts der nicht zusammengefassten Einträge aufwendiger. Mit steigender Anzahl an iterierbaren Einträgen, steigt die durchschnittliche Dauer der Suche nach einem passendem next-hop.

1.2 Anforderungen

- Die FIB muss Netz-Einträge verwalten können. Werden derzeit Netz-Adressen als Zieladresse eingetragen, wird der erzeugte FIB-Eintrag nicht als Netz erkannt oder genutzt, sondern wie ein Host-Eintrag behandelt. Es ist erforderlich, dass Netz-Einträge kenntlich gemacht werden, um sie von Host Zieladressen zu unterscheiden. RIOT[3] nutzt primär IPv6 Adressen. Somit muss die Unterscheidung des Präfix zwischen einer Host- und einer Netz-Eintrag durch IPv6-Präfixe umgesetzt werden. Es muss jedoch weiterhin gewährleistet sein, dass die Einträge der FIB nicht umfassend auf eine Adressart, wie IPv6 oder IPv4, eingeschränkt werden und erweiterbar bleiben.
- Die FIB muss Anfragen nach next-hops zu Zieladressen auf einen passenden Netz-Eintrag abbilden können, um diesen als möglichen next-hop zu identifizieren. Dafür muss eine

Präfix-basierte Suche implementiert werden, die gesuchte Zieladressen im Präfix eines Netz-Eintrags finden kann. Bei einer Anfrage nach einem next-hop ist ein FIB-Eintrag der explizit die Zieladresse des gesuchten Hosts beinhaltet, immer gegenüber einem Netz-Eintrag vorzuziehen. Des weiteren ist bei der Suche ein größerer Präfix, immer einem kleinerem Präfix vorzuziehen.

1.3 Konzept

Zur Unterscheidung zwischen Host- und Netz-Einträgen in der FIB, wird im Flag für die Zieladresse die Länge des Präfix gespeichert, vgl. Listing 1.1 `global_flags`. Diese wird in das oberste Byte der Flags eingetragen, vgl. Listing 1.2. Ist der Wert des Byte > 0 und kleiner der Adresslänge in Bits, handelt es sich bei der Zieladresse eindeutig um einen Netz-Eintrag. Entspricht der Wert genau der Länge der Zieladresse in Bits, handelt es sich bei der Zieladresse eindeutig um einen Host-Eintrag. Ist der Wert im Byte des Flags 0, muss die Zieladresse selbst zur Unterscheidung hinzugezogen werden. Hierbei sind zwei unterscheidbare Fälle möglich:

- Ist ein Bit der Zieladresse ungleich 0 handelt es sich um einen Host-Eintrag.
- Besteht die Zieladresse nur aus Nullen handelt es sich im Fall von IP um die *Unsepecified Address*. Diese Zieladresse wird von der FIB als *Default-Route* für den eingesetzten Adresstyp genutzt. In die *Default-Route* bilden alle Adressen des gleichen Adress-Typs ab.

Wird ein neuer FIB-Eintrag erstellt, muss das für die Präfix-Länge reservierte Byte der Flags für die Zieladresse entsprechend gesetzt sein. Bei der Suche nach einem next-hop benutzt die FIB diesen Wert zur Unterscheidung zwischen einem Host- und Netz-Eintrag sowie für den Vergleich der Zieladresse.

Listing 1.1: Definition eines FIB-Eintrags

```
1  /**
2   * @brief Container descriptor for a FIB entry
3   */
4  typedef struct fib_entry_t {
5      /** interface ID */
6      kernel_pid_t iface_id;
7      /** Lifetime of this entry (an absolute time-point is stored by the FIB) */
8      uint64_t lifetime;
9      /** Flags of the global address */
10     uint32_t global_flags;
11     /** Pointer to the shared generic address */
12     struct universal_address_container_t *global;
13     /** Flags of the next hop address */
14     uint32_t next_hop_flags;
15     /** Pointer to the shared generic address */
16     struct universal_address_container_t *next_hop;
17 } fib_entry_t;
```

Listing 1.2: Belegung des Flag ’uint32_t global_flags’

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                 | Prefix length |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

1.4 Umsetzung

Die Suche nach einem next-hop in der FIB ist dahingehend erweitert worden, dass die Länge des eingetragenen Präfix zum Vergleich hinzugezogen wird[1]. Sucht die FIB nach einem next-hop zu einer Zieladresse, wird immer zuerst der Flag der Zieladresse eines Eintrags zum Vergleich ausgewertet.
Daraufhin wird per *longest common prefix match* (LPM) ein passender next-hop Eintrag zur gesuchten Zieladresse gesucht. Ist die gesuchte Zieladresse identisch zur Zieladresse des Eintrags, ist es die am besten passende Host-Eintrag zur gesuchten Zieladresse und die Suche ist erfolgreich abgeschlossen. Die FIB gibt den next-hop des Eintrags direkt als Ergebnis der Suche zurück. Im Fall dass die Länge eines passenden Präfix in den FIB-Einträgen kleiner der gesuchten Zieladresse ist, wird die Suche fortgesetzt bis der next-hop des längsten passenden Präfix als Ergebnis der Suche zurückgegeben wird.
Ist die Länge des Präfix im FIB-Eintrag als 0 angegeben, wird überprüft ob die Zieladresse des Eintrags die *Default-Route* ist. Ist ein Bit in der Zieladresse des Eintrags ungleich 0, behandelt die FIB den Eintag als Host-Eintrag. Sind alle Bits der Zieladresse des Eintrags 0, handelt es sich um den *Default-Route* Eintrag. Dessen next-hop wird nur als Ergebnis zurückgegeben wenn die LPM-Suche sonst keinen besseren Eintrag findet. Wurde kein passender Eintrag gefunden, gibt die FIB den entsprechenden Fehlercode zurück.
Es sei darauf hingewiesen, dass die FIB keine Plausibilitätsüberprüfung der Flags vornimmt. Die Verantwortung über die Plausibilität des Flag liegt beim Routing-Protokoll. Das Reservierte Byte der Flags muss ”richtig” gesetzt sein, damit die FIB einen Eintrag als Host-, Netz- oder *Default-Route* identifizieren kann.

[1] https://github.com/RIOT-OS/RIOT/pull/4279

2 Source-Routen verwalten mit der FIB

Im IoT werden Routing-Protokolle eingesetzt, die Source-Routen verwenden. Zu diesen gehören beispielsweise das *Dynamic Source Routing* Protokoll (DSR) [4] sowie das *Routing-Protokoll for Low-Power and Lossy Networks* (RPL) [5] Source-Routen können derzeit nicht in der FIB gespeichert oder verwaltet werden.

2.1 Problemstellung

Die Länge einer Source-Route ist nicht feststellbar bevor sie tatsächlich gelernt wurde, da sie eine variable Anzahl an Hops zwischen dem 1. Hop und dem Ziel haben kann. Effizientes Festlegen des Speichers für eine einzelne Source-Route ist damit vorab nicht möglich.
Eine Zieladresse in einer Topologie kann über unterschiedliche Source-Routen erreichbar sein. Die Unterscheidung einzelner Source-Routen allein über ihre jeweiligen Zieladressen, so wie für FIB-Einträge beim einfachen Routing, ist so nicht möglich. Für Operationen an einzelnen Source-Routen ist es jedoch erforderlich, dass die FIB unterscheiden kann, welche der Source-Routen verändert werden soll. Eine Anfrage nach einer Source-Route zu einer gegebenen Zieladresse, kann mehrere Source-Routen als Ergebnis haben. Diese Mehrdeutigkeit des Ergebnisses ist durch die FIB nicht auflösbar, da sie die Bewertungskriterien des Routing-Protokolls die zum Lernen der Routen eingesetzt wurden nicht kennt.

2.2 Anforderungen

- Eine Implementierung zum Verwalten und Speichern von Source-Routen in der FIB, muss erlauben dass eine variable Anzahl an Hops in einer Source-Route gespeichert werden kann. Dabei soll der Speicher nicht dynamisch alloziert werden und muss vollständig zur Compilezeit festlegbar sein.

- Source-Routen Einträge müssen automatisch zum überschreiben freigegeben werden können, wenn die Lebensdauer einer Source-Route abläuft. Damit werden Aufrufe seitens des Routing-Protokolls vermieden, wenn beim Erstellen der Source-Route eine endliche Lebensdauer eingestellt wurde.

- Die FIB muss Source-Routen, die zu der gleichen Zieladresse führen, verwalten und eindeutig unterscheiden können. Eine Unterscheidung über die Zieladresse im Zusammenhang mit dem Hop-count ist bspw. nicht eindeutig, da zwei unterschiedliche Source-Routen zu einer Zieladresse einen identischen Hop-count haben können. Im Hinblick auf den potentiell sehr eingeschränkt verfügbaren Speicher eines Knotens, wird ein Vergleich

der kompletten Source-Routen zur Unterscheidung Kategorisch ausgeschlossen und muss vermieden werden. In DSR wird bspw. jeder Source-Route eine eindeutige ID zugewiesen, durch die sie von anderen Source-Routen mit der gleichen Zieladresse einfach zu unterscheiden ist. Erst wenn Source-Routen eindeutig und effizient unterscheidbar sind, wird es möglich gezielt eine angeforderte Operation durch ein Routing-Protokoll auf der ”richtigen” Source-Route auszuführen.

- Routing-Schleifen müssen beim Erstellen einer Source-Route durch die FIB verhindert werden. Hierfür muss die FIB sicherstellen, dass jeder Hop nur einmal innerhalb einer Source-Route vorkommt. Beim Einfügen eines Hops in eine Source-Route, muss die Einzigartigkeit des neuen Hops durch entsprechende Überprüfungen gewährleistet werden.

2.3 Konzept der Umsetzung

Die FIB-Implementierung wurde erweitert, sodass Source-Routen in einer FIB-Tabelle gespeichert und verwaltet werden können, vgl. Listing 2.1. Der Speicher für die Tabelle wird außerhalb der FIB angelegt und der FIB als Parameter zur Verfügung gestellt. Die Tabelle kann so einem bestimmten Bereich im RIOT Netzwerkstack zugeordnet sein, bspw. dem `gnrc_ipv6`-Modul. Die Source-Routen in der Tabelle sind so einem spezifischen Bereich im RIOT Netwerkstack zugeordnet, bspw. allen Implementierungen die IPv6 nutzen wenn die Tabelle im `gnrc_ipv6`-Modul angelegt ist. Zugriffe zum Verändern oder Suchen von Einträgen in der FIB werden so spezifisch für IPv6 gestellt, wenn die Tabelle des aus dem `gnrc_ipv6`-Modul als Parameter zu der gewünschten Operation mit angegeben wird. Damit ist jede Operation direkt auf die passende Tabelle beschränkt und verhindert rechenintensive Iterationen und Suchen in inkompatiblen Tabellen, bspw. hinzufügen einer IPv6 Source-Route in eine IPv4 Tabelle oder die Anfrage nach einer Source-Route zu einer inkompatiblen Zieladresse wie einer *Bluetooth low energy*[1] 48Bit Adresse.
Eine Tabelle hat einen Gemeinsamen Speicher für alle Hop-Einträge ihrer Source-Routen. Dies gewährleistet eine optimale Speicherbelegung für variabel lange Source-Routen. Jede Source-Route belegt nur den Speicher in der Tabelle, den sie für die Anzahl ihrer Hops benötigt. Jede Source-Route in der Tabelle hat einen Header. Dieser enthält:

- Die Interface ID. Sie gibt an, welche Kommunikationsschnittstelle mit der Source-Route verwendet werden soll.
- Die Lebensdauer der Source-Route, nach der sie automatisch durch die FIB überschrieben werden kann.
- Ein Flags Feld, in dem spezifische Eigenschaften der Source-Route festgelegt werden können.
- Den 1. Hop der Source-Route.

[1] https://www.bluetooth.com/specifications/adopted-specifications

- Die Zieladresse der Source-Route.

Vergleiche hierzu Listing 2.2.
Die Hops einer Source-Route werden in einer einfach verketteten Liste gespeichert. Die Reihenfolge der Einträge einer Liste vom 1. Hop bis zur Zieladresse entsprechen den aufeinanderfolgenden Hops der Source-Route. Der 1. Hop in dem zur Liste gehörendem Header zeigt auf den Listenanfang. Die Zieladresse des Headers auf das Listenende. Jedes Listenelement hat einen Zeiger auf `universal_address_container_t` typen, in denen die tatsächlichen Adressdaten abgelegt sind, vgl. Listing 2.3. Damit wird sichergestellt, dass der benötigte Speicher für eine Hop-Adresse nur ein einziges mal belegt ist, wenn die Adresse in mehreren Source-Routen vorkommt.

Listing 2.1: Definition einer FIB-Tabelle

```
/**
* @brief Meta information of a FIB table
*/
typedef struct {
    /** A single hop OR source route data array */
    union{
        /** array holding the FIB entries for single hops */
        fib_entry_t *entries;
        /** array holding the FIB entries for source routes */
        fib_sr_meta_t *source_routes;
    }data;
    /** the kind of this FIB table, single hop or source route.
    *   This value indicates what is stored in ‘data‘ of this table
    */
    uint8_t table_type;
    /** the maximim number of entries in this FIB table */
    size_t size;
    /** table access mutex to grant exclusive operations on calls */
    mutex_t mtx_access;
    /** current number of registered RPs. */
    size_t notify_rp_pos;
    /** the kernel_pid_t of the registered RPs.
    *   Used to notify the RPs by the FIB on certain conditions,
    *   e.g. when a destination is unreachable
    */
    kernel_pid_t notify_rp[FIB_MAX_REGISTERED_RP];
    /** the prefix handled by each registered RP.
    *   Used to dispatch if the RP is responsible for the condition,
    *   e.g. when the unreachable destination is covered by the prefix
    */
    universal_address_container_t* prefix_rp[FIB_MAX_REGISTERED_RP];
} fib_table_t;
```

Listing 2.2: Definition eines FIB Source-Route Header

```
/**
* @brief Container descriptor for a FIB source route
*/
typedef struct fib_sr_t {
    /** interface ID */
    kernel_pid_t sr_iface_id;
    /** Lifetime of this entry (an absolute time-point is stored by the FIB) */
```

```
8     uint64_t sr_lifetime;
9     /** Flags for this source route */
10    uint32_t sr_flags;
11    /** Pointer to the first hop on the source route */
12    struct fib_sr_entry_t *sr_path;
13    /** Pointer to the destination of the source route */
14    struct fib_sr_entry_t *sr_dest;
15 } fib_sr_t;
```

Listing 2.3: Definition eines Eintrags in der Source-Route

```
1 /**
2  * @brief Container descriptor for a FIB source route entry
3  */
4 typedef struct fib_sr_entry_t {
5     /** Pointer to the shared generic address */
6     struct universal_address_container_t *address;
7     /** Pointer to the next shared generic address on the source route */
8     struct fib_sr_entry_t *next;
9 } fib_sr_entry_t;
```

Eintragen einer Source-Route in die FIB Zuerst wird durch die FIB ein Header für die Source-Route angelegt. Hierfür wird der FIB die genutzte Tabelle, die Interface ID, die Flags sowie die Lebensdauer für die neue Source-Route übergeben, vgl. Listing 2.4. Die FIB wählt den 1. verfügbaren freien Speicherplatz für die Source-Route in der übergebenen Tabelle. Dort speichert sie den Header für die neue Source-Route mit den übergebenen Parametern. Ist die Lebensdauer abgelaufen, markiert die FIB die Einträge der Source-Route als gelöscht und frei zum überschreiben. Soll die FIB die Source-Route nicht automatisch als frei und Überschreibbar markieren, kann die Lebensdauer auf `FIB_LIFETIME_NO_EXPIRE` (unendlich) festgelegt werden. Ist die Lebensdauer der Source-Route auf unendlich festgelegt, kann sie nur noch vom entsprechenden Routing-Protokoll zum überschreiben freigegeben werden. Nach dem Erfolgreichen Eintragen des Headers, gibt die FIB den Zeiger auf den Header als eindeutige ID der Source-Route zurück, vgl. Listing 2.4 Parameter 2 (`fib_sr`). Zwei unterschiedliche Zeiger auf Header gehören so immer zu unterschiedlichen Source-Routen. Der Zeiger zum Header wird immer zusammen mit der Tabelle an die FIB für Operationen an Einträgen der entsprechenden Source-Route übergeben. Das gewährleistet dass Änderungen nur von dem Routing-Protokoll vorgenommen werden können welches die Source-Route erstellt hat, und die Source-Route eindeutig durch das Routing-Protokoll und die FIB identifizierbar ist. Die Interface ID, die Flags sowie die Lebensdauer können durch das Routing-Protokoll nachträglich verändert werden[2]. Ist der Header erstellt, werden die Hops der Source-Route eingetragen.

Listing 2.4: Funktion zum erstellen einer FIB Source-Route

```
1 /**
2  * @brief creates a new source route
3  *
4  * @param[in, out] table the table the new source route belongs to
```

[2] https://github.com/RIOT-OS/RIOT/blob/master/sys/include/net/fib.h#L274

```
 * @param[in, out] fib_sr pointer to store the new created source route pointer
 * @param[in] sr_iface_id the interface ID used for the created source route
 * @param[in] sr_flags the flags for the source route
 * @param[in] sr_lifetime the lifetime in ms of the source route
 *
 * @return 0 on success
 *         -EFAULT on wrong parameters, i.e. fib_sr is NULL and/or sr_lifetime is 0
 *         -ENOBUFS on insufficient memory, i.e. all source route fields are in use
 */
int fib_sr_create(fib_table_t *table, fib_sr_t **fib_sr, kernel_pid_t sr_iface_id,
                  uint32_t sr_flags, uint32_t sr_lifetime);
```

Hinzufügen von Hops in eine Source-Route Zum hinzufügen von Hops in eine Source-Route wird immer die Tabelle sowie der zuvor erstellte Header der Source-Route als Parameter übergeben. Ein neuer Hop kann an den Listenanfang vorangestellt, vgl. Lising 2.5, hinter einen bestehenden Eintrag oder an das Listenende der Source-Route angehängt werden, vgl. Listing 2.6. Bei jeder dieser Operationen zum Einfügen eines Hops, wird die komplette Source-Route nach der Adresse des neuen Hops durchsucht. Existiert sie bereits in der Source-Route, wird die Aktion abgebrochen. Der Aufrufer wird durch den Rückgabewert `-EINVAL` darüber informiert, dass der Eintrag nicht eingefügt wird da sonst eine Routing-Schleife entsteht. Beim voranstellen und anhängen eines Eintrags in die Source-Route wird entsprechend der Zeiger des Headers zum 1. Hop oder zur Zieladresse im Header angepasst. Um hinter einen bestehenden Hop der Source-Route einen neuen Hop einzufügen, implementiert die FIB einen Iterator für die Source-Routen. Ein Routing-Protokoll kann zu jedem Hop der Source-Route iterieren und die eingetragene Adresse des Hops von der FIB erhalten[3]. Das Iterator-Element hinter dem der neue Eintrag eingefügt werden soll wird mit der Adresse des neuen Hops an die FIB übergeben. Zusätzlich gibt das Routing-Protokoll per Parameter an, ob die übrige Source-Route hinter dem neuem Hop angehängt werden soll bestehen bleibt und sich damit die ursprüngliche Zieladresse der Source-Route nicht ändert, oder ob der neue Hop das Listenende und damit die neue Zieladresse der Source-Route ist. Soll der neue Hop das Listenende sein, werden alle nachfolgenden Einträge verworfen.

Listing 2.5: Funktion zum anhängen eines Hop an eine Source-Route

```
/**
 * @brief append a new entry at the end of the source route, i.e. a new destination
 *
 * @param[in] table the table with the source route to append the new entry
 * @param[in] fib_sr pointer to the sr to append a hop address
 * @param[in] dst pointer to the new destination address bytes
 * @param[in] dst_size the size in bytes of the destination address type
 *
 * @return 0 on success
 *         -EINVAL on the given destination is already on the path in the source
    route
 *         -ENOENT on expired lifetime of the source route
 *         -EFAULT on fib_sr and/or dst is NULL
 */
int fib_sr_entry_append(fib_table_t *table, fib_sr_t *fib_sr,
```

[3] https://github.com/RIOT-OS/RIOT/blob/master/sys/include/net/fib.h#L405

```
15                          uint8_t *dst, size_t dst_size);
```

Listing 2.6: Funktion zum hinzufügen eines Hop in eine Source-Route

```
1  /**
2  * @brief adds a new entry behind a given sr entry
3  *
4  * @param[in] table the table with the source route to add the new entry
5  * @param[in] fib_sr pointer to the sr to add a hop address
6  * @param[in] sr_path_entry pointer to the entry after which we add the new one
7  * @param[in] addr pointer to the new address bytes
8  * @param[in] addr_size the size in bytes of the address type
9  * @param[in] keep_remaining_route indicate if the remaining route after
       sr_path_entry
10 *            should be kept and appended after the new entry
11 *
12 * @return 0 on success
13 *         -EFAULT on fib_sr and/or sr_path_entry and/or addr is NULL
14 *         -ENOENT on expired lifetime of the source route
15 *         -EINVAL on the given address is already present in the path
16 */
17 int fib_sr_entry_add(fib_table_t *table, fib_sr_t *fib_sr,
18                      fib_sr_entry_t *sr_path_entry, uint8_t *addr, size_t
                         addr_size,
19                      bool keep_remaining_route);
```

Löschen von Hops aus einer Source-Route Zum Löschen eines Hop aus einer Source-Route übergibt das Routing-Protokoll die zu löschende Adresse. Zusätzlich wird angegeben ob die übrige Source-Route hinter dem zu löschenden Hop an seinen Vorgänger angehängt werden soll, vgl. Listing 2.7. Wählt das Routing-Protokoll den 1. Hop zum löschen ohne die übrige Source-Route beizubehalten, wird die gesamte Source-Route gelöscht. Soll der 1. Hop gelöscht werden und die übrige Source-Route beibehalten werden, wird entsprechend der Nachfolgende Hop des gelöschten zum neuen 1. Hop der Source-Route. Wird ein Hop zwischen 1. Hop und der Zieladresse zusammen mit der übrigen Source-Route gelöscht, wird automatisch der Vorgänger des gelöschten Hops zum neuem Ziel der Source-Route.

Listing 2.7: Funktion zum löschen eines Hop aus einer Source-Route

```
1  /**
2  * @brief removes an entry from a source route
3  *
4  * @param[in] table the fib instance to access
5  * @param[in] fib_sr pointer to the sr to delete a hop address
6  * @param[in] addr pointer to the address bytes to delete
7  * @param[in] addr_size the size in bytes of the address type
8  * @param[in] keep_remaining_route indicate if the remaining route
9  *            should be kept and appended after the predecessor of the removed
       entry
10 *
11 * @return 0 on success
12 *         -EFAULT on one of the passed pointers is NULL
13 *         -ENOENT on expired lifetime of the source route
14 */
15 int fib_sr_entry_delete(fib_table_t *table, fib_sr_t *fib_sr, uint8_t *addr,
```

```
16                              size_t addr_size,
17                              bool keep_remaining_route);
```

Aktualisieren von Hops in einer Source-Route Die Adresse eines Hop innerhalb einer Source-Route kann durch das Routing-Protokoll verändert werden. Hierfür wird der FIB die Adresse des Hops sowie die neue Adresse, welche den Hop ersetzen soll als Parameter übergeben, vgl. Listing 2.8. Die FIB durchsucht die Source-Route nach dem Hop mit der alten Adresse und überprüft dabei ob die neue Adresse nicht schon in der Source-Route vorhanden ist. Ist der Hop gefunden, wird die übrige Source-Route nach der neuen Adresse durchsucht. Ist die neue Adresse für den Hop noch nicht in der Source-Route, wird die Adresse des Hop mit der neuen Adresse ersetzt. Ansonsten wird der Aufrufer durch den Rückgabewert `-EINVAL` darüber informiert, dass der Hop nicht mit der neuen Adresse überschrieben wird da sonst eine Routing-Schleife entsteht.

Listing 2.8: Funktion zum verändern der Adresse eines Hop in der Source-Route

```
1
2  /**
3   * @brief overwrites the address of an entry with a new address
4   *
5   * @param[in] table the fib instance to access
6   * @param[in] fib_sr pointer to the sr to overwrite a hop address
7   * @param[in] addr_old pointer to the address bytes to overwrite
8   * @param[in] addr_old_size the size in bytes of the address type
9   * @param[in] addr_new pointer to the new address bytes
10  * @param[in] addr_new_size the size in bytes of the address type
11  *
12  * @return 0 on success
13  *          -EINVAL on the given address is already present in the path
14  *          -ENOMEM on no memory left to create a new address entry to overwrite the
        old one
15  *          -EFAULT on one of the passed pointers is NULL
16  *          -ENOENT on expired lifetime of the source route
17  */
18 int fib_sr_entry_overwrite(fib_table_t *table, fib_sr_t *fib_sr,
19                            uint8_t *addr_old, size_t addr_old_size,
20                            uint8_t *addr_new, size_t addr_new_size);
```

Anfrage nach einer Source-Route zu einer gegebenen Zieladresse Als Parameter bekommt die FIB die zu durchsuchende Tabelle, gesuchte Flags, einen Zeiger zum Speicher in den die Adressen der Source-Route gespeichert werden sollen und einen Parameter, ob die Adressen in umgekehrter Reihenfolge, Zieladresse zum 1. Hop, in den Speicher geschrieben werden sollen, vgl. Listing 2.9.
Die FIB durchsucht zunächst alle Header in der übergebenen Tabelle nach einem Eintrag in dem die gesuchte Zieladresse mit der eines Headers übereinstimmt. Für jeden passenden Header werden die Flags miteinander verglichen. Stimmen auch die Flags überein schreibt die FIB sukzessive, oder falls parametriert in umgekehrter Reihenfolge, die Adressen der Source-Route Hops in den Speicher des übergebenen Zeigers. Der Zeiger zu der gefundenen Source-Route

wird an den Aufrufer zurückgegeben.
Stimmen die gesuchten Flags nicht mit denen des Headers überein, speichert die FIB den Zeiger zu der Source-Route mit der gesuchten Zieladresse temporär ab und setzt die Suche fort. Wird keine Source-Route mit gleichen Flags und passender Zieladresse gefunden, schreibt die FIB sukzessive in parametrierter Reihenfolge die Adressen der temporär gespeicherten Source-Route in den Speicher des Aufrufers, und gibt den Zeiger zur Source-Route zurück. Der Aufrufer wird mit einem Rückgabewert informiert, dass die Source-Route zu der gesuchten Zieladresse passt, jedoch die Flags nicht.
Potentiell sind noch weitere passende Source-Routen in der FIB, deren Flags nicht mit den gesuchten übereinstimmen. Das Routing-Protokoll kann sukzessive alle eingetragenen Source-Routen zu der gegebenen Zieladresse anfordern. Hierfür wird die Anfrage nach einer Source-Route an die FIB mit dem zuvor zurückgegebenen Zeiger auf die letzte gefundene Source-Route als Parameter aufgerufen. Die FIB durchsucht die Source-Routen nach einem passenden Eintrag zur gesuchten Zieladresse. Dabei überspringt sie die übergebene Source-Route und gibt, falls vorhanden, den Zeiger zur nachfolgend passenden Source-Route und der Adressen ihrer Hops in der parametrierten Reihenfolge zurück. Damit können alle zu einer Zieladresse passenden Sorce-Routen sukzessive von der FIB angefordert werden.

Existiert keine passende Source-Route für die gesuchte Zieladresse in der Tabelle, werden die Präfixe der registrierten Routing-Protokolle nach dem zur Zieladresse passendem Präfix durchsucht. Ist das passende Routing-Protokoll gefunden, wird eine Nachricht des Typs `FIB_MSG_RP_SIGNAL_UNREACHABLE_DESTINATION` mit angehängter gesuchter Zieladresse an den Routing-Protokoll Thread geschickt. Dieser quittiert die Nachricht, und die FIB gibt `-EHOSTUNREACH` als Rückgabewert an den Aufrufer zurück.

Listing 2.9: Funktion zum Anfragen nach einer Source-Route

```
1  /**
2  * @brief copies a source route to the given destination
3  *
4  * @param[in] table table to search for a source route
5  * @param[in] dst pointer to the destination address bytes
6  * @param[in] dst_size the size in bytes of the destination address type
7  * @param[out] sr_iface_id pointer to the store the iface_id for this route
8  * @param[in, out] sr_flags pointer to store the flags of this route
9  * @param[out] addr_list pointer to the location for storing the source route
       addresses
10 * @param[in, out] addr_list_elements the number of elements available in addr_list
11 * @param[in, out] element_size the provided size for one element in addr_list
12 * @param[in] reverse indicator if the hops should be stored in reverse order
13 * @param[in, out] fib_sr pointer for cosecutive receiving matching source routes.
14 *                 If NULL only the first matching source route is returned.
15 *                 If !NULL the pointer will be overwritten with the current
       returned fib_sr.
16 *                 The FIB skips all entries until the provided fib_sr+1.
17 *                 The fib_sr pointer is only overwritten when a further matching
       sr has been found.
18 *
19 * @note The actual needed size for an element and the number of elements
20 *       is stored in addr_list_elements and element_size respectively
21 *       when the return value is NOT -EFAULT or NOT -EHOSTUNREACH.
```

```
22  *       However,the required size for may change in between calls.
23  *
24  * @return 0 on success, path to destination with equal flags
25  *         1 on success, path to destination with distinct flags
26  *         -EFAULT on one of the provided parameter pointers is NULL
27  *         -EHOSTUNREACH if no sr for the destination exists in the FIB
28  *         -ENOBUFS if the size to store all hops is insufficient low
29  */
30 int fib_sr_get_route(fib_table_t *table, uint8_t *dst, size_t dst_size,
31                      kernel_pid_t *sr_iface_id,
32                      uint32_t *sr_flags,
33                      uint8_t *addr_list, size_t *addr_list_size,
34                      size_t *element_size,
35                      bool reverse, fib_sr_t **fib_sr);
```

3 Implementierung eines angreifenden RPL Knotens

3.1 Problemstellung

Das RPL Protokoll beinhaltet Schwachstellen, die durch einen Knoten innerhalb der Topologie angegriffen werden können. Zur Evaluierung der Widerstandsfähigkeit der RPL-Implementierung gegen Angriffe innerhalb des DODAG, wird ein Knoten benötigt der kontrolliert Angriffe durchführen kann. Damit lassen sich einerseits die Effekte eines Angriffs untersuchen, andererseits Schutzmechanismen wie VeRA[6] oder TRAIL[7] evaluieren.
Im Folgenden werden kurz zwei wesentliche Eigenschaften von RPL beschrieben, der *rank* eines Knotens in der Topologie und die *DODAG-Versionsnummer*, die auch gleichzeitig angreifbare Schwachstellen des RPL Routing-Protokolls sind.

Rank RPL basiert auf einem streng hierarchischen Aufbau eines DODAG. Störung in der Monotonie der hierarchischen Beziehungen zwischen Knoten verursachen eine Restrukturierung dieser Beziehungen auf den betroffenen Knoten. Die Knoten versuchen die monotone Ordnung mit einer lokalen Reparatur wiederherzustellen und lösen dafür ihre Eltern-Kind-Beziehungen auf. Ein betroffener Knoten löscht alle eingetragenen Eltern, sowie die Weiterleitungsinformationen zu ihnen aus der FIB. Letztendlich verlässt er damit den DODAG. Dann setzt der Knoten seinen *trickle-timer* zurück und wartet auf eingehende DIO-Nachrichten, um sich erneut einem DODAG mit der Wahl eines *preferred-parent* anzuschließen. Die Wahl eines *preferred-parent* wird vom *rank* eines Knotens bestimmt. Der *rank* gibt die Distanz eines Knotens zum DODAG-root in der Topologie an. Zum ermitteln dieser Topologischen Distanz können verschiedene Parameter hinzugezogen werden. Im einfachsten Fall wird hierbei der Hopcount verwendet[8]. Ein niedriger *rank* bedeutet dass ein Knoten ein lohnender *preferred-parent* sein könnte, da über ihn der root des DODAGs zuverlässig und potentiell mit wenigen Hops erreichbar ist. Bekommt ein Knoten eine DIO eines Nachbarknotens mit einem niedrigerem *rank* als sein aktuell gewählter *preferred-parent*, ersetzt er ihn mit dem Nachbarknoten. Mit die Wahl eines neuen *preferrred-parents* stellt der Knoten seinen eigenen *rank* ein. Dieser wird, monoton steigend um einen festgelegten Differenzbetrag, größer als der *rank* des *preferred-parent* gesetzt. Dann annonciert er seine neue Position in der Topologie, seinen *rank*, indem er DIOs an die Multicast-Adresse `ff02::1a` versendet. RPL Knoten befinden sich automatisch in dieser Muticast-Gruppe. Alle Knoten die diese DIO empfangen, stellen ihrerseits ihren *preferred-parent* neu ein, falls der annoncierte *rank* attraktiver als der des aktuellen *preferred-parent*s ist, oder die DIO den *rank* des aktuellen *preferred-parent* ändert. Diese Umstrukturierung in der Topologie breitet sich in Richtung der Blätter des DODAG aus. Je kleiner und damit

attraktiver der annoncierte *rank* in der DIO ist, desto mehr Knoten sind potentiell von der Umstrukturierung betroffen.

DODAG-Versionsnummer Ist die streng monotone Ordnung des DODAGs nicht durch eine lokale Reparatur wiederherzustellen, beispielsweise wenn das Auflösen der Eltern-Kind-Knoten Beziehungen eines Zweigs bis hin zum root-Knoten wandert, kann der root-Knoten eine globale Reparatur vornehmen. Hierfür erhöht er die Versionsnummer des DODAGs und annonciert die Versionsänderung indem er DIOs an die Multicast-Adresse `ff02::1a` versendet. Empfängt ein Knoten eine DIO, die eine Versionsänderung des DODAG annonciert, löscht er alle seine eingetragenen Eltern, merkt sich die neue Versionsnummer des DODAG und wählt einen neuen *preferred-parent*. Da alle *parents* im Schritt zuvor gelöscht wurden, wählt er automatisch den Knoten von dem er die DIO erhalten hat als seinen *preferred-parent*, da dieser in dem Moment der einzige Knoten ist der die gültige neue DODAG-Versionsnummer annonciert. Mit der Wahl eines neuen *preferred-parent* und der damit verbundenen Neuberechnung seines eigenen *rank*, versendet er seinerseits DIOs an die Multicast-Adresse `ff02::1a` und annonciert damit die erhöhte Versionsnummer des DODAG und seinen neuen *rank*. Eine Erhöhung der Versionsnummer betrifft immer alle Knoten des DODAG. Jeder Knoten, unabhängig davon ob seine Eltern-Kind-Beziehung streng monoton und konfliktfrei ist, löscht alle seine *parents* und wählt seinen *preferred-parent* neu.

3.2 Anforderungen

- Es soll ein angreifender Knoten implementiert werden, der die zuvor genannten Schwachstellen in RPL für Angriffe ausnutzt.
- Die Implementierung muss ermöglichen den Angriff gezielt steuern zu können. Der angreifende Knoten muss sich RPL Standard-konform verhalten können bis der Angriff aktiviert wird.
- Der Angriff auf den *rank* soll beliebig zur Laufzeit aktivierbar und de-aktivierbar sein.
- Der für einen Angriff annoncierte *rank* soll zur Laufzeit verändert werden können.
- Es soll ermöglicht werden für einen Angriff auf die DODAG-Versionsnummer, diese während der Laufzeit zu erhöhen. Dabei soll es möglich sein einen bestimmten DODAG angreifen zu können in dem der angreifende Knoten teilnimmt.

3.3 Konzept

In diesem Abschnitt werden Angriffe auf die zuvor genannten Schwachstellen von RPL beschrieben, welche die Grundlage der Implementierung eines Angreifenden Knotens darstellen. Es wird zudem kurz diskutiert, welche Auswirkung der jeweilige Angriff in der Topologie hat.

Angriff gegen den Rank Ein angreifender Knoten kann unabhängig seiner tatsächlichen Position im DODAG einen beliebig kleinen *rank* wählen. Annonciert er diesen sehr attraktiven fingierten *rank* in DIOs, werden ihn alle Nachbarknoten als *preferred-parent* wählen wenn sie keinen *preferred-parent* gleicher oder besserer Güte haben. Erfolgreich angeschwindelte Knoten werden daraufhin ihren eigenen *rank* verbessern und ihn ihrerseits an die Nachbarknoten annoncieren. Die Manipulation pflanzt sich bis zu den Blattknoten im DODAG fort.
Nun führen alle Pfade des DODAG die durch den Angriff betroffen sind zum angreifenden Knoten. Ausgehend von dieser Situation hat der Angreifer die Möglichkeit mit verschiedenen Angriffen fortzufahren. Da potentiell ein Großteil des Kommunikationsflusses über ihn geleitet wird, kann er Angriffe auf den Nachrichteninhalt durchführen. Er kann die über ihn geleiteten Nachrichten selektiv weiterleiten, versuchen den Inhalt mitzulesen oder zu manipulieren. Er kann den Nachrichtenfluss zum DODAG-root unterbrechen, indem er alle durch ihn geleitete Nachrichten die zum root weitergeleitet werden sollen verwirft, und damit ein *sink-hole*[9] erzeugen.
Soll der Angriff die Einsatzfähigkeit der Knoten in der Topologie stören, kann der Angreifer folgendermaßen vorgehen. Im nächsten Schritt verschlechtert der Angreifer seinen *rank* und annonciert diesen unattraktiven *rank* in DIOs. Seine Nachbarn, die ihn als *preferred-parent* gewählt haben aktualisieren den neuen *rank* des Angreifers. Kommt ein anderer Knoten aus der Liste der *parents* als neuer *preferred-parent* in Frage tauscht der Knoten den *preferred-parent* aus. Um danach die Monotonie des eigenen *rank* zu bewahren, passt der Knoten den eigenen *rank* entsprechend an. Bedingt durch die Änderung des eigenen *rank*, wird der *trickle-timer* des Knotens zurückgesetzt und DIOs an die Nachbarknoten mit dem neu eingestellten *rank* versendet. Das pflanzt sich in allen Zweigen des DODAG fort die von der initialen Manipulation des *rank* durch den Angreifer betroffen sind. Von dieser Situation ausgehend kann der Angreifer nach einer gewissen Wartezeit, in der die *rank*-Änderungen im DODAG konvergieren, seinen *rank* erneut wesentlich verbessern und wieder alle Knoten mit der Restrukturierung der Eltern-Kind-Beziehungen beschäftigen. Durch periodisches oszillieren des *rank* bewirkt so der Angreifer, dass (i) das Übertragungsmedium häufig mit RPL-Kontrollnachrichten belegt ist und die Weiterleitung von Nutzinformationen erheblich stört. Und (ii) der Energieverbrauch der betroffenen Knoten wesentlich erhöht wird, da durch das Zurücksetzen des *trickle-timer*s innerhalb relativ kurzer Intervalle immer wieder viele DIO-Nachrichten versendet werden müssen. Damit sind Ruhephasen, in denen die Knoten Energie sparen können deutlich verkürzt.

Angriff gegen die DODAG-Versionsnummer Ein angreifender Knoten kann unabhängig vom DODAG-root die DODAG-Versionsnummer unerlaubt erhöhen. Versendet er die höhere Versionsnummer in einer DIO an seine Nachbarn, werden diese jeweils alle ihre *parents* löschen und zunächst den Versender der DIO als *preferred-parent* wählen. Die Knoten werden ihrerseits die neue Version in DIOs annoncieren und bewirken, dass die Eltern-Kind-Beziehungen ihrer Nachbarknoten neu eingestellt werden. Dies pflanzt sich in alle Richtungen im DODAG fort, sowohl in Richtung der Blätter als auch in Richtung des DODAG-root. Durch das unerlaubte erhöhen der DODAG-Version ist der Angreifer der erste Knoten in der neuen DODAG-Version. Dadurch bleibt er solange der *preferred-parent* für seine Nachbarknoten, bis diese eine DIO

von einem anderen Knoten empfangen der einen besseren *rank* annonciert. Tritt dieser Fall ein, werden die *rank*s im betroffenen Zweig im DODAG erneut eingestellt. Durch periodisches unerlaubtes erhöhen der DODAG-Version und gleichzeitigem versenden von DIO-Nachrichten mit der erhöhten DODAG-Version beeinflusst so der Angreifer alle Knoten im DODAG. Wie im Angriff auf den *rank* wird dadurch das Übertragungsmedium oft mit RPL-Kontrollnachrichten belegt, was die Weiterleitung von Nutzdaten erheblich stört. Ähnlich wie im Angriff auf den *rank* wird der Energieverbrauch auf den Knoten deutlich erhöht, da die Ruhephasen wesentlich verkürzt werden oder ganz ausbleiben. Anders als beim Angriff auf den *rank*, sind hier immer alle Knoten des DODAG vom Angriff betroffen.

3.4 Umsetzung

Der RPL Angreifer ist direkt in die RPL-Implementierung integriert[1]. Für die Manipulation des *rank* ist es erforderlich, dass der Angreifer seinen *rank* im DODAG struct[2] überschreibt. Dieser wird in der RPL-Implementierung gesetzt, wenn ein Knoten einen neuen *preferred-parent* wählt. Ein ehrlicher Knoten errechnet sich mit der *objective-function* den eigenen *rank* anhand des *rank* vom gewähltem *preferred-parent*. Die Implementierung ist für den Angriff dahingehend erweitert worden, dass zunächst ein zuvor festgelegter beliebiger *rank* in den DODAG struct eingetragen wird, anstatt des errechneten *ranks*[3], vgl. Listing 3.1. Ein beliebig niedrig gewählter *rank* bewirkt nun, dass der angreifende Knoten potentiell keinen Knoten mehr kennt den er als RPL parent nutzen kann, da kein Nachbarknoten einen niedrigeren *rank* hat. Als Folge werden bedingt durch die Implementierung von RPL alle *parents* entfernt und die next-hops zu ihnen aus der FIB entfernt. Hat ein Knoten keine *parents*, verlässt er korrekterweise den DODAG. Um einen Angriff auf den *rank* zu ermöglichen verhindert die Implementierung des Angreifers, dass er den DODAG verlässt. Dabei kann aus zwei Optionen gewählt werden.

(i) Der Angreifer behält einen *preferred-parent* obwohl die Monotonie im *rank* gestört ist[4]. Bekommt der Angreifer eine DIO von einem Nachbarn, errechnet er wie ein ehrlicher Knoten seinen *rank*. Diesen merkt sich der Angreifer und aktualisiert anhand des ehrlichen *rank* seine *parents*. Dadurch werden next-hops zu den ihnen in die FIB eingetragen und aktualisiert. Durch geschickte Wahl eines fingierten *rank*s kann so ein Angreifer einen Großteil der Kommunikation auf sich lenken und trotzdem einen Kommunikationspfad zum DODAG-root halten. Dies ist jedoch nur möglich, wenn mindestens ein Nachbarknoten des Angreifers einen anderen ehrlichen *preferred-parent* hat und der Angreifer den Knoten als seinen *preferred-parent* wählt. Unter Verwendung dieser Option können leicht *routing-loops* entstehen, die dazu führen dass die vom Angriff betroffenen

[1] https://github.com/RIOT-OS/RIOT/pull/4831
[2] https://github.com/RIOT-OS/RIOT/blob/master/sys/include/net/gnrc/rpl/structs.h#L228
[3] https://github.com/BytesGalore/RIOT/blob/add_rpl_attacker/sys/net/gnrc/routing/rpl/gnrc_rpl_dodag.c#L266
[4] https://github.com/BytesGalore/RIOT/blob/add_rpl_attacker/sys/net/gnrc/routing/rpl/gnrc_rpl_dodag.c#L309

Knoten eine lokale Reparatur durchführen. Dies passiert im einfachsten Fall, wenn der Angreifer einen Kind-Knoten als seinen eigenen *preferred-parent* wählt.

(ii) Der Angreifer verlässt nicht den DODAG wenn er keine *parents* mehr hat[5]. Dabei wird in den Mechanismus der lokalen Reparatur eingegriffen und das automatische entfernen der Zugehörigkeit zum DODAG verhindert. Diese Option verhindert, dass der Angriff *routing-loops* erzeugen kann und der Angreifer dem Verhalten nach der root des erfolgreich angegriffenen sub-DODAGs ist.

Zum Manipulieren der DODAG-Versionsnummer kann der Angreifer zu einem beliebigen Zeitpunkt seine lokal gespeicherte Versionsnummer des DODAG structs unberechtigt um den definierten Wert im DODAG erhöhen. Der Angreifer kann sich sonst RPL konform verhalten. Läuft sein *trickle-timer* zum annoncieren von DIOs ab versendet er die erhöhte Versionsnummer an seine Nachbarknoten.

Listing 3.1: Manipulation des *rank*

```
/**
 * @brief   Find the parent with the lowest rank and update the DODAG's preferred
     parent
 *
 * @param[in] dodag     Pointer to the DODAG
 *
 * @return  Pointer to the preferred parent, on success.
 * @return  NULL, otherwise.
 */
static gnrc_rpl_parent_t *_gnrc_rpl_find_preferred_parent(gnrc_rpl_dodag_t *dodag)
...
#ifdef RPL_ATTACKER
    if (atk_gnrc_rpl_opts.rank_atk_activated) {
        dodag->my_rank = atk_gnrc_rpl_opts.rank_override;
    }
    else {
        dodag->my_rank = dodag->instance->of->calc_rank(dodag->parents, 0);
    }
#else
    dodag->my_rank = dodag->instance->of->calc_rank(dodag->parents, 0);
#endif
...
```

Steuerung des Angreifers

Die Implementierung des angreifenden Knotens erlaubt seine Konfiguration und Steuerung zur Laufzeit. Zusätzlich zu den neu eingeführten Funktionen wurden *shell commands*[6] implementiert die einem Benutzer erlauben den Angriff auf den *rank* sowie die DODAG-Versionsnummer

[5] https://github.com/BytesGalore/RIOT/blob/add_rpl_attacker/sys/net/gnrc/routing/rpl/gnrc_rpl_dodag.c#L216
[6] https://github.com/RIOT-OS/RIOT/tree/master/sys/shell/commands

zu steuern[7]. Die Kommandos für den Angreifenden Knoten bestehen aus den Schlüsselwörtern '`rpl atk`' gefolgt von zusätzlichen Optionen:

- `rpl atk`: Zeigt die eingestellten Optionen[8] und den aktuellen Zustand des Angriffs an.
- `rpl atk init`: Initialisiert den angreifenden Knoten mit voreingestellten Optionen in denen der Angriff deaktiviert ist.
- `rpl atk rank <value>`: Stellt den gewünschten nummerischen *rank* aus `<value>` für den Knoten ein. Der eingestellte *rank* wird nur verwendet, wenn der Angriff durch den Knoten eingeschaltet wurde.
- `rpl atk rank [start|stop]`: Aktiviert oder deaktiviert den Angriff auf den *rank*.
- `rpl atk dodag [keep|loose]`: Stellt ein ob der angreifende Knoten nach einem durchgeführtem *local-repair* den DODAG regelwidrig beibehält oder regelkonform verlässt.
- `rpl atk parents [keep|loose]`: Stellt ein ob der angreifende Knoten seinen *preferred-parent* behält, auch wenn die monotone Ordnung des *rank* durch den Angriff verletzt ist.
- `rpl atk version [<id>|all]`: Erhöht die DODAG-Versionsnummer des Knoten für den zur DODAGID (`<id>`) passenden DODAG, oder die DODAG-Versionsnummer jedes DODAG in dem der Knoten derzeit teilnimmt.

[7] https://github.com/BytesGalore/RIOT/blob/add_rpl_attacker/sys/shell/commands/sc_gnrc_rpl.c#L309
[8] https://github.com/BytesGalore/RIOT/blob/add_rpl_attacker/sys/include/net/gnrc/rpl.h#L532

4 Portierung von TRAIL in die aktuelle RPL-Implementierung von RIOT

Ende 2013 wurde im Rahmen einer Masterarbeit[1] eine sehr frühe proof-of-concept TRAIL Demonstration für RIOT implementiert[2]. Sie demonstriert die *single-path-validation* Variante von TRAIL[10] auf msba2-Knoten[3]. In dieser Variante wird jeder Zweig von einem Blattknoten des DODAGs bis hin zum DODAG-root separat validiert. Validierungen verschiedener Knoten, die einen gemeinsamen Vorgänger-Knoten auf dem Pfad zum root-Knoten haben, können in dieser Variante nicht zusammengefasst werden.

4.1 Problemstellung

Die in der Demo eingesetzten msba2-Knoten haben einen CC1100 transceiver[4], der eine proprietäre drahtlose Übertragung im sub-GHz Bereich ermöglicht. Dadurch ist die Demo nicht auf anderen Knoten lauffähig.
Zum Zeitpunkt der Demonstration war der Zustand des Netzwerkstack von RIOT in einer sehr frühen und inzwischen komplett überholten Version. Die Paketübertragung und das Routing durch RPL war fest ineinander verschachtelt und der Netzwerkstack war nicht klar in Schichten getrennt. Mit der stetigen Entwicklung von RIOT wurde der Netzwerkstack von Grund auf neu implementiert und ersetzt. In der aktuellen Release-Version von RIOT (2015.12[5]) sind die Schichten des Netzwerkstacks klar voneinander abgegrenzt. Viele der durch RIOT unterstützen Boards benutzen einen IEEE802.15.4[11] konformen Transceiver für die drahtlose Kommunikation.
Die TRAIL Demo kann damit in ihrer ursprünglichen Fassung nicht im aktuellen RIOT eingesetzt werden. Die Neuerungen und Verbesserungen des RIOT Netzwerkstacks sowie der RPL-Implementierung können nicht in der ursprünglichen Demo evaluiert werden.

4.2 Anforderungen

- Die ursprüngliche Demo soll in das aktuelle RIOT überführt werden.

[1] http://www.inet.haw-hamburg.de/thesis/completed/Heiner_Perrey-MA.pdf/view
[2] https://github.com/hper/RIOT/tree/rpl_trail
[3] https://github.com/RIOT-OS/RIOT/wiki/Board:-MSBA2
[4] http://www.ti.com/lit/ds/symlink/cc1100.pdf
[5] https://github.com/RIOT-OS/RIOT/tree/2015.12-branch

- Dabei soll der aktuelle Netzwerkstack sowie die aktuelle RPL-Implementierung eingesetzt werden.
- Die Demo soll nach der Portierung ermöglichen TRAIL auf aktuell durch RIOT unterstützten Knoten mit IEEE 802.15.4 Transceivern auszuführen.
- Zur Demonstration verschiedener Szenarien sollen die Knoten auf denen TRAIL ausgeführt wird über Eingaben in der Konsole steuerbar sein.

4.3 Konzept der ursprünglichen Demo-Implementierung

Die TRAIL *single-path-validation* wird gestartet wenn ein Knoten eine DIO von einem Nachbarknoten empfängt und der annoncierte *rank* in der DIO den Nachbarn zu einem potentiellen *parent* macht. Als erstes überprüft die TRAIL Implementierung, ob der Versender der DIO mit dem annoncierten *rank* bereits erfolgreich validiert wurde. Hierfür wird eine zirkuläre Liste durchsucht, in der bereits durch TRAIL validierte *parents* gespeichert werden. Jeder Eintrag der Liste beinhaltet die IPv6-Adresse und den durch TRAIL validierten *rank* eines Knotens. Ist der potentielle *parent* nicht in der Liste und damit noch nicht validiert, speichert TRAIL alle Informationen aus der DIO des Nachbarn in einer Liste ab, dem TRAIL parent buffer. Daraufhin bereitet TRAIL eine *TRAIL validation object*(TVO)-Nachricht[6] vor. Eine TVO-Nachricht beinhaltet:

- einen nounce den der initiierende Knoten zur Wiedererkennung der TVO einträgt
- die eigene IPv6-Adresse, die den Initiator der TRAIL-Validierung identifiziert
- den *rank* des Knotens der validiert werden soll
- eine fortlaufende Sequenznummer

Bis auf den *rank* des potentiellen *parent*-Knotens werden alle Felder ausgefüllt. Die TVO wird dann an den potentiellen *parent*-Knoten gesendet. Beim Empfang einer TVO prüft ein Knoten, ob er selbst zum Validieren seines in der DIO annoncierten *rank*s aufgefordert ist. Dafür überprüft er, ob im *rank* Feld der TVO bereits ein gültiger *rank* eingetragen wurde. Ist ein gültiger *rank* eingetragenen, muss er größer als der *rank* des Empfängers der TVO sein, oder die TVO wird aufgrund der Verletzung der *rank*-Ordnung verworfen. Wurde noch kein gültiger *rank* in die TVO eingetragen, trägt der Knoten seinen *rank* in die TVO ein. Dann speichert der Knoten die TVO temporär in einer Liste ab und merkt sich zusätzlich die link-lokale IPv6 Adresse des Knotens von dem er die TVO erhalten hat. Der Empfang der TVO wird bestätigt, indem eine TVO-ACK[7] Nachricht mit der Sequenznummer aus der empfangenen TVO zurück an den Sender geschickt wird. Die TVO wird dann an den *preferred-parent* weitergeleitet. Dieser

[6] https://github.com/hper/RIOT/blob/rpl_trail/sys/net/rpl/rpl_structs.h#L143
[7] https://github.com/hper/RIOT/blob/rpl_trail/sys/net/rpl/rpl_structs.h#L161

geht genau so vor wie der vorherige Knoten und leitet seinerseits die TVO an seinen eigenen *preferred-parent* weiter. Letztendlich wandert die TVO bis zum DODAG-root. Der root-Knoten überprüft zuerst ob der eingetragene *rank* in der TVO größer als der eigene *rank* ist. Wurde die *rank*-Ordnung nicht verletzt, signiert der root die TVO. Dabei ist zu beachten, dass RIOT zum Zeitpunkt der Implementierung der Demo sowie zum jetzigen Zeitpunkt kein Asymmetrisches Signaturverfahren beherrscht. Unter der Annahme, dass digitale Signaturen hinreichend sicher und auf Sensorknoten einsetzbar sind, wird lediglich ein Platzhalter für die Signatur in die TVO eingetragen. Die signierte TVO wird vom root zurück an den Knoten geschickt, von dem er die unsignierte TVO erhalten hat. Empfängt ein Knoten eine signierte TVO, prüft er dir Gültigkeit der Signatur. Auch hier wird nur der Platzhalter der Signatur gelesen und als valide erachtet. Daraufhin vergleicht er, ob der richtige nounce in der TVO steht und überprüft, ob der eingetragene *rank* in der signierten TVO noch immer monoton kleiner ist als sein eigener. Ist die Signatur oder die *rank*-monotonie manipuliert verwirft der Knoten die TVO und löscht den entsprechenden Eintrag des potentiellen *parents* aus dem TRAIL parent buffer. Wenn die TVO gültig ist, überprüft der Knoten, ob seine link-lokale IPv6 Adresse in der TVO eingetragen ist und es sich damit um seine eigene TVO handelt. Handelt es sich nicht um die eigene TVO, durchsucht der Knoten die Liste der zwischengespeicherten TVOs nach der Sequenznummer der signierten TVO, und schickt die signierte TVO an die gespeicherte link-lokale IPv6 Adresse. Ist es die eigene TVO, ist damit der getestete potentielle parent mit seinem annoncierten *rank* validiert. Mit den zwischengespeicherten Informationen aus dem TRAIL parent buffer wird dann der *parent* in RPL eingetragen und verwendet. Der entsprechende Eintrag im TRAIL parent buffer wird zum Überschreiben freigegeben und die zwischengespeicherte TVO durch die signierte TVO ersetzt. Bekommt ein Knoten über einen gewissen Zeitraum keine signierte TVO als Antwort, wiederholt er die Pfadvalidierung für den potentiellen *parent*. Nach einer fest eingestellten Anzahl an erfolglosen Versuchen einer Pfadvalidierung, entfernt der Knoten den temporär gespeicherten potentiellen *parent* aus dem TRAIL parent buffer.
Für die Demonstration eines Angriffs auf den *rank*, können die Knoten per *shell command* einen festgelegten niedrigen *rank* in DIOs annoncieren. Auch TRAIL kann über einen *shell command* ein und wieder ausgeschaltet werden.

Visualisierung Für die Visualisierung der Demo wird RIOT-TV[8] verwendet. Dieses Visualisierungswerkzeug wurde für den Einsatz zum Darstellen von Topologien auf dem DES-Testbett[9] entwickelt, kann aber auch auf beliebigen Rechnern lokal ausgeführt werden. RIOT-TV ist ein client-server basiertes Werkzeug. Die Clients (*reporter*[10]) stellen jeweils eine Verbindung zu einem der RIOT-Knoten in der Topologie her. Jeder *reporter* verbindet sich über den seriellen Port mit einem RIOT Knoten. Über den seriellen Port liest der *reporter* die Ausgaben des Knotens und sendet diese an die Serverapplikation (*anchor*[11]) von RIOT-TV.

[8] https://github.com/RIOT-OS/Riot-TV
[9] http://www.des-testbed.net/
[10] https://github.com/RIOT-OS/Riot-TV/blob/master/reporter/reporter.js
[11] https://github.com/RIOT-OS/Riot-TV/blob/master/anchor/anchor.js

Über die *Socket.IO*[12] verbindet sich ein *reporter* mit dem *anchor*. Die mitgelesenen Ausgaben des Knotens werden über *JavaScript Object Notation* (JSON)-Objekte[12] an den *anchor* übermittelt[13]. Der *anchor* durchsucht die eintreffenden JSON-Objekte der *reporter* nach festgelegten Schlüsselwörtern[14] um Aktionen in der Visualisierung anzuzeigen.
Für die Anzeige verwendet RIOT-TV *sigmajs*[15]. Die Knoten die in der Topologie durch RIOT-TV angezeigt werden sollen, müssen vorab in RIOT-TV als JSON-Objekte festgelegt werden[16]. Für jeden dargestellten Knoten wird eine eindeutige ID für die Anzeige festgelegt. Die ID wird durch RIOT-TV benutzt um Aktionen wie bspw. den Empfang einer DIO-Nachricht, oder das versenden einer TVO-Nachricht für den Knoten zu visualisieren.

4.4 Umsetzung der Portierung

Die Portierung von TRAIL in das aktuelle RIOT orientiert sich sowohl an der ursprünglichen sehr frühen proof-of-concept Demo, als auch an dem dahinter liegendem Konzept der Demo[17].
Die Portierte TRAIL-Implementierung greift direkt in RPL an zwei Stellen ein:

(i) Beim Empfang und der Verarbeitung von eingehenden RPL-Kontrollnachrichten. Die RPL-Implementierung ist als Empfänger solcher Nachrichten im aktuellen Netzwerkstack von RIOT eingetragen. Dadurch wird per *message passing* RPL darüber informiert, wenn RPL-Kontrollnachrichten, wie bspw. DIO-Nachrichten, durch den Knoten empfangen werden. Zum Verarbeiten der Nachrichten implementiert RPL eine *eventloop* die auf Kontrollnachrichten aus dem Netzwerkstack wartet und sie zur Weiterverarbeitung an andere Funktionen verteilt[18].
TVO und TVO-ACK Nachrichten werden in der Portierung als RPL-Kontrollnachrichten verwendet. Beim Empfangen einer TVO oder TVO-ACK werden sie wie die ursprünglichen RPL-Kontrollnachrichten im Netzwerkstack per *message passing* an die *eventloop*[19] von RPL weitergeleitet. Die eventloop verteilt beim Empfang die neuen TRAIL Kontrollnachrichten an separate Funktionen zur Weiterverarbeitung, vgl. Listing 4.1.

Listing 4.1: Verteilung der TRAIL Kontrollnachrichten

```
1 static void _receive(gnrc_pktsnip_t *icmpv6)
2 {
3 ...
```

[12] http://socket.io
[13] https://github.com/RIOT-OS/Riot-TV/blob/master/reporter/reporter.js#L122
[14] https://github.com/RIOT-OS/Riot-TV/blob/master/anchor/anchor.js#L292
[15] http://sigmajs.org/
[16] https://github.com/RIOT-OS/Riot-TV/blob/master/anchor/data/trail.json
[17] https://github.com/BytesGalore/RIOT/tree/TRAIL_SAFEST_DEMO_Legacy_migration
[18] https://github.com/BytesGalore/RIOT/blob/TRAIL_SAFEST_DEMO_Legacy_migration/sys/net/gnrc/routing/rpl/gnrc_rpl.c#L1125
[19] https://github.com/BytesGalore/RIOT/blob/TRAIL_SAFEST_DEMO_Legacy_migration/sys/net/gnrc/routing/rpl/gnrc_rpl.c#1088

```
4       icmpv6_hdr = (icmpv6_hdr_t *)icmpv6->data;
5       switch (icmpv6_hdr->code) {
6   ...
7       case (ICMP_CODE_TVO): {
8            recv_rpl_tvo((struct rpl_tvo_t *)(icmpv6_hdr + 1),
9            &ipv6_hdr->src);
10           break;
11           }
12
13      case (ICMP_CODE_TVO_ACK): {
14          recv_rpl_tvo_ack((struct rpl_tvo_ack_t *)(icmpv6_hdr + 1),
15          &ipv6_hdr->src);
16          break;
17          }
18  ...
19      }
20
21      gnrc_pktbuf_release(icmpv6);
22  }
```

Die Verarbeitung eingehender TVO[20] sowie TVO-ACK[21] Kontrollnachrichten implementiert das zuvor im Abschnitt 4.3 beschriebene Verhalten.
Empfängt der Knoten eine TVO mit valider Signatur[22], stellt dies einen erfolgreich abgeschlossenen TRAIL Validierungsprozess für den potentiellen *parent* dar. Daraufhin wird der durch TRAIL zwischengespeicherte potentielle *parent* in RPL eingetragen[23]. Dabei wird der temporär gespeicherte *parent* gelöscht.

(ii) Bei der Verarbeitung von eingegangenen DIO-Kontrollnachrichten[24]. Wird eine DIO empfangen in der ein Knoten einen lohnenden *rank* annonciert, und damit als *parent* oder sogar *preferred-parent* in Frage kommt, greift die TRAIL Implementierung ein. Sie verhindert dass RPL den *parent* ohne die erfolgreich durchgeführte TRAIL Validierung nutzt[25]. Für den Prozess der Validierung werden alle Daten aus der DIO, die bei der Wahl eines *parents* benutzt werden, temporär durch TRAIL in einem buffer gespeichert[26]. Die TRAIL Implementierung erstellt dann eine TVO und sendet diese zum Versender der

[20] https://github.com/BytesGalore/RIOT/blob/TRAIL_SAFEST_DEMO_Legacy_migration/sys/net/gnrc/routing/rpl/gnrc_rpl.c#L606
[21] https://github.com/BytesGalore/RIOT/blob/TRAIL_SAFEST_DEMO_Legacy_migration/sys/net/gnrc/routing/rpl/gnrc_rpl.c#L907
[22] https://github.com/BytesGalore/RIOT/blob/TRAIL_SAFEST_DEMO_Legacy_migration/sys/net/gnrc/routing/rpl/gnrc_rpl.c#L679
[23] https://github.com/BytesGalore/RIOT/blob/TRAIL_SAFEST_DEMO_Legacy_migration/sys/net/gnrc/routing/rpl/gnrc_rpl.c#L745
[24] https://github.com/BytesGalore/RIOT/blob/TRAIL_SAFEST_DEMO_Legacy_migration/sys/net/gnrc/routing/rpl/gnrc_rpl_control_messages.c#L499
[25] https://github.com/BytesGalore/RIOT/blob/TRAIL_SAFEST_DEMO_Legacy_migration/sys/net/gnrc/routing/rpl/gnrc_rpl_control_messages.c#L603
[26] https://github.com/BytesGalore/RIOT/blob/TRAIL_SAFEST_DEMO_Legacy_migration/sys/net/gnrc/routing/rpl/gnrc_rpl.c#L60

DIO[27].
Gleichzeitig erstellt TRAIL einen *thread*, der in vier festgelegten Zyklen den TRAIL Validierungsprozess für den potentiellen parent nach einem *timeout* wiederholt[28]. Ist der letzte Zyklus durchlaufen, löscht TRAIL den zwischengespeicherten potentiellen *parent*. Ist die Validierung erfolgreich beendet, oder die festgelegte Anzahl an Zyklen durchlaufen, wird dieser *thread* beendet.

Bei der Verarbeitung von RPL-Kontrollnachrichten, erzeugt die Portierung Ausgaben für RIOT-TV, bspw. zur Visualisierung einer empfangenen TVO[29].
Solche Ausgaben wurden sowohl beim Verarbeiten von eingehenden sowie für ausgehende RPL-Kontrollnachrichten für RIOT-TV hinzugefügt. Dadurch ist es möglich, die gesamte RPL und TRAIL spezifische Kommunikation zwischen RPL-Knoten in RIOT-TV darzustellen.
Zum steuern der Demonstration wurden neue *shell commands* hinzugefügt:

- `trail`: aktiviert die TRAIL-Validierung auf dem aktuellem Knoten
- `attack <rank>`: startet einen angreifenden Knoten, der einen übergebenen nummerischen `<rank>` vorgibt.
  Ein negativer Wert für den *rank* schaltet den Angriff auf dem Knoten ab und der Knoten berechnet seinen tatsächlichen *rank* den er propagiert.
  Damit ist es möglich zur Laufzeit jeden Knoten als Angreifer einzusetzen und verschiedene Szenarien mit einem oder mehreren Angreifern zu demonstrieren.
- `tabula_rasa`: veranlasst, dass die Lebensdauer jedes *parents* sofort abläuft und RPL alle *parents* des Knotens löscht.
- `drain_lifetime <addr>`: lässt die Lebensdauer des *parent* mit der IPv6-Adresse `<addr>` auf 1/4 schrumpfen.
  Dadurch wird ermöglicht zur Laufzeit verschiedene Konstellation der Knoten in kurzen Intervallen zu demonstrieren.
- `sze1`: startet ein zur Compilezeit festgelegtes Demo-Szenario auf dem Knoten.
  Das generelle Demo-Szenario besteht aus einem DODAG mit 5 Knoten. Zwei der Knoten sind nicht direkt mit dem root des DODAG verbunden und müssen einen 1-hop Nachbarn des root Knotens als *parent* nutzen.
  Anhand seiner link-lokalen IPv6-Adresse bestimmt der Knoten automatisch seine zur Compilezeit festgelegte Position in der Topologie und erzeugt blacklist-Einträge für zuvor festgelegte Nachbarknoten.
  Ausgeführt auf 5 Knoten kann dieser definierte Ausgangszustand für das Demo-Szenario schnell hergestellt werden.

[27] https://github.com/BytesGalore/RIOT/blob/TRAIL_SAFEST_DEMO_Legacy_migration/sys/net/gnrc/routing/rpl/gnrc_rpl.c#L527
[28] https://github.com/BytesGalore/RIOT/blob/TRAIL_SAFEST_DEMO_Legacy_migration/sys/net/gnrc/routing/rpl/gnrc_rpl.c#L990
[29] https://github.com/BytesGalore/RIOT/blob/TRAIL_SAFEST_DEMO_Legacy_migration/sys/net/gnrc/routing/rpl/gnrc_rpl.c#L790

Durch die Portierung von TRAIL, konnte auf der EWSN'16 die Demo mit dem Titel: *Demonstrating Topological Robustness of RPL with TRAIL*[13] mit der zu der Zeit aktuellsten RIOT Version auf weit verbreiteten samr21-xpro[30] Sensorknoten, die einen IEEE802.15.4 transceiver nutzen, erfolgreich vorgeführt werden.

[30] https://github.com/RIOT-OS/RIOT/wiki/Board:-SAMR21-xpro

5 RPL Security Interfaces

5.1 Problemstellung

DIO-Nachrichten können durch einen Angreifer manipuliert und so für verschiedene Angriffe genutzt werden, vgl. Abschnitt 3. Verfahren zum Absichern der monotonen Ordnung des *rank* der Knoten im DODAG und der Validierung der DODAG-Versionsnummer, wie TRAIL oder VeRA, können RPL gegen manipulierte DIO-Nachrichten schützen. Die Verfahren müssen dafür in die RPL-Implementierung zur Verarbeitung von DIO-Nachrichten eingreifen. Die genannten Verfahren TRAIL und VeRA sind optional und nicht im RPL Standard definiert. Die direkte Integration eines Verfahrens in die RPL-Implementierung, wie bspw. im Abschnitt 4 beschrieben, macht einen Knoten auf dem das Verfahren zum Einsatz kommt inkompatibel zu anderen RPL Knoten. Ein individuell zu pflegender Code-Zweig der RPL-Implementierung von RIOT mit jeweils intrigierten Verfahren ist hierbei keine Option, da dies zu einem erheblichen Wartungs- und kontinuierlichen Integrationsaufwand führen würde.
Trotz der unterschiedlichen Ansätze von TRAIL und VeRA, schützen beide Verfahren den Teil des RPL Routing-Protokolls in denen ein Knoten die Wahl eines *parent* und Reparaturmechanismen bei einer Erhöhung der DODAG-Versionsnummer durchführt.

5.2 Anforderungen

- Es sollen Schnittstellen definiert und implementiert werden, die den für die genannten Angriffe anfälligen Teil der DIO-Verarbeitung in RPL kapseln. Diese Schnittstellen sollen von Verfahren wie TRAIL oder VeRA genutzt werden um ihre Schutzmaßnahmen gegen Manipulation des *rank* sowie der DODAG-Versionsnummer durchzuführen. Dabei soll gewährleistet werden, dass die neuen Schnittstellen die Implementierung von RPL und einem Verfahren klar voneinander abgrenzen. Dies soll ermöglichen die Verfahren beliebig austauschbar zu machen und vereinfachen neue Verfahren zum Schutz während der Laufzeit einzubinden.

- Es soll möglich sein, verschiedene Verfahren gleichzeitig einzusetzen, bspw. kann jeder DODAG, in dem der Knoten teilnimmt, ein anderes Verfahren zum Schutz verwenden.

- Es muss gewährleistet sein, dass ein Knoten auch kein Verfahren zum Schutz für einen DODAG einsetzen kann. So soll der Knoten in ungeschützten DODAGs teilnehmen können.

- Kommt kein Verfahren zum Einsatz, soll die RPL-Implementierung standardkornform und ohne Einschränkungen funktionieren. Dabei soll der Mehraufwand zur Laufzeit durch die neuen Schnittstellen minimal sein.

- Ein Verfahren muss in der Lage sein, einen *parent* zeitversetzt in RPL einzutragen.
  Im Fall von VeRA wird die Validierung eines potentiellen *parents*, mit dem eingetragenen *rank* des Versenders aus der DIO-Nachricht, direkt auf dem Knoten durchgeführt. Beim Einsatz der reparierten und erweiterten Version VeRA++, die zusätzlich gegen widereinspielen von zuvor empfangenen Informationen durch den Angreifer schützt, müssen weitere Knoten in die Validierung einbezogen werden. Bei TRAIL werden immer alle Knoten auf einem direkten Pfad bis zum root des DODAG in den Prozess der Validierung einbezogen.
  Die zusätzliche Kommunikation für die Validierung benötigt Zeit bis zum Abschluss des Prozesses. In dieser Zeitspanne können weitere DIOs vom gleichen potentiellen *parent* oder auch von anderen Knoten eintreffen, die ihrerseits validiert werden müssen.

5.3 Konzept

Es wird ein neues Modul in RIOT eingeführt, dass die Verarbeitung von DIO-Nachrichten der RPL-Implementierung von RIOT kapselt. Die Kapselung durch das neue Modul ist optional und wird zur Compilezeit festgelegt. Kommt das Modul zum Einsatz, wird der zu schützende Bereich der Verarbeitung von DIO-Nachrichten erweitert, sodass die DIO zuerst an das neue Modul zur Verarbeitung weitergereicht wird. Zusätzlich wird der aktuelle *rank* des Knotens, die IPv6-Adresse des Versenders der DIO sowie die aktuelle DODAG-Versionsnummer an das neue Modul weitergereicht. Mit diesen Informationen kann ein konkretes Verfahren seine Maßnahmen zum Schutz des *rank* sowie der DODAG-Versionsnummer durchführen.
Die eingesetzten Verfahren registrieren sich zuvor bei dem Modul um aufgerufen zu werden wenn DIO-Nachrichten eintreffen. Zum Registrieren stellt das neue Modul eine Funktion bereit, die als Parameter einen Präfix sowie einen Funktionspointer zu einer Callback-Funktion des konkreten Verfahrens hat. Der übergebene Präfix legt fest, für welche DODAGIDs[1] das konkrete Verfahren zuständig ist. Das Tupel aus Präfix und Funktionspointer wird in einer Liste des Moduls verwaltet. Sind DODAGIDs für die ein Verfahren zuständig sein soll nicht aggregierbar, kann das Verfahren weitere Tupel registrieren aus Präfix und Callback-Funktion registrieren. Der Prototyp der Callback-Funktion ist im Modul definiert und muss von einem Verfahren implementiert werden. Die Callback-Funktion wird durch das neue Modul aufgerufen wenn das registrierte Verfahren für eine eingetroffene DIO zuständig ist. Kehrt die aufgerufene Callback-Funktion zurück, gibt der Rückgabewert an, ob die Validierung der DIO durch das Verfahren erfolgreich war.
Die Verarbeitung in RPL wartet bis das Ergebnis der Validierung vorliegt. Die gekapselte Verarbeitung der DIO in RPL erlaubt das zurückgegebene Resultat aus dem Modul umzusetzen. Ist die Validierung erfolgreich, wird die DIO in RPL weiterverarbeitet. Ansonsten wird die

[1] https://tools.ietf.org/html/rfc6550#page41

Verarbeitung der DIO durch RPL übersprungen und die DIO unverarbeitet verworfen.
Das Anhalten des RPL Protokolls bis zum Ergebnis der Validierung kann dazu führen, dass während dieser Zeitspanne neuere eingetroffene RPL-Kontrollnachrichten nicht zeitnah durch RPL verarbeitet werden können. Dies ist der Fall wenn ein Verfahren asynchron arbeitet, bspw. werden bei TRAIL Nachrichten zwischen Knoten ausgetauscht um das Ergebnis der Validierung zu ermitteln. Damit RPL nicht synchron auf das Validierungsergebnis eines asynchronen Verfahrens wie TRAIL blockierend wartet, kann das eingesetzte Verfahren Maßnahmen gegen das Anhalten von RPL implementieren. Beispielsweise kann ein Verfahren die Validierung in zwei Schritten durchführen. Das aufgerufene Verfahren speichert die übergebene DIO temporär ab, und lässt die Callback-Funktion direkt ein negatives Ergebnis der Validierung zurückgeben. Die Validierung durch das Verfahren wird jedoch weiter durchgeführt. Das negative Ergebnis der Validierung bewirkt, dass in RPL die DIO verworfen wird. Der Knoten nutzt den Versender der DIO nicht als *parent* und die RPL-Implementierung kann auf dem Knoten weiter eintreffende Kontrollnachrichten verarbeiten. Ist der Validierungsvorgang des Verfahrens abgeschlossen und der Versender erfolgreich validiert, kann das Verfahren die temporär gespeicherte DIO in RPL einspielen. Durch die Kapselung wird die DIO erneut zur Validierung an die registrierte Callback-Funktion des Verfahrens übergeben. Da die DIO bereits validiert wurde, kann das aufgerufene Verfahren direkt mit positivem Ergebnis der Validierung zurückkehren.
Es sei darauf hingewiesen, dass dabei das eingesetzte Verfahren dafür Verantwortlich ist DIOs und damit die Sicht des Knotens auf die RPL Topologie Konsistent zu halten.

5.4 Umsetzung

Zur Umsetzung des Konzepts wurde ein neues RIOT Modul, `routing_sec_interfaces`, implementiert[2]. In dem Modul ist die Callback-Funktion, die durch ein konkretes Verfahren implementiert wird definiert [3], vgl. Listing 5.1.

Listing 5.1: Definition der Callback-Funktion

```
1  /**
2   * @brief callback prototype definition for DIO protection.
3   *        The implementation is specific to the actual security module that
4   *        provides this callback function.
5   *
6   * @param[in] dio pointer to the DIO structure received by RPL
7   * @param[in] src the ipv6 address of the node that sent the DIO
8   * @param[in] len the fulll size in bytes of the DIO
9   * @param[in] own_rank the current set rank of this node
10  * @param[in] version the current DODAG-version of this node
11  *
12  * @return RPL_SEC_IF_SUCCESS if the actual called security module decided
```

[2] https://github.com/BytesGalore/RIOT/tree/rpl_add_sec_interfaces/sys/net/routing/routing_sec_interfaces
[3] https://github.com/BytesGalore/RIOT/blob/rpl_add_sec_interfaces/sys/net/routing/routing_sec_interfaces/rpl_sec_interfaces.h#L82

```
 *                              the given parameters are verified and the DIO
 *                              procesing in RPL may be safely continued.
 *
 * @return RPL_SEC_IF_VERIFICATION_FAILED if the actual security module decided
 *                                        the given parameters are not valid
 *                                        and the DIO should not be processed
     further.
 */
typedef int (*rpl_sec_if_dio_cb)(gnrc_rpl_dio_t *dio, ipv6_addr_t *src, uint16_t
    len, uint16_t own_rank, uint16_t version);
```

Die konkret implementierte Callback-Funktion registriert ein Verfahren zusammen mit einem Präfix, für den eine Validierung vollzogen werden soll, bei dem `routing_sec_interfaces` Modul[4], vgl. Listing 5.2.

Listing 5.2: Funktion zum registrieren der Callback Funktion

```
/**
 * @brief register a callback function being called on arriving DIOs
 *
 * @param[in] cb the function to be called as callback
 * @param[in] prefix the prefix this function belongs to,
 *            i.e. define which dio->dodag_id should be protected with this call
 * @param[in] prefix_len the number of significant bits used in the given prefix
 *
 * @return RPL_SEC_IF_SUCCESS if the callback has been succesfully registered
 * @return RPL_SEC_IF_ERROR_REGISTER_CB_FAILED if the callback could not be stored
 */
int rpl_sec_if_register_dio_cb(rpl_sec_if_dio_cb cb, ipv6_addr_t prefix, size_t
    prefix_len);
```

Das Tupel aus Callback-Funktion und Präfix wird in einer Liste gespeichert, deren maximale Größe zur Compilezeit festgelegt wird[5].
Die RPL-Implementierung ist erweitert worden, sodass eintreffende DIO-Nachrichten das `routing_sec_interfaces`-Modul aufrufen bevor die DIO weiterverarbeitet wird, vgl. Listing 5.3.

Listing 5.3: Einstiegspunkt zum `routing_sec_interface`-Modul

```
#ifdef MODULE_ROUTING_SEC_INTERFACES
        if(rpl_sec_if_verify_dio_parent(dio, src, len, dodag->my_rank, dodag->
            version) == false) {
            RPL_SEC_DIO_UNLOCK();
            return;
        }
#endif
```

[4] https://github.com/BytesGalore/RIOT/blob/rpl_add_sec_interfaces/sys/net/routing/routing_sec_interfaces/rpl_sec_interfaces.h#L100

[5] https://github.com/BytesGalore/RIOT/blob/rpl_add_sec_interfaces/sys/net/routing/routing_sec_interfaces/rpl_sec_interfaces.h#L33

Die aufgerufene Funktion unterbricht dabei die Verarbeitung der DIO in RPL und kann abhängig vom Rückgabewert die Weiterverarbeitung der DIO erlauben oder überspringen lassen. Im letzteren Fall wird die DIO durch RPL verworfen.
Wurde eine DIO eines potentiellen *parents* an das `routing_sec_interfaces`-Modul übergeben, verteilt es die Validierungsanfrage an ein konkretes registriertes Verfahren. Das `routing_sec_interfaces`-Modul ruft dafür die Callback-Funktion des zur DODAGID per LPM am besten passenden Verfahrens auf[6].
Das Resultat dieses Aufrufs wird weiter an die im Listing 5.3 aufgeführte Funktion in der DIO-Verarbeitung von RPL zurückgegeben. Existiert kein passender registrierter Präfix für die DODAGID aus der DIO, wird ein zur Compilezeit voreingestellter Rückgabewert zurückgegeben[7].
Für den Fall, dass zeitverzögertes Eintragen eines *parent* durch ein Verfahren erforderlich ist, wurde die RPL-Implementierung dahingehend erweitert, dass konkurrierend DIOs an RPL übergeben werden können. Hierfür wurde ein Mutex in die RPL-Implementierung eingeführt der ausschließlich einkompiliert wird, wenn das `routing_sec_interfaces`-Modul genutzt wird, vgl. Listing 5.4.
Das Besetzten und die Freigabe des Mutex wurde in Makrofunktionen umgesetzt. Dies erlaubt, dass die RPL-Implementierung keinen zusätzlichen Funktionsaufruf durchführt wenn das `routing_sec_interfaces`-Modul nicht zum Einsatz kommt. Die dann leere Makrofunktion wird durch den Compiler komplett heraus optimiert.

Listing 5.4: Mutex für konkurrierende DIO-Verarbeitung

```
1  #ifdef MODULE_ROUTING_SEC_INTERFACES
2  #include
3  /**
4   * @brief mutex to protect concurrent passing of DIOs to RPL.
5   *        This can happen when a security module perfoms
6   *        an asynchronous protection,
7   *        e.g. in TRAIL the path validation requires a certain amount
8   *        of communication before a path can be verified.
9   */
10 mutex_t mtx_dio_process = MUTEX_INIT;
11 /** macro to lock the receive DIO mutex */
12 #define RPL_SEC_DIO_LOCK()   (mutex_lock(&mtx_dio_process))
13 /** macro to unlock the receive DIO mutex */
14 #define RPL_SEC_DIO_UNLOCK() (mutex_unlock(&mtx_dio_process))
15 #else
16 /** without using MODULE_ROUTING_SEC_INTERFACES we just do nothing */
17 #define RPL_SEC_DIO_LOCK()
18 /** without using MODULE_ROUTING_SEC_INTERFACES we just do nothing */
19 #define RPL_SEC_DIO_UNLOCK()
20 #endif
```

[6] https://github.com/BytesGalore/RIOT/blob/rpl_add_sec_interfaces/sys/net/routing/routing_sec_interfaces/rpl_sec_interfaces.c#L77
[7] https://github.com/BytesGalore/RIOT/blob/rpl_add_sec_interfaces/sys/net/routing/routing_sec_interfaces/rpl_sec_interfaces.h#L41

Durch den eingeführten Mutex kann eine konkretes Verfahren zu jeder Zeit eine DIO erneut in RPL einspielen, um eine erneuten Validierungsvorgang zu starten. Hierfür wird vom konkreten Verfahren die DIO direkt an die RPL Funktion zur Verarbeitung einer empfangenen DIO übergeben[8].
Die neuen Erweiterungen in RPL werden nur einkompiliert, wenn das `routing_sec_interfaces`-Modul eingesetzt wird. Da eine konkretes Verfahren die angebotenen Schnittstellen des `routing_sec_interfaces`-Moduls nutzt, muss das Modul als Abhängigkeit in der `Makefile` des Verfahrens eintragen sein. Damit werden die Erweiterungen in die RPL-Implementierung von RIOT automatisch eingefügt, wenn ein Verfahren dass die angebotenen Schnittstellen des `routing_sec_interfaces`-Moduls nutzt, zum Einsatz kommt. Wird kein Verfahren genutzt, und damit das `routing_sec_interfaces`-Modul nicht eingesetzt, kommt so automatisch nur die ursprüngliche RPL-Implementierung zum Tragen.

[8] https://github.com/BytesGalore/RIOT/blob/rpl_add_sec_interfaces/sys/net/gnrc/routing/rpl/gnrc_rpl_control_messages.c#L542

Literaturverzeichnis

[1] V. Fuller, T. Li, J. Yu, and K. Varadhan, "Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy," RFC 1519, IETF, September 1993.

[2] R. Hinden and S. Deering, "IP Version 6 Addressing Architecture," RFC 4291, IETF, February 2006.

[3] Emmanuel Baccelli, Oliver Hahm, Mesut Günes, Matthias Wählisch, and Thomas C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *Proc. of the 32nd IEEE INFOCOM. Poster*, Piscataway, NJ, USA, 2013, IEEE Press.

[4] D. Johnson, Y. Hu, and D. Maltz, "The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4," RFC 4728, IETF, February 2007.

[5] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, JP. Vasseur, and R. Alexander, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks," RFC 6550, IETF, March 2012.

[6] A. Dvir, T. Holczer, and L. Buttyan, "VeRA - Version Number and Rank Authentication in RPL," in *IEEE 8th International Conference on Mobile Adhoc and Sensor Systems (MASS)*, Oct. 2011, pp. 709–714.

[7] Heiner Perrey, Martin Landsmann, Osman Ugus, Matthias Wählisch, and Thomas C. Schmidt, "TRAIL: Topology Authentication in RPL," in *Proc. of Intern. Conf. on Embedded Wireless Systems and Networks (EWSN '16)*, New York, NY, USA, Feb. 2016, pp. 59–64, ACM.

[8] P. Thubert, "Objective Function Zero for the Routing Protocol for Low-Power and Lossy Networks (RPL)," RFC 6552, IETF, March 2012.

[9] Kevin Weekly and Kristofer Pister, "Evaluating Sinkhole Defense Techniques in RPL Networks," in *Network Protocols (ICNP), 2012 20th IEEE International Conference on*, Nov. 2012, pp. 1–6.

[10] Martin Landsmann, Heiner Perrey, Osman Ugus, Matthias Wählisch, and Thomas C. Schmidt, "Topology Authentication in RPL," in *Proc. of the 32nd IEEE INFOCOM. Poster*, Piscataway, NJ, USA, Apr. 2013, IEEE Press.

[11] IEEE Std. 802.15.4-2011, *Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (LR-WPANs)*, IEEE, 2011.

[12] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," RFC 7159, IETF, March 2014.

[13] Martin Landsmann, Peter Kietzmann, Thomas C. Schmidt, and Matthias Wählisch, "Demo: Topological Robustness of RPL with TRAIL," in *Proc. of Intern. Conf. on Embedded Wireless Systems and Networks (EWSN '16), Demonstration*, New York, NY, USA, Feb. 2016, pp. 219–220, ACM.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 13. Juni 2016 Martin Landsmann