INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Implementing IPsec ESP in RIOT OS

Maximilian-Matthias Malkus

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Implementing IPsec ESP in RIOT OS

Maximilian-Matthias Malkus

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Tobias Guggemos
Abgabetermin:	18. September 2019

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 17. September 2019

. . . . . .
(Unterschrift des Kandidaten)

Abstract

Despite high security demands in distributed embedded hardware, IPsec is not often deployed in IoT. One reason for this is the packet overhead taking its toll, especially with wireless communication. To significantly reduce this overhead EHC and Diet-ESP are in development and are already in the process of standardization. Many other additions to IPsec, improving its IoT suitability and versatility, are also in the making. After Diet-ESP was successfully brought to the Contiki operating system, the plan was minted to also bring it to the modern, IoT focussed RIOT OS operating system. To enable EHC and other advanced extensions to the IPsec suit, a versatile but minimal IPsec implementation was needed. In this thesis I am designing, implementing and evaluating a realization of the basic IPsec features, considering the demands of the related works, especially in the realm of my department. The goal is an agile implementation that fits a broad set of scenarios, is easy to comprehend and modify, while following the coding standards of the RIOT OS Open Source project, so that it can eventually be deployed into the official kernel code.

Contents

1 Introduction

Even though the promises of Moore's Law started to fade a bit in recent years, the technological trend to ever more integrated and power efficient devices continues. This is especially true in the segment of lower end computing power. It seems like every other day a new System on Chip (SoC) is introduced to the public, beating all predecessors in functionality, resources, clock speed, size and cutting the price in half at the same time. More often than not they also pack the incredible marvel to be extremely power efficient if configured just right. This development started an ever increasing shift, away from stationary mains powered devices to an every increasing number of battery powered and often mobile solutions. In recent years, these systems have become so efficient, that there are devices without the need to charge or change battery during their full intended lifetime. These low power, single purpose devices are tasked with ever increasing variety of roles in the industrial and domestic realms. No longer left to simply being controllers for something but often featuring as sensors and switches that need to also communicate information to the other entities, often by radio signals. Despite all this progress in integration and thus reduction in power consumption, a device that communicates wirelessly still needs to generate a carrier signal. Producing radio waves is and will always stay physical work and thus remains a power hungry task. Because of that we are already at the technological tipping point, where the transmission of information makes up for the major energy depletion for this type of devices. [18] Thus every reduction of network bandwidth, resulting is less radio frames to be sent out, will directly benefit the battery powered lifetime of a device.

For distributed systems it is easy to use the already existing world wide network, that the internet provides, to communicate over longer distances and outside of closed local networks. Consequently the term Internet of Things (IoT) was coined in the notion that every item could in theory be connected and addressable inside this global network. In this case it is rather obvious that some form of traffic protection is desirable. But even in closed networks, encryption can drastically reduce the vulnerability of a system and is often mandatory to protect sensitive data from unauthorized entities. An established and routed standard for secure communication over the internet is the third layer IP extension header ESP from the IPsec framework. The framework also provides an IP level firewall and a range of dynamic key and session negotiation techniques.

Naturally, the use of ESP protection has its downsides. One is an increased processing overhead. But that is close to negligible in the light of ever increasing processing efficiency. The other is an overhead created in the networking bandwidth. To make matters worse, increasingly power efficient standards are deployed for IoT wireless networks. These standards often significantly reduce the bandwidth and the maximum payload size in favour for power efficiency. The most established modern standard at the time of writing is IEEE 802.15.4. [5] It defines the Maximum Transfer Unit (MTU) to be only 127 bytes long. In a realistic scenario of encrypted traffic, this leaves the application layer with only 33 bytes of data payload, considering the depletion by the encryption and the MAC, IPv6 and UDP headers.

[13] With a payload this small it is not a far stretch to imagine a single bit of overhead causing the need for a second packet to be sent out, doubling the packet number and thus the already high energy consumption of the wireless message. This is where ESP Header Compression (EHC), deployed in the form of the Diet-ESP strategy, comes into play. It aims to crunch every last bit of irrelevant header information. Since ESP communication is based on a secure session that is negotiated by an Internet Key Exchange (IKE) protocol, many connection details are already known to the initiated peers and are thus partially or even fully redundant inside the datagram headers. Thus encryption with Diet-ESP compression can, in a best case scenario, even reduce the packet count compared to the unencrypted non ESP message. All that is possible while providing full security compliance with regular IPsec and not affecting the routing of the packet. Therefore this is a valuable addition to secure distributed communication.

The specialized embedded devices, used in low power distributed networks use operating systems, specifically tailored to these scenarios. At the time of writing my department prioritized the LGPL licensed and open source operating system RIOT OS. It combines many desired features with an active community and a license that does the splits between open access and professional commercial interests. Detailed reasoning for this is provided in section 2.3. At the current state though, Riot OS does not feature ESP or any IPsec functionality, hence the goal of this thesis is to build a foundational implementation of IPsec into the GNRC network stack of RIOT OS. This work does aim for a feature complete IPsec implementation, but for it to serve as a baseline for future development and the implementation of Diet-ESP and various other advanced protocols working with or within IPsec. Some other examples for this are G-IKEv2 or Implicit IV, who I will address and explain in more detail in the Related Work section 4. Aiding further development, the provision of tidy code, clear structure and verbose documentation was as important as making sure the implementation would offer enough versatility to serve as a platform for the mentioned protocols. At the same time I also tried to stay compliant to the coding community standards of the RIOT OS Open Source Project in order to ease a potential integration into the public kernel code.

The structure of this thesis will firstly elaborate on the backgrounds and related works, to then derive the design from them. Following this, will be the report about the implementation, including results, obstacles and solutions that occurred while manifesting the design into RIOT OS. After a short evaluation of the work the thesis will conclude with my experiences regarding IPsec and RIOT while implementing it and my suggestions for future progression of this Implementation.

2 Background

The protocols and standards my thesis is build upon and the background referred to throughout the thesis will be presented in the following. I will also elaborate some concepts that will help with future development and analysis of the implementation.

2.1 IPv6

IPv6 is in recent years finally getting the widespread use that it deserves and slowly pushes IPv4 toward deprecated technology. This is fuelled by IoT and its need to communicate over the internet. In the past often only servers were in the need for unique global IP addresses. The now rising demand for them - and especially the lack of them - gives way to a more modern IP standard. IPv6 vastly increases the address space and starts to get rid of IPv4 necessities like Network Access Translation (NAT). But with its features also come limitations. An IPv6 header uses 32 bytes alone for its source and destination address, compared of the meagre 8 byte of IPv4. This hits especially in combination with every smaller packets for low powered wireless communication.

In figure 2.1 you can see the structure of the IPv6 header. Any optional Extension Header (EXT) would directly follow on this and its protocol number will reside in the IPv6's Next Header field. The Next Header field of the EXT header will give the protocol number of the payload or the next EXT header. There can be multitude of EXT header, but in general every header may only appear once and although there is no strict rule for it, it is universal that if the ESP type EXT header is used, it always comes last.

2.2 Internet of Things (IoT)

The modern term Internet of Things (IoT) is associated with the high level of integration and numerous numbers of devices. The number of internet connected devices in this domain has joined ranks with the number of mobile phones in use worldwide in just a short time. There are a 6.2 billion estimated devices and this number is expected to more than double until the year 2023. [6]

Embedded Hardware

Packing limited resources, embedded devices were historically mostly found in a hardware controller setting, where the devices software acted more as a firmware to the physical device. Through increasing capabilities, these devices became ever more interconnected and remote from greater hardware. Now one can even find single sensors, remotely deployed, powered by a battery that will last for the devices lifetime, communication wirelessly over the Internet. To be able to differentiate between the different systems and their needs RFC 7228 defines device classes and coined the term 'constrained node' to describe these connected systems

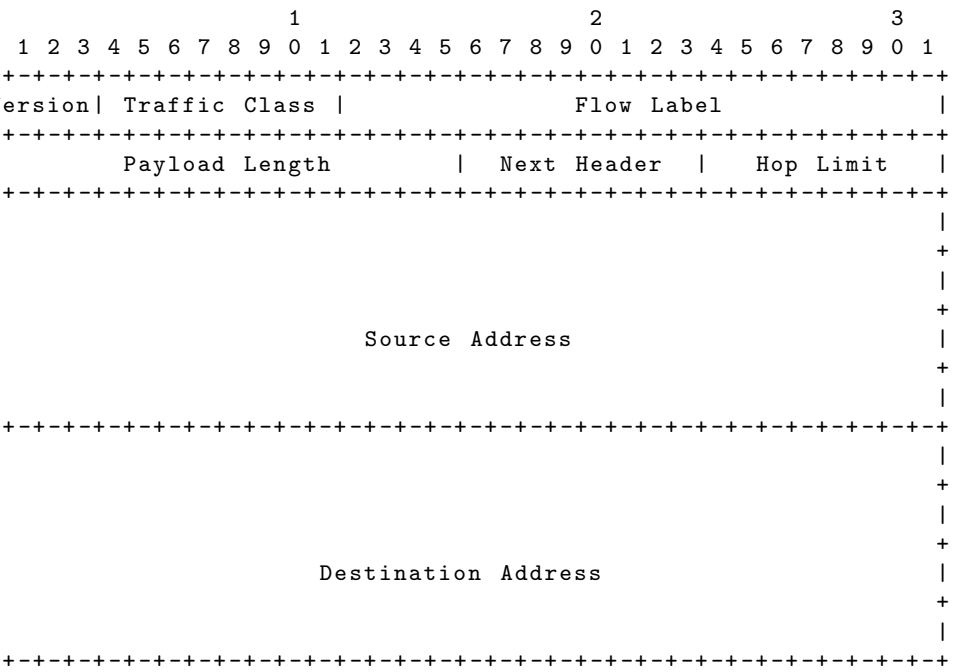

Figure 2.1: IPv6 Header

that work with a mere fraction of resources compared to regular desktop or server machines. [9] These devices form the basis for modern IoT systems. Our focus lies on the higher end of constrained nodes that are able to wield an embedded operating systems like RIOT OS [8] or Contiki [10] while using an IPv6 stack at the same time.

Besides the advances in integration and production miniaturisation that provide the bulk of additional computation power and lower prices, the increased battery life also stems from efficient use of sleep states. These sleep states can go down all the way to a so called 'deep sleep'. Here only the tiniest fraction of circuitry is running a timer, needing less power than a wristwatch, that will wake the system in regular intervals. All hardware is designed the way, that quick wakeups utilize the full potential of the chips so the device can go to sleep again often in fractions of seconds.

This sleepy behaviour can cause all sorts of troubles in the design of operating systems, software and hardware. This would be beyond the scope of this thesis. But besides availability there are other features that these highly effective devices are often missing. The most significant one is possibly that nearly all IoT devices operate on the idea of stateless reboots. That is, between reboots or power loss, no data is preserved and the boot sequence solely relies on the device configuration. That way a device can compensate for a system failure without risking to boot back into the failed state. And that is also why devices often auto-reboot on system error events. Moreover many constraint nodes also do not supply reliable time keeping by their own, thus any system component or protection logic relying on timely factors need to account for this. Another factor to mind is the amount of integration for certain ciphers. Established ciphers and suits like AES and Transport Layer Security (TLS) are often already supported in embedded hardware and thus gain a major processing

performance benefit.

6LoWPAN

The general standard of 'Low Power Wireless Personal Area Networks' (6LoWPAN) defining a separated network, called a Personal Area Network (PAN) was developed to increase efficiency of IPv6 packets in IEEE-802.15.4 networks. It is comprised of three sub headers. Mesh addressing header, fragmentation header and compression header. On IEEE-802.15.4 networks it especially enables improved payload fragmentation compared to IPv6 without 6LoWPAN. [27]

The standard sits at the upper end of OSI Layer 2, the Data Link Layer, and lowers packet forwarding from Layer 3 to Layer 2 inside of the PAN, to ease mesh networking. Mesh networks are defined by packets often taking multiple routing hops to their destination throughout the PAN. Since a PAN is a closed group with entities of comparable hierarchy, the MAC address is enough for routing inside of a PAN. Thus IP header can be nearly fully omitted and replaced by a PAN identifier. Additional to this, 6LoWPAN enables a very effective header compression for OSI Layer 3 and Layer 4 protocols. The compression context can be pre negotiated but normally is included into the packet in the form of compression flags. [28]

The 6LoWPAN edge router at the border of a PAN enables full compatibility with regular IP based networks. Inside of the PAN devices are divided between end point devices called Hosts and Routers. Hosts are normally comprised of sleepy, low energy devices, where routers are more powerful - always available - nodes of the network. The downside is that the 6LoWPAN protocol can only be deployed inside of a PAN scope and always needs an edge router to communicate outside the border of the PAN. As such, single remote devices intended to communicate over common internet network infrastructure can not utilize 6LoWPAN.

6LoWPAN Compression for AH and ESP headers has been considered, but for various reasons never left the stage of draft [24] Most notably since this compression would only occurs after a possible ESP encryption, most of the reductive potential for the ESP header and the protocol headers residing inside of its payload are neglected.

PF_KEY abstraction API

The benefits of abstracting the Key Engine from the Key Management are manifold. Besides an increased agility, using a standardized abstraction could enable one to connect two independently developed solutions. One such standard is the PF_KEY API designed by the American military. [19]

2.3 RIOT OS

RIOT OS [7] has a slim footprint and already offers support for a lot of the common network capable development boards, used for IoT. The system is very modular and its monolithic kernel gets compiled with minimal dependencies for the specific task. Because of this, its resource use is very low, regarding memory utilisation and processing overhead. This aids in our goal to minimize power consumption. Another valuable aspect of RIOT OS is it being licensed under GNU-LGPL-2.1. This Lesser General Public License combines the benefits

of open source development, while at the same time offering companies the option to secure their intellectual properties. Code derived from LGPL code must be published under the same open license, but when only linking to LGPL code the remaining source can be licensed anyway one wants. This is a strong foundation for an IOT operating system, since it is a good tradeof between secure code, public development and the challenges of the professional industry and corporate practices.

Features

In figure 2.2 one can see the common talking points giving RIOT OS an edge over its competitors. In standard configuration it already supports Multithreading while at the same time providing realtime computation, through priority scheduling. This combination will get increasingly important with growing computational power of constrained nodes. The high modularity of the operating system also makes it easy to implement and deploy a variety of standards to its kernel.

OS	Min RAM	Min ROM	C Support	C++ Support	Multi-Threading	MCU w/o MMU	Modularity	Real-Time
Contiki	<2kB	<30kB	○	✘	○	✓	○	○
Tiny OS	<1kB	<4kB	✘	✘	○	✓	✘	✘
Linux	~1MB	~1MB	✓	✓	✓	✘	○	○
RIOT	~1.5kB	~5kB	✓	✓	✓	✓	✓	✓

TABLE I
KEY CHARACTERISTICS OF CONTIKI, TINYOS, LINUX, AND RIOT. (✓) FULL SUPPORT, (○) PARTIAL SUPPORT, (✘) NO SUPPORT. THE TABLE COMPARES THE OS IN MINIMUM REQUIREMENTS IN TERMS OF RAM AND ROM USAGE FOR A BASIC APPLICATION, SUPPORT FOR PROGRAMMING LANGUAGES, MULTI-THREADING, MCUS WITHOUT MEMORY MANAGEMENT UNIT (MMU), MODULARITY, AND REAL-TIME BEHAVIOR.

Figure 2.2: RIOT OS Comparison - Source: [8]

Memory management

As already mentioned, most embedded systems do not provide dynamic memory allocation. So does RIOT, but some kernel modules provide for a higher level dynamic data management. For example the concept of the packet buffer (pktbuf) enables network packet creation and memory reservation for packets in the form of a linked list. Every item of the linked list is a packet snip (pktsnip) and can hold a whole network packet, or single segments of it. On sending, all snips get merged into one data block. If a packet has no more users, the according buffer space will be freed by the kernel and can be reused.

2.4 GNRC

In figure 2.3 one can see the architecture of the GNRC network stack. It is RIOTs default network stack. The application layer communicates using the Sock API, while packets get passed between the layers by utilization of the netapi message buffer. Every layer runs on its own thread, with different priorities to ensure processing.

To split a received packet into smaller snips it needs to be marked accordingly. The order of the pktsnips in a packet is reversed depending on the direction of the packet. Traffic to be sent (SND) is stored in reversed order to received traffic (RCV). This reduces unnecessary

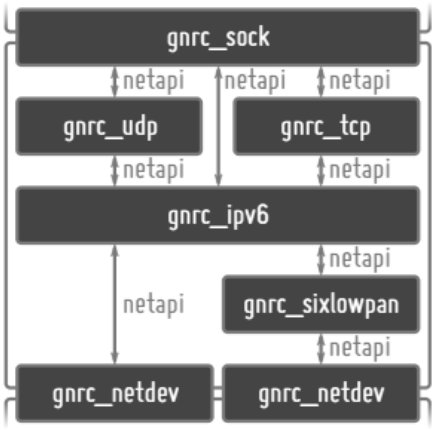

Figure 2.3: GNRC Network Stack - source: [1]

memory duplications while processing. The kernel routines mind this and also abstract this behaviours of the GNRC IP stack, for example in 'gnrc_pktbuf_mark()', used to split a pktsnip.

GNRC NetAPI

The NetAPI system provides a message bus, that links messages to pktsnips, keeping track of the users. It works by threads registering their interest in messages about a certain packet type to the message bus and then waiting for a response. They will get called by the kernel if a netapi message for their registered type is available.

3 IPsec

Making up the bulk of background, I saw it necessary to give the details on the IPsec suit an own chapter to be able to talk about the numerous features. Not often in computer science is there such a behemoth of interconnected protocols, methods and functionalities intertwined into one single interdependent standardized concept. Therefore the standard is stretched over strikingly many articles, contemplating specific aspects of some submodules and adding obligatory details to others. After going over all the details necessary for my design I will formulate a set of Requirements the implementation needs to meet to be of proper use for our goals.

I want to mention, that I will purely focus on IPsec in the context of IPv6, since RIOT OS only supports IPv6 networking. IPsec is designed to protect IP based communications on a kernel level. Thus it is abstracted from the application layer and valid for all applications that use the kernels IP stack. The suit is comprised of a set of specialized protocols for encryption and negotiation, a traffic detection and filtering mechanism and a database architecture that defines the behaviour and ties the parts together.

3.1 IPsec Protocols

The IPsec protocols reside at the lower end of OSI layer 3, occurring after the IP generation but before the network layer. The protocols available in IPsec are Authentication Header (AH), Encapsulating Security Payload (ESP) and Internet Key Exchange (IKE). While AH and ESP are designed as IPsec EXT headers, IKE is a separate protocol proposal. Though there are multiple protocols the quasi standard at the time of writing is IKEv2 [16]. From here on IKE and IKEv2 will be used interchangeably if not stated differently.

ESP

The Encapsulating Security Payload (ESP) is an IPv6 EXT header capable of of encrypting and authenticating the IPv6 payload. The ESP protocol enables us to protect communication from point to point but it also features the ability to securely transport and mask other traffic by tunnelling it. This is a vital and widely used component of IPsec and allows us to build encrypted and authenticated Virtual Private Networks (VPN). Especially in our field of focus, the Internet of Things, this helps deploying distributed networks in a safe, agile and reliable way.

ESP Protocol Fields

The ESP block is essentially one long IPv6 EXT header without a trailing payload. Itself consists of a header, the payload and a trailer. Depending on the encryption algorithm, additional information like Initialization Vectors (IV) can also be stored inside the payload, but

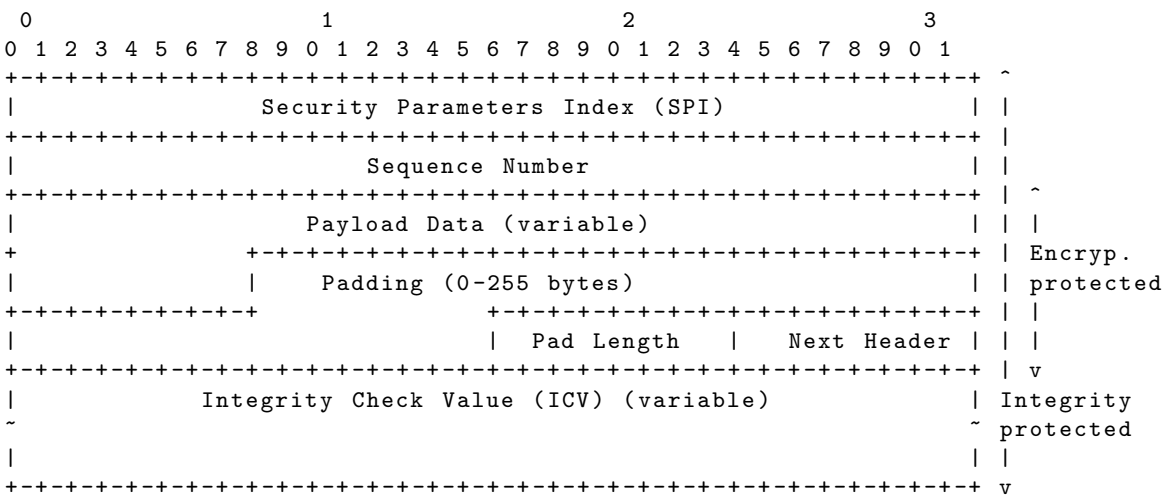

Figure 3.1: Generalized IPsec ESP header

that is to be handled by the cipher method itself. A graphical description of the standardized ESP header can be found in figure 3.1.

For common ESP traffic the header begins with the 32-bit Security Parameters Index (SPI). This SPI is used by the receiver to identify the negotiated session parameters associated with this connection. It is followed by the 32-bit Sequence Number (SN) used to protect from replay attacks and often as a nonce by ciphers. The payload is followed by up to 255 bytes of padding to conform to standard IPv6 64-bit word width and to enable traffic shaping. The payload is followed by an 8-bit field to store the padding length, an 8-bit field for the IP Protocol Number of the encapsulated packet. A Integrity Check Value (ICV) trails the packet if authentication is used. The size is determined by the cipher used but mostly in a 32-bit field. There are alternative setups that can be negotiated between two peers. For example an extended 64-bit SN can be used. This is a reason why the negotiated session parameters are firstly looked up via SPI, in the IPsec databases, before further processing of the packet.

As marked in figure 3.1 certain parts of the ESP header are encrypted, namely the payload and trailer and a bigger portion of it is integrity protected. The encryption is performed first and the integrity algorithm is applied afterwards. Because the ESP Authentication Header in the form of a ICV is not protected by encryption, a keyed integrity algorithm must be employed to compute the ICV.

AH

The main difference between AH and ESP lies in the fact that for ESP authentication, only the header is hashed, while AH hashes the whole IP datagram. This includes all immutable IP header fields, like source and destination addresses, ports, etc. AH is intended to protect from IP header forging. The AH protocol also supported an extended role in the IPsec suit and could in theory be used in combination with ESP or as authentication alone.

Against common opinions, there are a few cases where authentication without encryption is needed. For example in cases where integrity is needed but encryption is prohibited, like communication over the amateur radio band in the USA. [23] But at the time of writing, the AH protocol has few champions.

Why to choose ESP over AH

AH fell out of favour in times the Internet emerged and the limitations of IPv4 became clear. The technique of Network Address Translation (NAT) is designed to compensate for these limitations by changing every port based connection from a machine in a local network into a request from the Gateway on different connection specific ports. Obviously this forges the IP header and thus the authentication of AH rightfully failed. Thus AH was limited to be used for direct connections only. Since there was a way to route ESP traffic through the NAT, by using NAT Traversal (NAT-T) UDP wrapping, it was common practice to use self encapsulating ESP packets to substitute for AH on the few special cases where the whole IP packet needed authentication. Consequently to AH being not supported by many implementations and the fact that its features can be emulated by ESP Self Encapsulation, its role in Internet traffic is marginal and some even call for it to be removed from the musts of the IPsec standard. I will give a more detailed description about full packet integrity protection by ESP in subsection 3.3 under Self Encapsulation.

3.2 IKE

The standardized way for key management and session negotiation over public networks is the Internet Key Exchange. Commonly the negotiation of Security Associations (SA) is handled by the Diffie Hellman [25] based IKEv2 Protocol, being the quasi standard at the point of writing [16]. IKEv2 is not the end of the line, though. There are other protocols supporting special use cases. One protocol our department has a greater focus on is the 'Group Key Management using IKEv2' (G-IKEv2) protocol [29]. It uses a centralized Group Key Manager entity in the network that provides for very efficient, dynamically negotiated multicast ESP communication. This includes renegotiation for forwards- and backwards-secrecy in case the group constellation changes. Before this, multicast communication in IPsec was handled crudely and offered little versatility. I kept these advanced possibilities in mind leaving them room to fit into the design of the IPsec implementation, especially when designing the IPsec databases.

3.3 ESP Traffic

The traffic generated by the ESP protocol can take various forms, fulfilling different use cases and having specific requirements for key and session negotiations. Therefore ESP demands special attention in implementation and integration into an existing IP stack.

Transport Mode Traffic

Transport Mode traffic is a unidirectional form of ESP encryption of IP traffic between two peers. After settling on the session details and keys needed, the sender's IP packet is modified using ESP. The ESP header will wrap around the application layers datagram. The

receiver will detect and disassemble it. The application datagram will be extracted from the encrypted IP packet. Intermediate entities simply route the IP packet to its destination. If a bidirectional communication is needed, a second channel with reversed hierarchies needs to be negotiated and managed in the database. The payload encapsulated by the ESP packet only consists of the encrypted payload of the original IP packet and possibly additional vectors used by the chosen cipher algorithm. The structure of this assembly, especially in contrast to Tunnel Mode can be seen in figure 3.2 and 3.3.

Tunnel Mode Traffic

The Tunnel Mode traffic differs from the Transport Mode in a way, that the full original IP packet will be encapsulated inside the encrypted ESP payload. Also a new IP header will be generated based on the tunnel addresses defined in the connections SA. This enables various network topologies, since the traffic can be routed however the SA defines. The most common use is the tunnelling of traffic by two routing gateways. That way the routers protect the traffic they route by using ESP Tunnel Mode, enabling a secure transmission and making the route invisible for the communicating clients, since the receiving Gateway will simply rebuild the original IP packet. This setup is often used in Virtual Private Networks (VPN) to securely connect two separate networks over an untrusted connection.

As stated in 2.1 an ESP header may only occur once. Under those circumstances, packing ans ESP transport packet into an ESP tunnel packet, thus ending up with two ESP headers in one packet is achieved the following way: The ESP trailer is no separate entity but simply the end of the "first" EXT header and the according IPv6 packet has no ordinary payload. The original payload is encapsulated inside the ESP EXT header and thus the payloads content is masked from regular EXT header handling of the operating system and irrelevant for the current EXT processing. This is until after the decapsulation happens. At this stage the "first" EXT header is consumed and removed from the packet, leaving the packet with new decapsulated ESP header. In theory this behaviour can be stacked.

Self Encapsulation

A special way to deploy the Tunnel Mode is to generate a self originating ESP Tunnel packet. In this setup the newly created IP header is a mere copy to the encapsulated original header and not generated anew. It serves the purpose of full packet integrity protection and could in theory even be deployed without encryption, if a special case demanded it. The receiver can then check the IP header's fields on similarity, minding the changed packet length and the mutable fields of the IP header. The self encapsulation is not triggered by a specific signal in the database, but automatically detected and concluded by the identity of the original addresses and the ones defined as the tunnel destination and source addresses within the SA entry.

Multicast Traffic

In group communication scenario negotiating a SA with every group member and sending every group packet out to every participant would be tedious. To ease this one can use IPv6 multicast. Clients can register for a multicast group, share the multicast SA for receiving as well as for sending and get routed traffic accordingly. Thus a sender as well as any router only needs to forward a multicast packet once per interface. Also a single client does not

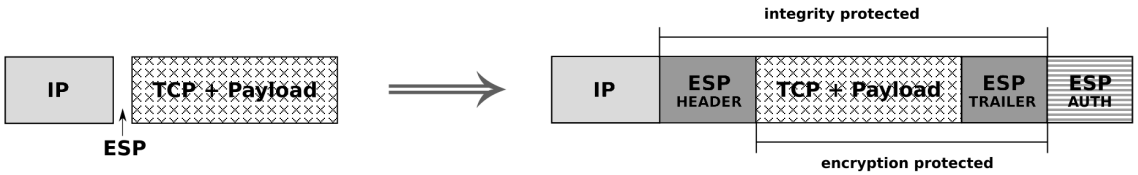

Figure 3.2: ESP wrapping in Transport Mode

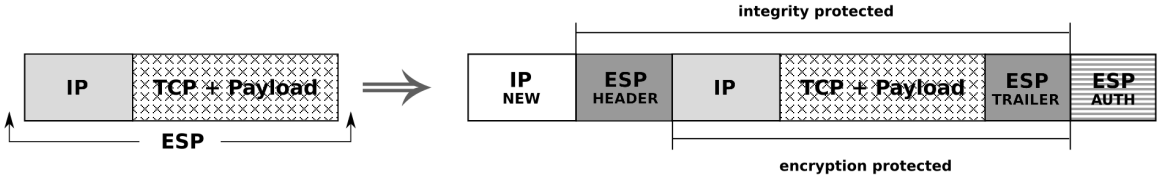

Figure 3.3: ESP wrapping in Tunnel Mod

need to keep track of all members. This is even more relevant in IPsec terms, where a pair of SA would need to be negotiated and kept by every communicating pair of entities.

The problem that comes with encrypting multicast IPv6 traffic is that a regular Diffie-Hellman exchange is not possible with more that two parties. A negotiated secret could be shared, but keeping forward and backward secrecy for all entities is not possible. Therefore regular multicast ESP works with manual static keys and one secret shared by all members of this multicast group. This makes the group complex to setup and on a topology change, or a security breach with one member, all members need to manually change their setup. This is an especially bad scenario in IoT where device settings are often compiled into the system.

Multicast setup in IoT is desired though and thus there are multiple approaches aiming to solve the issue of a modern dynamic key management for multicast IPv6 ESP traffic. None of these are standardized yet.

ESP Traffic Caveats

To get ESP to work some things need to be considered. In the following section, a few issues that need to be taken into account and can complicate implementation, are described. For an overview of the ESP packet flow inside of a system one can take a look at the ESP traffic flow diagram in figure 5.1

For ESP tunnel traffic, the broadcasting of ICMPv6 over the tunnel is discouraged and the use of manual static routes on the gateways is highly recommended by the standards.

For ESP to work in a dynamic setting ICMPb6, IKEv2 and ESP packets must be able to travel between the communication peers. For self originating and destined ESP traffic the bypassing is handled by the ESP implementation. For IKEv2 and the neighbour discovery protocol ICMPv6 rules for bypassing must be created. See the example setup in section 6.1 for more detailed information.

Routing denotes the forwarding of traffic that the system received but that is not addressed for it. On tunnelling, a a self generated and encapsulated packet, is essentially dropped at the start of IP handling of incoming packets. Thus routing behaviour is triggered. For tunnelling getaways, a rule must be set to allow the bypassing of routed ESP traffic from the sending to the destined gateway.

Traffic Selector

Traffic Selector (TS) is the term for a set of variables used to uniquely identify traffic in the scope of IPsec [26]. As far as the values are available in the specific packet, it is comprised of the source and destination address of the IP header, the next layer protocol type and the destination and source ports of the next layer protocol.

Security Policy Ruleset

For the system to decide how to process the identified traffic, an administrator formulates a set of rules. To work as intended, these rules are to be ordered from specific to more general, as the first match will be selected. Each entry is comprised of:

```
A Traffic Selector to identify the traffic.
A filter rule deciding to either BYPASS, DISCARD or PROTECT.
An option for TUNNEL or TRANSPORT mode of ESP.     (IF PROTECTED)
An option for the encryption mode.                 (IF PROTECTED)
A source and a destination address for tunnelling. (IF TUNNELLED)
A source and destination port for tunnelling.      (IF TUNNELLED)
```

For more information on the encryption modes, see 'Encryption Modes' in section 3.4 The Traffic Selectors can can be used with ranges and wildcards to enable a broader traffic definition. Erroneous setups are possible, for example filtering ports for a non port based protocol or bypassing traffic unintentionally because of wrong order. A correct setup without contradictions is - per IPsec intentions - left in the full responsibility of the administrative entity. More details for the specific elements the rules and what exactly different database entries are comprised of, one best refers to the standards papers describing this in all detail. [26]

3.4 Session Information

Security Association (SA)

A security association is a collection of settings and variables needed by the ESP module and the accompanying encryption methods to process IP traffic flagged as protected. Its content is generated from the managed SPD ruleset, the specific connections details and the results of negotiated session parameters in case of dynamically managed keys over IKE. It generally describes one unidirectional connection and holds the ciphers needed for traffic protection.

The SA can be set up manually but is then limited in its choice of ciphers. [30] The general way to set up an SA is by IKE negotiation of ciphers, cipher options, encryption keys, and other values as for example anti replay windows, timer and byte limits, or further details supported by IKE extensions. Renegotiation of an SA can occur if a peer requests this or because lifetime limiters based on time or transmission quantity were exceeded.

Encryption Modes

ESP supports a subset of encryption and authentication combinations. Firstly there is the authentication only mode, which I already addressed. This mode only verifies the authenticity of the packet. For encryption there are two other modes. One is encryption followed

by authentication. Here both methods, cipher and hash are called one after another and have individual security details defined in the SA. Then there is the combined mode where authentication and encryption are handled by a single call to a modern cipher combination.

Ciphers

ESP is intended for the use with a multitude of different ciphers. The IPsec design separates the cipher and hashing methods from the ESP processing and the traffic filtering logic. Thus in theory any cipher can be chosen for a project. Of course there are a set of standards ciphers, deemed mandatory, suggested or rejected for an IPsec implementation. The responsible committee tries to keep a informed balance between modern cipher support and backward compatibility in the RFC8221 [30] and updates to this topic are frequent.

Manual keying is not supported by modern ciphers and only AES_CBC is allowed by RFC8221 for manual keyed ESP. The modern, counter based, algorithms are very fragile in a manual keyed setup, since the reuse of the same key and counter pair would break all prior and future encryption. Since the capability to produce randomness is limited on embedded systems and a reboot will reset all counters, only dynamic keying is allowed with counter based ciphers. Besides the standard ciphers for IPsec, SHA-256 and SHA-512, a recent development is the combined cipher Chacha20Poly1305 [22]. It offers many advantages. Most notably the good computational performance and a small block size, reducing the need for cryptographic padding. It also uses a shared key for encryption and hashing, reducing the impact on the SAD and also negotiation. RFC8221 handles it as a contender, to be the future standard algorithm for IPsec. Conveniently, support for this cipher was just added to the RIOT OS kernel whilst development of this thesis.

3.5 IPsec Databases

The information, defining the behaviour of IPsec Traffic Filtering are held an managed in the IPsec database. The Security Associations are also saved here. This database is assembled from a set of logical databases for whom the standard intentionally does not have a strict implementation guideline. [17] Therefore a multitude of realizations are possible. The specific design of the databases for this thesis is described in the design section under 5.2 but generally is made of the SPD, SPD-Caches and the SAD.

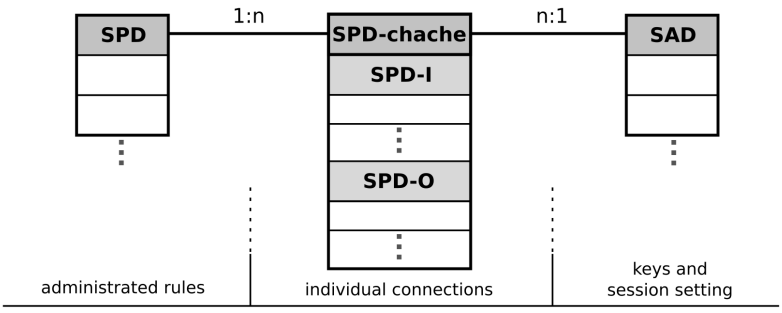

Figure 3.4: IPsec DB Architecture

Security Policy Database (SPD)

The SPD is comprised of entries derived from a manageable SPD rule set. It selectors are based on Traffic Selectors. In an administrated setting any changes to the rule set would update the SPD and the caches, described in the next section. But IoT devices are generally not managed. Because of ranges and wildcards a single SPD rule can be valid for a multitude of connections, a specific cache entry is generated for every real connection at the moment when the traffic is encountered for the first time.

SPD-Caches

On first encounter of a specific traffic selector, a cache lookup will naturally fail. The SPD database generated from the SPD rule set is then queried and a cache entry is created and acted upon. Only when the traffic is ruled to be discarded no cache entry will be generated and the packet is silently dropped. Conceptually the caches are divided in three logical parts. The SPD-I for incoming unprotected traffic, SPD-O for outgoing unprotected traffic and SPD-S for protected traffic. The flow diagram for cache generation can be seen in figure 3.5 For larger systems these caches can be separated into different databases to optimize lookup performance.

It is important to mention that incoming protected traffic gets identified by its SPI and is only checked against the SPD rule entries after an according SA entry was found and the packet was decapsulated.

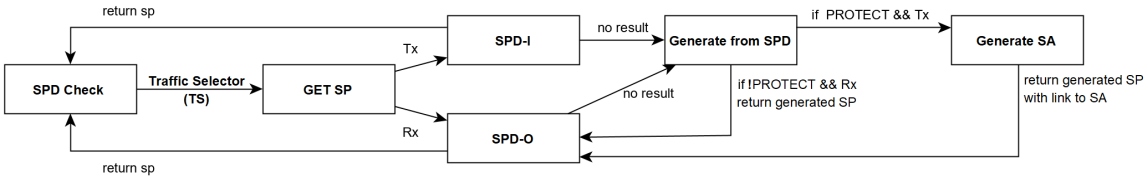

Figure 3.5: Overview over SPD cache generation

Security Association Database (SAD)

The SAD simply contains the SAs that the system has negotiated or was set up with. Every SA can be linked to one or more SPD-S entries. Figure 3.4 illustrates this.

Standardized communication between Key Engine and Key Management

Simultaneously to this thesis, other work on RIOT networking is conducted. Including our department. Various works on Key Management Systems for RIOT are of interest. To ease the integration of different systems, a more generalized message system for communications between Key Engine and Key Management would be good. The PF_KEY message suit [19] defines an overall code hierarchy and a strict messaging based system, combining API abstraction and data exchange. It provides an SOCK based abstraction layer between the Key Engine and the Key Management System.

3.6 Traffic Protection

As noted prior, IPsec is intended to protect all of a systems IP traffic. This protection is made possible by integration of gatekeeping routines into the IP stack. They perform detailed analysis of all IP traffic and are authorized to reroute and even discard any IP traffic according to the IPsec database manifestation. For IP traffic routed through the device the gatekeeping routines even have to perform manual packet inspection of unmarked IP traffic.

Traffic Filtering

After completing the analysis of a packet, one out of three action will be taken. DISCARD, BYPASS or PROTECT. This is decided upon the according information, received from querying the SPD caches, unless for incoming ESP traffic, which is BYPASSED without checks. More specifics on this can be found in the design chapter, section 5.1 under 'Traffic Flow'.

On DISCARD the packet gets dropped and all resources are freed. The BYPASS action will cause the packet to proceed through the IP stack, just as it would have if the IPsec submodule was not deployed. As stated above, the PROTECT action can only occur at the IPsec gateways for outgoing traffic. The packet to PROTECT will therefore be sent to the ESP encapsulation routines.

Traffic Shaping

For the security of a remote system it is often not sufficient to encrypt and authenticate the traffic. In many cases critical private information can be extracted by simply monitoring the traffic pattern [12], also called metadata. By analysing this metadata the type and purpose of specific nodes or even the information transmitted can be extracted. For example, by monitoring for reoccurring message lengths of a connection, conditional information can be snatched, or the obvious example of a door-switch sensor system, sending data whenever a switch is triggered. Even without knowing the position of a specific device, a lot of information can be gathered about the general situation inside the system and even the tracking of individuals is possible. Therefore masking of traffic is not to be neglected in any IoT traffic scenario that would call for protection. This masking of sensitive metadata is called Traffic Shaping and IPsec provides a special method to abstract the generation of spoofed traffic from the application layer. The ESP headers next header field is therefore initialized with toe IP Protocol Number 59 for 'IPv6-NoNxt' and the payload filled with random value of the length requested. On arrival these packets then are discarded without error, despite being of the wrong Protocol Number according to the SPI and are not passed to the application layer. For Diet-ESP it was decided to shift the liability of this into the application layer for it knows best in which ways traffic information can be vulnerable and can counter this.

3.7 Requirements Summary

In the following I will condense the most relevant targets for the design stage.

R1 Reliably detecting traffic by its Traffic Selectors for IPsec to act as a gatekeeper in the IP stack.

R2 Detecting ESP IPv6 EXT headers and relay PROTECTED traffic to the ESP routines.

R3 A database system holding the SAD, SPD and the SPD caches.

R4 Database management, including dynamic cache generation, SA negotiation triggering and sanity checks.

R5 Generation, decomposition and verification of ESP packets based on the according SA information.

R6 Ability to modify self originating IP packets and build ESP Transport Mode headers, based on the according SA.

R7 Detection and composition of self originating ESP traffic in Tunnel Mode.

R8 Implementation and integration of a hash and a cypher for encryption and integrity protection.

R9 A manual way to simulate dynamic key generation and injection of SA and SPD-cache information into the database, in the absences of IKEv2.

R10 Manageability of the basic filter rule set.

R11 The database access should be abstracted by using a Key Engine system to ease modifications to the database realization.

R12 Low resource use regarding threads, memory and computation time to meet the need of running on an constrained node.

R13 Opposed to R12 is the need for a high degree of modularity, to maximize the adaptability of the design to the broad range of specialized scenarios and to generally ease future development.

R14 Conformity to the coding guidelines of RIOT OS Open Source development to enable a integration into the kernel, fuelling a wider adoption of ESP and thus Diet-ESP.

4 Related Work

The motivation for this work were the numerous fascinating developments in the realm of IoT surrounding ESP based IPsec communication. From the handling of complex multi user traffic, where the versatility of IPsec plays out its potential, to the utilization of the session based nature of IPsec for limiting the encryption overhead. The following section aims to give a brief overview and shed some light on the related endeavours on this topic.

4.1 Further exploration and similar endeavours

Implicit IV

An Initialization Vector (IV) is used by most ciphers to encrypt the payload and sent out with the encrypted data. Because of the separated nature of ESP and the cipher suit, the cipher simply writes the IV into the payload block. Thus IV and ciphertext are handled as one block by ESP. Depending on the cipher, the generation of this IV can often be prenegotiated by the peers and thus is reducible or even dispensable. This setup is called Implicit IV [21]. It needs to be supported by the SAD, an IKE extension and the cipher itself. But no changes would need to be done to the ESP processing. Using this method 8 octets per packet could be saved for the most common ciphers e.g. AES-GCM, AES-CCM, AES-CTR and ChaCha20-Poly1305.

IPsec and IKEv2 for the Contiki Operating System

This thesis did an extensive job both in documenting and in integrating IPsec as well as IKEv2 into the Contiki operating system. [15] A lot of IoT related problems regarding IKEv2 and ESP surfaced. Though the thesis conclusion for the use of IPsec in IoT is pessimistic, I think with the recent developments in ESP extensions a bigger picture forms wherein unique features can be achieved by the use of IPsec.

Minimal G-IKEv2 Implementation for RIOT OS

This thesis dives into the real life roadblocks and performance results of applying 'Group Key Management using IKEv2' (G-IKEv2) within RIOT OS and deploying it on limited embedded hardware and IoT networks. [14]

Group Key Management with Strongswan

The GPL licensed software Strongswan [2], generalizes and packets the rich feature set of Internet Key Exchange services IKEv1 and IKEv2 into one suit, easing the integration and administration. It supports multiple platforms while keeping a low footprint, even being deployed on smartcards. The thesis 'Group Key Management with Strongswan' successfully implemented a minimal G-IKEv2 into Strongswan. [11]

4.2 ESP Header Compression (EHC)

ESP encrypted communication is desirable in distributed IoT networking, even though the overhead of an IPv6 packet itself is already very big. ESP protection further eats away on this limited space, possibly triggering the need to generate a second IPv6 packet to transfer the information. By reducing the bandwidth one would noticeably improve the battery lifetime of low power devices and deliver ESP to more constrained systems. But even nodes with more generous power support and high traffic rates, like encrypted digital mobile radio, in use by military and emergency forces, would no doubt benefit from the compression and thus enhanced life time or reduced battery size.

Header compression is important in an IoT context to reduce packet size. When IPsec ESP is deployed with 6LoWPAN, the header compression is no longer effective. Since 6LoWPAN resides in a lower OSI layer, many of the compressable fields are encrypted by that point. Therefore a header compression protocol integrated into ESP is needed to compress the respective headers before encapsulation and encryption. This is addressed by the 'in development' standard of ESP Header Compression (EHC). [20] This concept utilizes the modularity, extendability and the session based communication of IPsec to enable a header compression without breaking any compatibility of the IPv6 protocol. A good overview and a minimal implementation can be found in the 2014 thesis "Diet-ESP: Applying IP Layer Security in Constrained Environments". [13] It should be noted, that the additional compression of the IPv6 header by 6LoWPAN is possible and feasible, but not discussed any further in the scope of this thesis.

EHC Principle

The basis of an established IPsec communication is a set of connection-specific session information saved in a database by both peers. It is negotiated prior to sending the traffic. IPsec resides in the kernel's IP stack. It is not triggered externally but has to reliably identify the traffic that should to be protected or discarded. It utilizes the concept of Traffic Selectors (TS) to precisely identify a connection. This information is also stored in a separate database. While this enables precise filtering - down to layer 4 protocol ports - and selective encryption, many fields in the ESP header and the following headers are becoming redundant. On the one hand, this comes from the fact that every ESP packet and the according session, are identified by their unique identifier called the Security Payload Identifier (SPI). On the other hand many fields are expendable information, considering their encapsulated fields. ESP can provide integrity for the packet and fits every SPI with its individual message counters. This renders the bulk of counters and checksums residing within the ESP payload headers redundant, since they can simply and securely be recalculated after verification and decryption. Additionally the padding, added in many protocols to provide for proper IPv6 block alignment, is deterministic and can be removed while being encrypted within the ESP packet. Lastly, many header fields, like certain lengths, can be pulled from lower layers or be recalculated from lower layer sizes. The concept of EHC is integrated directly into the IP stack alongside the ESP packet generation and aims to remove these redundancies prior to encrypting. On reception the original packet will be restored without alteration after decryption of the ESP packet. Thus the compression is fully concealed from the communicating application layers. There are a couple of methods to reduce this superfluous information. Be it by recalculation on reception, the simple dropping of redundancy, or by reduction of larger

counters and identifiers, with predictable range, to their Least Significant Bytes (LSB). [20]

Rule Negotiation

While it may seem obvious for the reduction of some header fields, the communicating peers need to agree on which and how the individual fields will be handled. A general negotiation of protocol versions and compression capabilities is also evident. This initial negotiation is achieved by utilizing the possibility to deploy extensions to the key management protocol of IPsec. The whole IPsec system is designed in a spirit of great modularity, loosely defined structures and a focus on extensibility. Therefore, two peers supporting the same extension can probe another on compatibility and then negotiate any kind of specialized information alongside the regular key negotiations. The underlying ESP implementation and the IPsec databases then just need to provide support for the additional information inside of the database entries. Of course it would be tedious to propose and negotiate the compression of every single header field. Therefore a set of general strategies are predefined and can be proposed. These are stored inside the EHC Database.

EHC Database and Diet-ESP

Certain scenarios for ESP deployment share certain possibilities for header compression. Thus providing composed EHC rule sets, also called strategies, not only reduces the amount of negotiation needed but also makes EHC and its setup much more approachable by providing an already optimized and matured general setup. One such strategy is Diet-ESP. It is mainly aimed at low frequency, low bandwidth IoT devices. It incorporates UDP and TCP compression, choosing the shortest practical LSB lengths assuming a rather low amount of packets. Using this strategy results, the fields that need to be negotiated prior to ESP communication are reduced from an initial 44 down to only 9 fields. [20]

The efficiency of Diet-ESP compression can be demonstrated in an impressive fringe example. By reducing all unnecessary fields and omitting padding a packet can be generated in a way, that the savings are greater than the overhead created by the ESP header and the padding needed for the cipher. Therefore deploying ESP with Diet-ESP can result in a lower message size, reducing the packets needed from two down to one and thus nearly halving the power consumption for a single message compared to the unprotected traffic. Although this example surely is exaggerated, it shows the immense potential of EHC to secure IoT traffic, while minimizing the bandwidth at the same time.

5 Design

This section takes on to the design phase of the thesis. Considering all that is mentioned prior, I designed the functionalities upon the addressed principles and the specific requirements proposed. I also want to mention that the full functionality of IPsec extends way over the minimal scope this design covers. Some aspects of the design did not make it into the final implementation for different reasons. Some proved to be impractical, some showed up to be ambiguous in implementation and therefore where left open ended. A simplified overview over the design and the interconnection of the separate methods can be seen in figure 7.1.

The design of a minimal ESP implementation called for a certain set of features, that are interlinked but relatively independent and where thus modularized. The result of this software architecture can be found in figure 5.3. I wanted the code to be minimally invasive in regards to existing RIOT OS source code in order to ensure modularity and also hardness against accidental side effects from changes to other parts of the source of the IP stack.

5.1 IPsec packet filtering

To protect the system according to the SPD ruleset, all incoming and outgoing IP traffic needs to be filtered and ESP encapsulation needs to be reliably generated and processed. I wanted the design to change as little of the existing kernel code as possible. The first approach was to utilize the netapi message queue of the GNRC and make the code fully modular, but I soon realized it was not that easy. On the receiving side, I would break the idioms of the netapi abstraction if I deleted or modified packets that should be discarded and the general setup would have been rather fragile, since I would have needed to rely on thread priorities. On the other end, the momentary IPv6 implementation does not utilize the netapi message bus to communicate with the interface but rather builds the interface header itself and sends the packet right away to the appointed interface. So instead of rewriting parts of the IPv6 flow we decided to set some submodule hooks and break out of regular IPv6 handling if needed. This approach is less invasive and more robust, since all IPsec hooks reside in the same file and are easy to notice and not obfuscated by some weird priority handling on the message bus. The only thing that branches of the regular packet flow outside of the IPv6 routine is the ESP header processing which is handled within IPv6's EXT header handling routines.

While the information for traffic analysis mainly come from the IP header fields, to access the port values of packet we have to check the corresponding labels in the transport layer header. This was a issue to take a little care of, since I wanted to make sure not to change the original packet flow for any packet that was just bypassed. Since the GNRC way of handling a packet lets protocol every entity mark their own headers, without handing them back before further processing, we decided to just have a binary peek into the following packet. And since the only protocols with ports I was interested in are UDP and TCP, I

were able to keep it short. The ESP header should be the last header before any payload, thus it can be assumed that the transport layer begins just behind the EXT header.

The versatility of ESP offering Transport, Tunnel and Multicast traffic needed to be joined with my goal of a minimal impact on existing code. Especially in the case of positioning the gateway routines, any redundancy needed to be avoided. Since routed, self originated and self encapsulated traffic all take different ways through the system, a trade off was accepted for positioning the gate keeping routines not at the easiest places but the most versatile ones. For now I omitted Multicast traffic from this implementation, since I want to neglect manual keying in DietESP and at the moment the only way to multicast ESP traffic, according to RFC standards, is by manual keying. Thus this additions to the code should be made by the first one implementing a multicast setup and having a clear scenario and definitions of the routines involved.

An important note in implementing Self Encapsulated Tunnel Traffic opposed to regular tunnel traffic is not only the check for similarity on receiving, but also that one has to change the sending process. The packet is not handed to the IP processing again after generation of the new payload like it is done with foreign tunnelled traffic, but the packet is forged by the ESP routines, equipped with the new payload and then sent out the same way traffic in Transport Mode would.

Traffic Flow

The seemingly simple IPsec behaviour can become very complex in structure and therefore also the realization in code. A good example for this are incoming ESP packets. This PROTECTED traffic is not analysed for its TS. Packets identified to be ESP encapsulated get passed down the IP stack, where they later enter EXT header demultiplexing and there the ESP handling is initiated. After finding an SAD entry fitting the SPI of the packet, the packet is then decrypted and decapsulated. Now the restored packets TS is analysed and checked against the SPD rule set. This is because a SA can be shared by multiple SPD rules. A quick example will help to illustrate that. Think an SA, shared by multiple SPD rules to protect all traffic between two peers, but one SPD rule states to DISCARD a specific TS of that traffic for example, all TCP packets.

Another example is that on regular ESP tunnelling the processed packet, regardless being incoming or outgoing, will be send to the beginning of IPv6 processing again to either generate the new IP header or "receive" the decapsulated IP packet. Self encapsulated packets on the other hand will be sent directly to the interface, since their active IP header is just altered in mutable header fields accounting for the changed size and next header. A full overview over the generalized traffic flow, relevant for the IPsec implementation can be found in figure 5.1

5.2 IPsec Databases

The realization of the IPsec databases gives a lot of room for individual solutions and optimizations. On the maximal compression level, all caches and databases could be reduced to one single array. Some small scale implementations of IPsec intended for IoT save a lot of complexity on the Databases. I did not deploy these simplifications since the standards definitions enable maximal versatility in network configuration. Especially in IoT networks

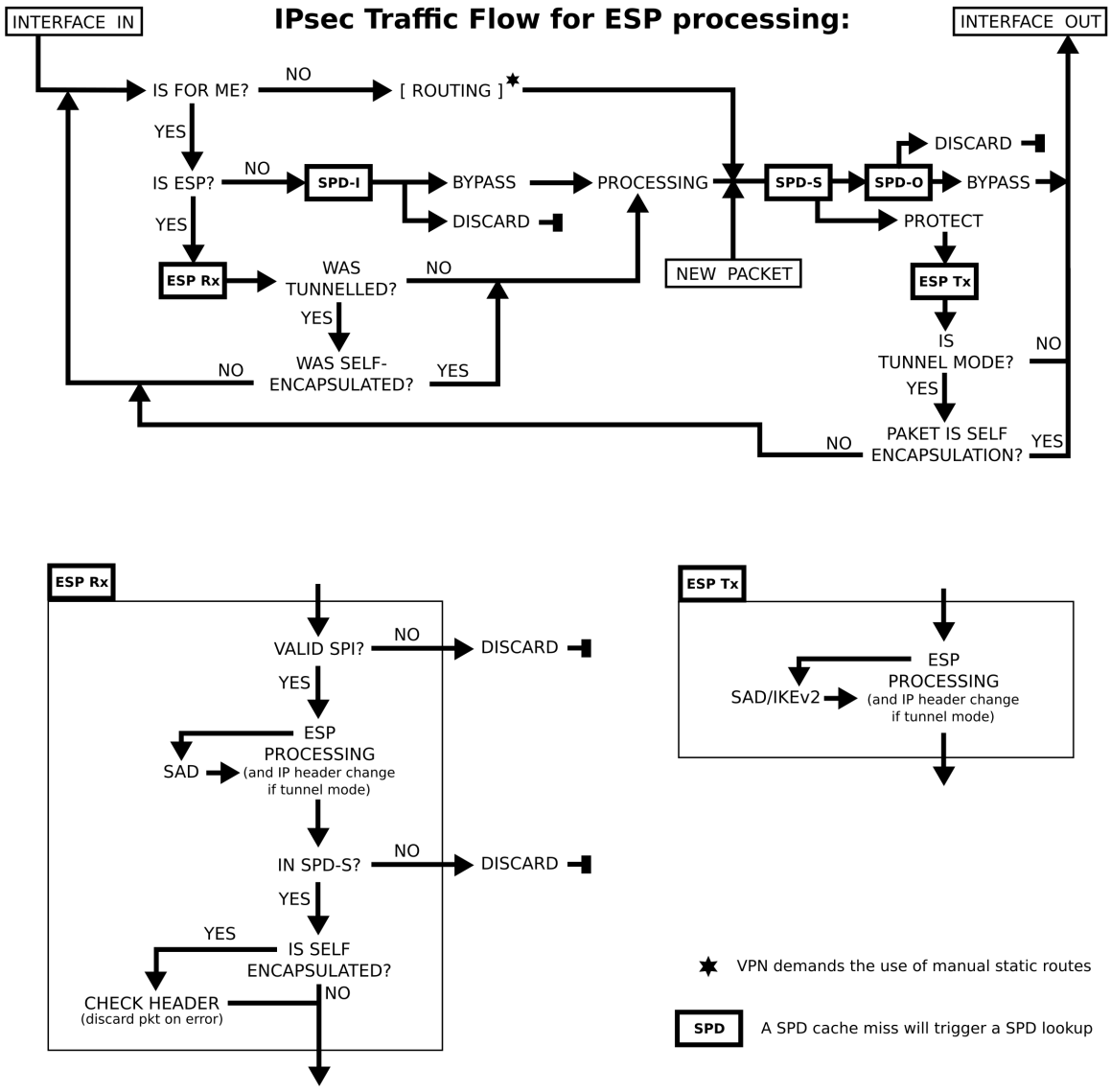

Figure 5.1: ESP packet processing flow chart

a lot of rather bold networking choices can be expected. I also can not easily foresee the scope of more complex key and SA management for example in the form of G-IKEv2. [29]

Though some features have been omitted, beginning with SPD mutability and the resulting cascading. Since the ruleset is not managed no entry will need change or removal.

In the design I will also include the SPD-S inside of SPD-O, since I did not encounter the need to treat them separately throughout development. Database checks for incoming protected ESP traffic will simply be performed on the SPD, since the specific SA is already known. The only caveat to this would be a legitimate but false SPI from a trusted connection. But then the authentication would simply fail as intended for any erroneous traffic.

The SPD databases and their indexes work by 'First In Last Out' principle and are written in static memory. The SAD is an unordered database identifiable by its name or SPI

fields. SPD cache entries do not need to be ever removed, since the SPD rules are immutable and compiled into the code. For testing and debugging purposes, a tool was deployed to circumvent this static restriction. See 'Managed SPD Setup' in section 6.1

In the scenario of static memory spaces, a revoked SA should get their identifiers set to zeros to avoid accidental reuse of the same memory block. Any SPD entry using this SPI also needs to get its SPI field zeroed, keeping the cache entry but triggering a renegotiation the next time it is called upon.

Key Engine

The database management is called the Key Engine. It purpose is to manage all dynamic activity in the databases, but not inside its entries contents. It generally abstracts the access, providing getter and setter methods and ensures the sanity of the interlinked databases. Regular tasks like the triggering of SA negotiations on new SPD-S cache entries and also the renegotiation of an SA are initiated by it.

There are many veritable scenarios to model the entries of the databases after and it is highly likely that a specialized use case will make changes to the databases. Therefore the Key Engine also works as an abstraction layer. It will use setter and getter methods to hand out the entries as struct elements. That way the content is addressed by named identifiers in code and thus the entries are easily extendable.

Key Management

The Key Management thread takes care of the negotiation of cipher keys and session details. It is initially intended to negotiate using IKEv2, but should be able to support any IKE protocol. It provides an abstraction layer between the IKE protocols, whom are part of the Key Management engine, and their interactions with the Key Engine. It should be deployed as a threaded process to be able to act as a netapi message receiver for the IKE protocol.

Database memory management

The implementation needs a persistent memory location to store and manage the IPsec databases. Since only a thread can keep persistent variables, I attribute the ownership of the databases to the Key Engine thread, hence it already takes care of the database management.

Traditionally embedded systems lack dynamic memory management for good reasons. Their scope is often clearly defined at compile time and handing static memory to the

different modules, one ends up with a very predictable and reliable system. The downsides to this are of course, that every memory space is single purpose only. Therefore fixed variable and buffer size, decided upon at compile time are needed.

That is fine as long as we have a predictable environment. But one of the strong sides of IPsec is its dynamic key management and the ability to adapt to a changing environments, provided only with a limited rule set. In those scenarios the databases, especially the SAD can be very volatile. Therefore it would be desirable to have a more dynamic memory management. The TLSF [3] system enables us to reserve a block of memory for all databases at once and have real dynamic memory management in that restricted memory area. An alternative solution would be to write an independent module providing our own dynamic allocation on a fixed bit of memory. But any such endeavour should happen outside of the Key Engine thread.

Thus I did not define a fixed solution inside the design but left it to future implementations and the use cases of their deployment to experiment with the performance of different memory solutions.

Logging

For investigative and security reasons and also to better fight attacks like Denial of Service, logging of discarded or faulty ESP traffic and all unexpected behaviour is strongly encouraged by the specification. But since most IoT device setups are compiled and thus aren't administrated, logging outside of a development situation is in generally useless overhead and a needless source of system failure. Therefore all logging routines are disabled outside of a debugging environment.

Since there is no logging, ICMP Error Message [26] generation and handling were also omitted from the design.

5.3 Inter Process Communication

The PF_KEY API was initially intended as the abstraction layer for the communication between Key Engine and Key Management. Despite its uses, implementing this standard would only make sense when being fully compliant to it. Since PF_KEY is overly complex for the use with ESP in IoT, I stopped the integration giving the already limited resources of constrained nodes. The outlines of this system are still exiting in the implementation. I made sure that if a follow up evaluation decides to not continue to regard PF_KEY, the code can be removed without serious dependencies.

The exchange of constructed packets is done by netapi messages. It is a bus system allowing to send a pktsnip to all interested parties. This is freeing the sending thread and threading is needed to receive this message. This is used to relay outgoing ESP traffic from the IPv6 to the IPsec thread in order to unblock the IPv6 thread. After all, outgoing ESP potentially needs negotiation.

For more direct communication there is also the pid (Process Identifier) based message system. On outgoing ESP, this is used to send the generated IP pktsnip directly to the according Interface. Just as IPv6 would do. This is redundant code, but the relevant context to send the packet can not be reestablished in the IPv6 thread without making major changes to the IPv6 thread, or without blocking it.

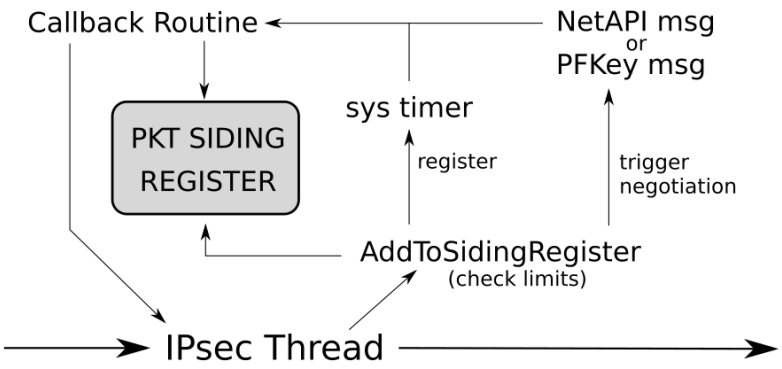

Figure 5.2: Siding Register Flow

5.4 Threads

Any additional thread is a negative resource impact, even the limited threads RIOT OS provides. Therefore I limited the design to three threads. They can be found in figure 5.3. A reduction to two threads would be possible by omitting the feature that an ESP packet, waiting for negotiation, is kept in memory.

First is a thread for IPsec, enabling it to receive outgoing netapi ESP packets. This ESP Tx traffic, sent by a netapi message, allows for holding the ESP packet, while freeing the IPsec to enable negotiation and other IP traffic.

To enable this functionality without the use of a thread, the concept of a siding register was minted, holding the IPsec packet and its context. Given the low traffic volume nature of constrained nodes and the rarity of SA negotiations, I dropped it from the design. Assuming higher traffic demands, this concept could become viable, thus the general concept of the siding register can be seen in figure 5.2

The second thread resides in the Key Engine module and holds the persistent memory for the IPsec databases. It also enables us to schedule checks on the SA lifetimes, triggering renegotiation when needed.

The third thread is needed for the Key Management to be able receive the netapi messages for the according IKE protocols. It also serves persistent memory to be able to have a stateful communication, needed for the Diffie Hellman exchange.

5.5 Packet Generation and Alteration

IP packets filtered for PROTECTion will be handled by the routines residing in 'gnrc_ipsec.c'. They will be filtered by conditional statements to end up at the right routine and with the correct context. They will then be split, altered and reassembled by pointer based memory modification. The information needed to process the packets are acquired by requesting the according database entries from the Key Engine. They then get handed a pointer to the entries memory location in the form of a struct.

It is important to note, that any alteration that would need to be made to these entries, e.g. incrementing a packet number, needs to be triggered by calls to the Key Engine. This will keep the database abstraction and also centralizes any modification of the database or

its entries in the Key Engine module.

ESP and IP packets will be automatically padded in accordance with the standards. This will need alterations to enable padding reduction by DietESP.

5.6 Cryptography

Since the driving force behind this thesis are protocols and processes that are still in development, it makes sense to focus on modern IoT devices and technologies to come, rather than backwards compatibility for old standards. ESP and IPsec are already quite resource hungry, thus emphasis on modern and power efficient ciphers should be made. Still, the design enables the integration of any ciphers supported by IPsec.

The cipher of choice would be chacha20poly1305 which was recently added to RIOT OS. It is a combined cipher, thus encrypting and hashing at the same time and with the same key. Among the strength of this counter based block cipher is its processing efficiency, the need for limited padding and no need for randomness of the nonce. [22] Because of this its often seen as the future standard algorithm for efficient and secure encryption.

The main goal was to create the backbone of IPsec to work and provide all relevant internal connections and build or at least layout all relevant modules. Therefore the factual integration of a cipher had little priority and would just have been a show piece. But handing the paket to a cipher to encrypt, authenticate and add the relevant fields to ESP is an essential part of the ESP header generation process. Therefore I integrated a mockup hash and cipher referred to by 'MOCK' in code and in this thesis. This mockup cipher uses and generates all fields needed for the ESP packet generation, but leaves the payload unencrypted and does not calculate a real hash. Thus it is of course in no way protecting any traffic. I could have made the MOCK to deploy a naive obfuscation, but keeping the to-be-encrypted parts of the ESP datagram in plaintext, gave an easy way to verify the correct packing for the ESP payload. The correct placement of the Integrity Controll Vector (ICV) and the Initialization Vector (IV) could just as easily be verified as the correct processing of Tunnel Mode traffic. That way I could verify correctly generated IPsec ESP packets, even though I only was able to test against my own implementation.

5.7 IPv6 packet fragmentation

At the time of design and implementation of this thesis, RIOT OS did not support general IPv6 packet fragmentation outside the scope of 6LoWPAN. Thus any need for fragmentation of application data is to be managed by the application layer. It would therefore be nice to have a method to request for pre-calculation of the available payload space including encryption, but since fragmentation is already in development for RIOT, it makes sense to await this and then properly merge it with the IPsec design. Considering that ESP and Diet-ESP encapsulation change the need for fragmentation, it is obvious, that one would need to integrate it into the fragmentation process. Maybe even side the fragmentation and reassembly of ESP packets into an specialized method. Integration and functionality of the design regarding RIOT OS's 6LoWPAN implementation was not considered in this thesis.

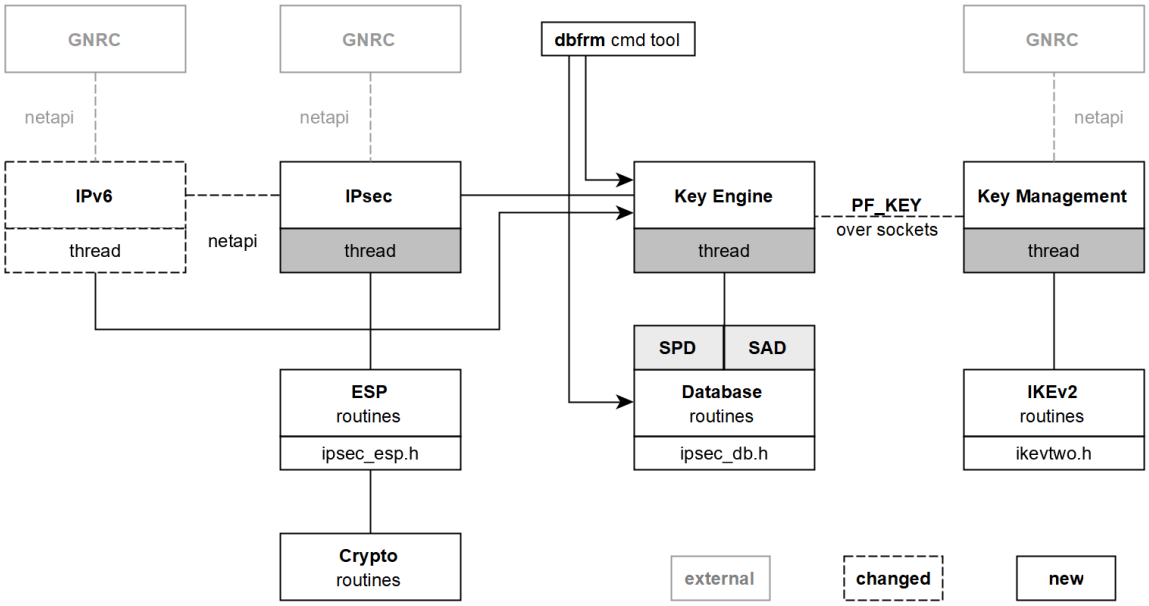

Figure 5.3: Software Architecture and Control Flow

6 Implementation

The Implementation section will give an overview over the manifestation and realization of the design into the RIOT OS network stack. Overall, a lot of ground was covered and only the mainly ambiguous topics where left to be considered in light of more tangible use cases. A simplified overview of successfully and partially implemented features can be found in figure 7.1. The goal of the implementation is to lay a groundwork for future implementations of EHC and especially DietESP. Therefore I explored the RIOT OS code, evaluated where to engage with the existing code and created a minimal ESP implementation with limitations and a feature set in focus of a future EHC deployment. I deviated from the ESP standards only in omitting complexity that will not be used in EHC deployment. For example Authentication Header and older encryption standards. In the end the implementation will most likely communicate with other EHC implementations and thus, full interoperability with other ESP implementations is of lower priority. Still I tried my best to abide to the details, defined in numerous RFC papers. The scope of the work changed multiple times throughout the process and thus there are some looser ends to the code. I did not scrap the parts that could be useful depending on the environment it is deployed in, or where its use could only be evaluated based on the requirements of a specific use case. Overall the code is beyond the stage of an POC and should be possible to merge into the kernel as soon as Key Management and encryption is added and minor changes are deployed after being decided upon. Although it was often the harder road to avoid goto statements, I still designed all recursions and control flows without them for the sake of code comprehensiveness. For optimization I think, at the time of writing, one can rely on modern compiler optimization.

6.1 Realization

Besides all good design intentions, a realization of a design has to deal with a lot of roadblocks, especially if it is intended to integrate into an existing code base without changing it to much. From very low level integration choices like correct variable selection to restructuring of existing code. I also tried to comply to the coding conventions for C in general and for RIOT OS in specific. Since we are intending to merge this code into the RIOT code base and my work should lay ground for future development on this topic, effort was spent to structure the code in a most readable way that is easy to refactor. This for example includes the omission of goto statements in favour for nested conditional structures and function definitions. Since the latter is more verbose, its functionality is easier to grasp and the flow of execution is better contained.

Initialization

The process of initializing the implementation comes in the shape of a RIOT OS example project utilizing a simple UDP packet exchange to test and demonstrate certain features. In Appendix A all details are listed for how to set up the Linux environment for testing in

RIOT OS native. Example values and 'dbfrm' arguments for the test of different scenarios are also provided.

Integration into the existing RIOT code base

The integration into exiting code works by the use of submodules. These can be activated in the Make file and the relevant code is then encapsulated by conditional compiler directives. Some parts of RIOT OS kernel source were just roughed out and were not used anywhere in the code. These parts were not fully functional and documented poorly, especially some parts of EXT header handling. I added documentation and assured on every change, that nothing in the kernel or the examples depends on the modified sections. Additional to the EXT handling showed in listing 6.1, EXT needed to be "activated" in the EXT header 'demux' step of gnrc_ipv6_ext.c

Listing 6.1: ESP packet detection and rerouting to ESP processing integrated in IPv6 EXT header handling of RIOT OS(2019.01) : gnrc_ipv6_ext.c

```
static gnrc_pktsnip_t *_demux(gnrc_pktsnip_t *pkt, unsigned protnum)

...
  switch (protnum) {

  ...

    case PROTNUM_IPV6_EXT_ESP:
#ifdef MODULE_GNRC_IPV6_IPSEC
              pkt = esp_header_process(pkt, protnum);
              if( pkt == NULL) {
                  DEBUG("ipv6_ext: Rx esp header processing failed or pkt was
    consumed by ext handling\n");
                  gnrc_pktbuf_release(pkt);
                  return NULL;
              }
              break;
#endif

...

    return pkt;
}
```

The existing IPv6 code order and some conditionals had to be changed to be able to reliably filter traffic. I was always assured, through analysis as well as testing, that besides all code changes and additions, the original code continues to run as intended. After that I added the conditional hooks that control the traffic filtering and protection into gnrc_ipv6.c.

Listing 6.2: IPsec Traffic filtering for incoming IPv6 traffic, integrated in IPv6 handling of RIOT OS(2019.01) : gnrc_ipv6.c

```
static void _receive(gnrc_pktsnip_t *pkt)
{

...
```

```
#ifdef MODULE_GNRC_IPV6_IPSEC
    ipsec_ts_t ts;
    if(ipsec_ts_from_pkt(pkt, &ts, GNRC_IPSEC_RCV) == NULL) {
        DPRINT("ipv6_ipsec: Rx: couldn't create traffic selector\n");
        gnrc_pktbuf_release_error(pkt, EPROTO);
        return;
    }
    switch (ipsec_get_filter_rule(GNRC_IPSEC_RCV, &ts)) {
        /* PROTECTED packets are allready handled in ext processing */
        case GNRC_IPSEC_F_BYPASS:
            DPRINT("ipv6_ipsec: Rx BYPASS\n");
            break;
        case GNRC_IPSEC_F_PROTECT:
            /* EXT header handling will process/reject ESP header */
            DPRINT("ipv6_ipsec: Rx PROTECT\n");
            break;
        case GNRC_IPSEC_F_DISCARD:
            DPRINT("ipv6_ipsec: Rx DISCARD\n");
            gnrc_pktbuf_release(pkt);
            return;
            break;
        case GNRC_IPSEC_F_ERR:
            DPRINT("ipv6_ipsec: Rx: SPD check returned no result. Discarding
   packet.\n");
            gnrc_pktbuf_release(pkt);
            return;
            break;
    }
#endif

...

  _demux(netif, pkt, first_nh);
}
```

Listing 6.3: IPsec Traffic filtering for outgoing IPv6 traffic, integrated in IPv6 handling of RIOT OS(2019.01) : gnrc_ipv6.c

```
static void _send_to_iface(gnrc_netif_t *netif, gnrc_pktsnip_t *pkt)
{

...

#ifdef MODULE_GNRC_IPV6_IPSEC
    ipsec_ts_t ts;
    if(ipsec_ts_from_pkt(pkt, &ts, GNRC_IPSEC_SND) == NULL) {
        DPRINT("ipv6_ipsec: couldn't create traffic selector. Release pkt\n")
    ;
        gnrc_pktbuf_release_error(pkt, EPROTO);
        return;
    }
    switch (ipsec_get_filter_rule(GNRC_IPSEC_SND, &ts)) {
        case GNRC_IPSEC_F_BYPASS:
            DPRINT("ipv6_ipsec: SND BYPASS\n");
            break;
        case GNRC_IPSEC_F_PROTECT:
```

```
            DPRINT("ipv6_ipsec: SND PROTECT\n");
            if (!gnrc_netapi_dispatch_send(GNRC_NETTYPE_IPV6_EXT_ESP,
    GNRC_NETREG_DEMUX_CTX_ALL, pkt)) {
                DPRINT("ipsec: no IPsec thread found\n");
                gnrc_pktbuf_release(pkt);
            }
            return;
        case GNRC_IPSEC_F_DISCARD:
            DPRINT("ipv6_ipsec: SND DISCARD\n");
            gnrc_pktbuf_release(pkt);
            return;
        case GNRC_IPSEC_F_ERR:
            DPRINT("ipv6_ipsec: GNRC_IPSEC_F_ERR\n");
            gnrc_pktbuf_release(pkt);
            return;
            break;
    }
#endif

...

}
```

Loopback traffic in RIOT OS does not go through all the hassle that regular IPv6 packet must go and thus is sent directly to the interface it came from with a specialized routine. It is thus intercepted, and filtered there, too.

Listing 6.4: IPsec Traffic filtering for loopback traffic, integrated in IPv6 handling of RIOT OS(2019.01) : gnrc_ipv6.c

```
static void _send_to_self(gnrc_pktsnip_t *pkt, bool prep_hdr,
                          gnrc_netif_t *netif)
{

...

#ifdef MODULE_GNRC_IPV6_IPSEC
    ipsec_ts_t ts;
    if(ipsec_ts_from_pkt(pkt, &ts, GNRC_IPSEC_SND) == NULL) {
        DPRINT("ipv6_ipsec: couldn't create traffic selector\n");
        gnrc_pktbuf_release_error(pkt, EPROTO);
        return;
    }
    switch (ipsec_get_filter_rule(GNRC_IPSEC_SND, &ts)) {
        case GNRC_IPSEC_F_BYPASS:
            DPRINT("ipv6_ipsec: SND SELF BYPASS\n");
            break;
        default:
            DPRINT("ipv6_ipsec: SND SELF DISCARD\n");
            gnrc_pktbuf_release(pkt);
            return;
    }
#endif

...

}
```

ESP Header Processing

The header processing of ESP headers relies on named elements and conversion functions, wherever possible. This ensures for example that the implementation is independent of the endianness of the host system it is deployed on. To extract the inner header information of unmarked packets, while not disturbing the packet flow, I had to use manual pointer arithmetic. This way I obtained the the information that I needed, to filter the packet prior to processing. I enabled all desired IPsec functionality and behaviour, laid out in the design section.

Key Engine

The Key Engine features the intended level of abstraction, serving the database information to the requesting entities. It also reliably creates the dynamic parts of the database, based upon the SPD rules. Features that did not get integrated are accounted for and laid out in documentation and comments.

Managed SPD Setup

To counter the lack of dynamic key management I created the 'dbfrm' command line tool. This tool breaks certain abstractions and is therefore only intended for kernel development use. It enables to inject SA and SPD-Cache data into the IPsec system. Avoiding SPD checks it directly creates cache entries. So for testing purposes, no changes need to be made to the SPD ruleset. Usage details can be obtained by execution of 'dbfrm -h' or reading the 'README.md' file of the RIOT OS example for the implementation 'gnrc_networking_ipsec'. Because of its low level functionality it easily breaks when changes are made to the definition of the database structs.

6.2 Documentation

I tried to give helpful code commentary, documentation and notes. In addition to a readable coding layout, I used meaningful names and declarations. Wherever practical I took named ENUMS for selectors. That way the code is easier to comprehend. It was often not easy to prevent opaque code or stashed side effects, considering the interlaced nature of IPsec. But it will hopefully aid in further development and ease the integration into open source projects.

7 Evaluation

In this chapter of the thesis I will present my evaluation of the implementation, the methods and results of testing. I will also present a few numbers on the implementation and will explain some of the road blocks and how they were detected and overcome. Furthermore I will give some estimations on general and minimal resource consumption when using plain ESP.

7.1 Design Completeness

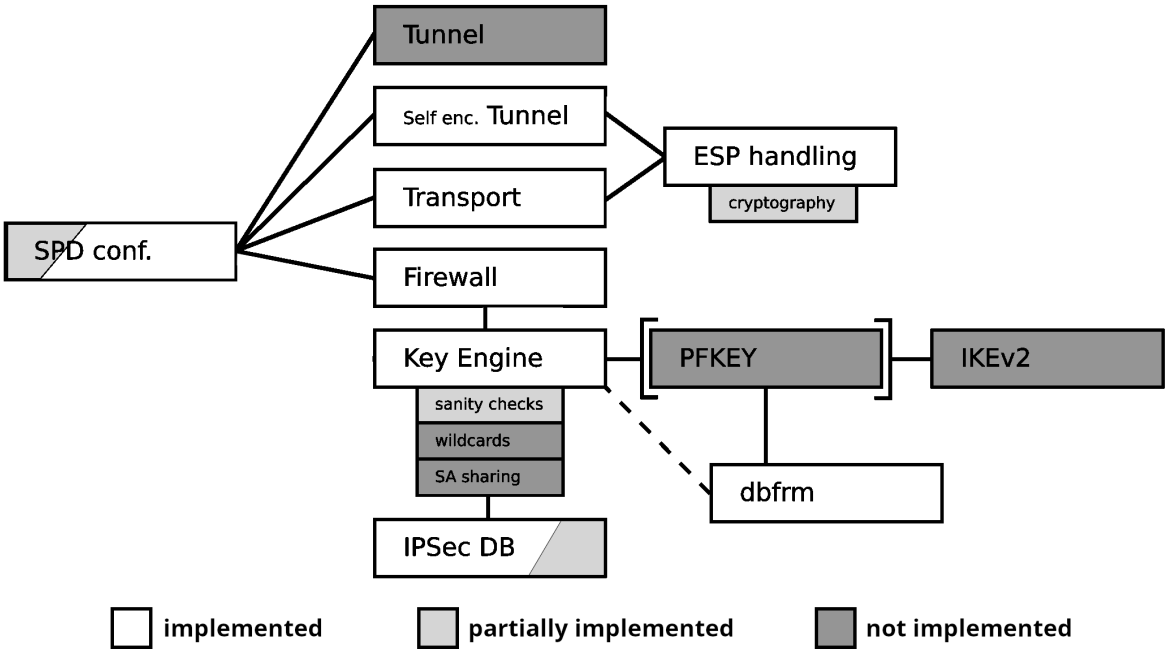

Figure 7.1: Realized elements of the implementations design

The illustration in figure 7.1 gives a rough overview about the parts of the Design that made it into the implementation. All sections that are rendered in white are implemented and function in an example scenario as well as probably in most of scenarios the standards have in mind. Recalling figure 5.1 I kept the implementation close to RFC standards regarding protocols and scenarios, and followed the open source coding guidelines for RIOT OS. I also laid a foundation for the parts that are not implemented yet. So in the following I will only elaborate on the parts of the design that were not or only partially implemented.

- The PF_KEY messaging behaviour got put aside for various reasons already explained in the design section 5.3. But the need for a bit more general and powerful message

system remains, especially if one would intend to deploy the dfrm tool beyond the scope of implementation testing.

- IKEv2 is a vital part of IPsec but also a very demanding to implement, thus it is left open for future work on this topic.
- The standard ESP Tunnel Mode is only deployed when a device is used as a router. We did not sense this as a very viable option for RIOT OS in general and as it diverges noticeably from Self Encapsulated Tunnel Traffic, it fell due to the timeline. Although specialized packet flow and specific methods will need to be deployed, it should be rather quick to be implemented into the existing structure if needed. The source holds several notes hinting to the right spots.
- The Key Engine is a bit cut short in features beyond the basics. The support for wildcards and ranges in the SPD rules is missing, making rule setting a lot more verbose. Automated sanity checks of lifetime timers were dropped but are probably niche to be deployed anyway. Another relevant aspect I do not support is the sharing of SAD entries. This should and will be implemented as soon as the Key Management is brought in and Keys are linked with SPD-S cache entries dynamically.
- The SPD rule configuration is working just fine, but the rule set is still residing in the kernel code. That is of course something to be changed. A parser could be used to read from a file or a mandatory class could be utilized in the project folder. I favour the file solution, instead of adding another big piece of code in the form of a parser, but I the kernel code depending on a project file is not viable. I did not find an easy solution for this and it works reliably for testing the implementation.
- I already mentioned the lack of proper cryptography, but I am sure that adding the cipher of choice will be a quick thing, as soon as IKE functionality is had. Bending and twisting around with manual keying or deprecated ciphers when the result would be just as insecure as using the MOCK cipher right away, did not seem to a worthwhile effort.
- Last but not least I come to IPsec Database memory positioning. While the core functionality and hierarchies of the database is fleshed out well enough, its underlying memory management is not mature. I think the problems regarding the dynamic nature of the SAD and the SPD Caches where laid out sufficiently in 5.2. This is thus to be decided by the way this code will be put to use in future work.

7.2 Testing and Performance

The lack of dynamic key negotiation hindered me to easily test against another implementation of ESP without the need to modify the key management of the counterpart. So for testing and debugging purposes I have set up a Tun/Tap system of virtual interface devices. This is explained in more detail in the 'Initialization' section 6.1. Since the generation and demultiplexing of ESP traffic is handled by two different routines of my implementation, an error in only one of them would result in a failed connection. But to test the correctness of ESP generation beyond the scope of testing against my own implementation, I used the

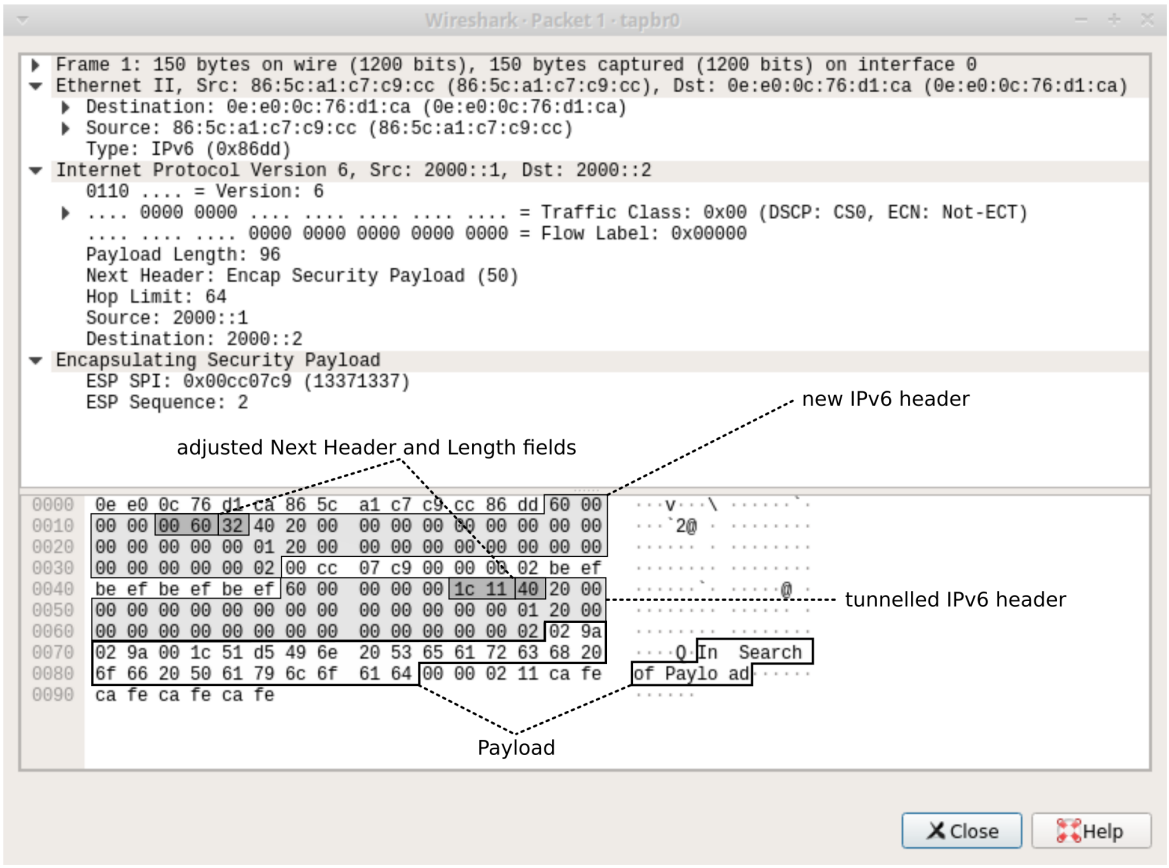

Figure 7.2: ESP Tunnel Mode packet dump - Inner and outer IPv6 packet highlighted

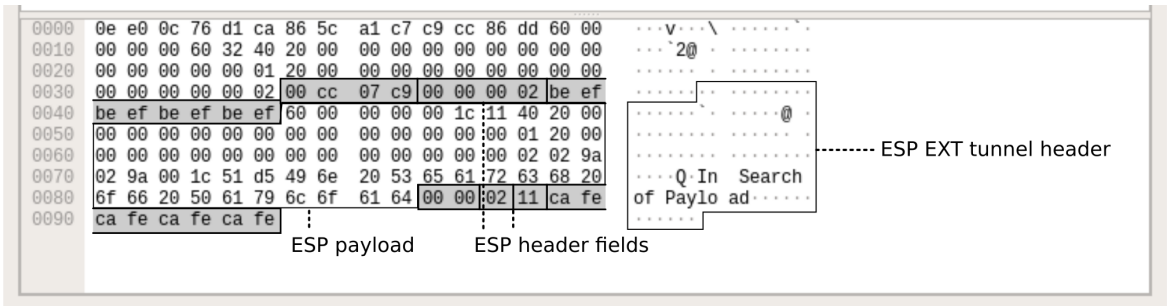

Figure 7.3: ESP Tunnel Mode packet dump - ESP EXT packet highlighted

open source GPL licensed software Wireshark [4]. It not only gave me the ability to read the dump of the traffic being sent over the tap bridge, but Wireshark comes with protocol detection itself and can be seen a minimal ESP detection. So despite not being able to test against a foreign implementation, I could check if the ESP and IP headers were modified and generated correctly and also inspect the bit values of the packet. The MOCK cipher helped a great deal when analysing the packets. Especially with deep packet analysis of Tunnel Mode ESP traffic as can be observed in the figures 7.2 and 7.3.

With a clear view on the encapsulated packet, I could ensure not only the correct placement and adjustment of the replicated IP packet, but also correct use of SA values and their placement at the positions designed for the IV and ICV. The same was done for regular Tunnel Mode traffic with the same results. This shows that the implementation is sending out proper ESP packets, every other ESP implementation with the right SA installed should be able to process.

7.3 Development

In the development process, packet dumping was often used to aid debugging. Since the analysing, restructuring, building and processing of ESP and IP packets in C is all done by manual memory editing and prone to initial errors and mishaps. In this context the endianness often created roadblocks, though strict data types and the internal conversion routines, provided by RIOT, allowed for development that is independent of the hosts endianness.

One major task was to grasp the full flow of the packets, the database and the caches. While developing I often returned to the design and tweaked and refined my understanding of IPsec and only in a hands on approach I could see the pieces fall in place and all making sense. Refer to 5.1 for the full result of my exploration.

7.4 Resource impact

Modern devices intended to run a IPv6 stack are already very capable. Regarding processing power as well as memory size. The encryption will probably take the major bite out of the performance and IKEv2 will have its toll, too. Thus the processing impact of the ESP backbone will be comparatively low and I will focus on memory impact both Flash and RAM wise.

Code size

A common metric for the scope of a project is the rectified number of Lines of Code (LoC) and the amount of documentation. I used a combination of the terminal tool 'cloc' for files solely created by me and manual measurement of changes in existing files. This assessment resulted in the following numbers. 13 files newly created and 7 changed, with more than 700 lines of comment and documentation and over 1800 Lines of C code for this project.

Memory size

In the following I will estimate the memory needs of the IPsec suit for a minimal scenario of dynamic peer detection and key negotiation.

There are two common scenarios for a minimal bidirectional setup. 'Permissive filtering' and 'strict filtering'. 'Permissive filtering' aims to only protect specific traffic and let all other flow freely. Therefore only one general SPD rule is needed that permits all traffic. The only other rules are for the traffic that is to be protected. In this scenario though, SPD cache entries would still be created for every connection. In the 'strict filtering' scenario all traffic beside the one explicitly permitted is dismissed. Therefore rules need to be added to enable neighbour discovery and key negotiation. The number of SA entries would not vary between the scenarios. Without changes to the implementation of SPD cache generation, both solutions do not have a huge difference. I then estimated the minimal amount of database entries for a strict filtering scenario and their cumulative data size after successful initiation.

Database Entries of Client A		
SPD Rules struct size: 61 byte	SPD Caches struct size: 42 byte	SAD struct size: 83 byte
PROTECTED Tx A⇒B PROTECTED Rx B⇒A IKEv2 TRx BYPASS ICMPv6 TRx BYPASS loopback TRx BYPASS default TRx DISCARD	PROTECTED Tx A⇒B * BYPASS Tx IKEv2 BYPASS Rx IKEv2 BYPASS Tx ICMPv6 BYPASS Rx ICMPv6 BYPASS Rx loopback BYPASS Tx loopback	PROTECTED Tx to B PROTECTED Rx from B
366 byte	294 byte	166 byte
826 byte		

* this cache entry holds a pointer linking to the according SA

Tx Transmission - Rx Reception - TRx Transception

Table 7.1: Calculating the minimal database use for one client in a bilateral connection protected by IPsec

As can be see in table 7.1 analysing the minimal memory use of the IPsec database, even in the simplest setup the allocated space is close to an Kilobyte of memory. The struct sizes are taken directly from the implementation.

If needed the memory allocation could be reduced by making the structs into dynamic structures that instead of holding unused fields, are linked to a NULL pointer if the field is empty or idle. I saw this as to much added complexity to the database system and especially to the methods that would do an immense amount of NULL pointer checks, obfuscating the code extremely.

7.5 Security

Since there is no user space, any application could bypass IPsec, using raw sockets under Gnu/Linux. But the attack scenarios are different for IoT since outside access on a deployed

system normally very limited. But IoT systems are weak against DoS attacks in general [12]. It is unclear if ESP increases or reduces this vulnerability. On the one hand, traffic can be filtered early on in the processing, on the other hand, any packet will trigger an IPsec databases check and a possible cascade. So especially DDoS attacks could fill up the memory available for the database quickly. Countering this by limits for SPD-I cache can still render the device unresponsive. Specific rulesets on how to react when the memory is filling by a - intentional or unintentional - DoS, are essential for the deployment of ESP. Fabricated malicious ESP packets could put even more load on the system, but will be countered by the integrity check.

8 Conclusion and Future Work

8.1 EHC and DietESP on the Horizon

I already mentioned the advanced possibilities when deploying EHC or DietESP in section 4.2. While implementing the framework for IPsec I always kept the requirements for this concept at hand and tried to build verbose code wherever it would ease future integration. One limitation I encountered was created by the fact that some information would not be available at the time of IPsec traffic filtering when payload compression would be used. Since the correct setup of the SPD rule set, especially in multicast traffic, requires that the SPD and its caches aren't checked on incoming PROTECTED packets before checking on the SAD, the SA information defining the LSB length for the SPI under Diet-ESP would need to be identified before ESP processing. One solution I came up with, would be that all devices that are connected, use the same size of LSB for the SPI reduction. But this contradicts with the idea of a worldwide interconnected use. There certainly is more to be cleared up and discussed, regarding this topic. Besides this there are a handful of additions and changes that would need to be made to the implementation to get EHC and Diet-ESP going. At multiple places in the code I have left comments regarding the implementation of Diet-ESP and deem it a vital if not even essential feature to pave the way for ESP protection in IoT.

8.2 IPsec for IoT

This thesis designed a generalized ESP implementation for development and scientific use. For special use cases and field deployments extreme optimizations could be made. E.g. could a send only sensor with only one centralized server, drop ip filtering and esp handling completely and reduce the IPsec Databases to a single static memory entry. But this scenario would exclude a multitude of other scenarios, where dynamic network size, multicasting and additional session negotiation are needed. But I think this specialized use case is very common and worthwhile to explore. The goal would be, to minimize the database while still allowing dynamic key exchange and the secure switching between different master systems. This restricted implementation would be able to be so extremely optimized, that an own identifier would be needed for this Single Master, limited ESP (SML-ESP). This miniaturization can not be achieved as a mere configuration setup and thus needs separation from more general ESP. In conclusion it is clear that the IPsec suit is complicated, because it is so extremely versatile. And the invention of Diet-ESP underlines this. Attempts to limit an IPsec implementation to a specialized use case, should be seen independently from a general implementation, utilizing the strengths if IPsec for group communication and meshed networks.

The complexity of ESP and all its dependencies is the major drawback. No single implementation of ESP can be optimal for all IoT scenarios. The trade-off between dynamic

and static memory management and the need to precisely and tediously setup any of them, understanding the inner workings of IPsec and considering the specific scenario, reduces the intended abstraction and will make the use of integrated ESP implementations a complex task. Though there are easier ways to protect traffic, the protection of IPv6 network traffic on the IP layer using ESP combined with EHC and the resulting reduction of radio frames, clearly beats the resource overhead for any professional application. This is already true for state of the art SoC hardware and will become even more relevant with future hardware development featuring ever more memory to work with. So even if IPsec is neither simple nor perfect, IPsec is the most common, modern and especially a very modular solution to secure low level IP traffic.

8.3 Emphasis for future development

There is no single optimal solution for the issues of memory management in embedded hardware. But the options need to be easy to configure, since one cannot expect every developer to rewrite the memory management for every RIOT IPsec project. If possible a set of options to choose and setup from would be good.

Observing that the majority of database size is used by the SPD caching of unprotected traffic, one should consider to deviate from the standards and only construct SPD caches for entities that are to be protected. The length of SPD rule sets deployed in IoT should be reasonable.

Working with these protocols generates many ideas for extended concepts. So far SAs have been either manually or dynamically deployed and there is no established way to change the SPD rules on an IoT system. An idea emerged from this, to deploy a half static, half dynamic setup. Just as Diet-ESP is relying on predefined strategies one could deploy an IPsec addition in which one SA, precompiled into the system, serves as the initial connection to an authentication entity. All further SPD rules and connection details would then be deployed remotely over this secure connection. Thus a device could be deployed anywhere and whenever it achieves a communication with the global network it will automatically get integrated in this distributed network, even if the network already changed since deployment.

To integrate EHC and Diet-ESP into the implementation an extension would be needed for the SA struct, resulting in changes to at least the dbfrm. Also the EHC database would need to be established if the use of more than one hard coded strategy is intended.

Since a full detachment from the existing IPv6 routines was neither possible nor feasible, the code hooks are woven into the regular IPv6 handling. It is assured no packet can bypass the IPsec gatekeeper routines for incoming and outgoing IP traffic. A small uninformed change to the source though, even in a different place of the kernel, could route a packet around this firewall and the esp detection mechanics. For a safe deployment into productive code, one must deploy a set of tests to check if packets get filtered and bypassed as intended. Since these require not only straight forward tests but the initiation of complex system variables - e.g. the IPsec databases - I omitted them from my implementation for complexity reasons.

Most important though is the integration of an IKE protocol into the framework generated by this work to lift the implementation from theoretical research into real world scenarios.

8.4 Conclusion

Foremost I could prove that my design and implementation work in the scope of native RIOT OS and yield packets correctly detected and inspected by Wireshark.

While I implemented ESP and parts of IPsec into RIOT OS, many question and roadblocks showed up, some of which could not be finally answered in this thesis. Especially when it comes to certain flaws of IPsec, like its complexity and its clash with the limited resources of constrained node operating systems. Others have been answered and clarified. Especially the details regarding how exactly the individual components are interconnected and how the traffic flows. Details, that all had to be accounted for when placing the gateway routines in the existing code, have helped to lay a solid and open source foundation for more experimental projects to build on. Overall I think the generalized and widely supported traffic protection of IPsec is becoming ever more relevant for a modern and dynamically interconnected world. This thesis and its implementation therefore hopefully serve to gain ground for IPsec as a modern and secure solution to the next generation of computer networks.

Appendix A: ESP Test Setup on Linux

In the following the procedures are described to test the ESP implementation for RIOT OS inside the RIOT native environment on a Linux machine.

1 Environment Setup

Following the tuntap setup instructions for Linux, stated in the README.md file of the example 'gnrc_networking', or by executing the bash script 'RIOT\dist\tools\tapsetup\tapsetup', one creates virtual interfaces and a bridge between them. It should then be possible to attach two RIOT client sessions to these interfaces by calling:

```
Term1:  ..\examples\gnrc_networking_ipsec\$ PORT=tap0 make term

Term2:  ..\examples\gnrc_networking_ipsec\$ PORT=tap1 make term
```

2 Filter Rule Setup

The basic filter rules are for now hardcoded inside of the gnrc_ipv6_keyengine.c file since any dynamic setup is handled by dbfrm injection anyway. Thus they should not need to be changed. It is comprised of three rules:

```
1. BYPASS   on loopback traffic

2. BYPASS   on all IPv6-ICMP traffic (IP Protocoll Number 58)

3. DISCARD  on any packet not catched by a previous filter rule
```

3 Client Setup

After setting up the filter rules according to the planned environment in appendix A, we give a global IPv6 address to the two test clients and use the command-line tool dbfrm to inject every clients database set with a manual SPD-cache rule and an according SA entry. Then we start an UDP server on the receiving client 'Rx' and send UDP packets from client 'Tx'. The TRANSPORT example shows a setup to generate ESP traffic in transport mode, the TUNNEL example self encapsulates the traffic with ESP. The Rx clients USP routine will show a message when successfully receiving and decoding the ESP message. The manual for

the dbfrm tool can be accessed by calling 'dbfrm –help' or by reading the README.md of the 'gnrc_networking_ipsec' example.

TRANSPORT setup:

```
Rx:  ifconfig 8 add 2000::2/64
Rx:  dbfrm protect '42' '13371337' '2000::2' '2000::1' 17  NULL NULL transport
     comb none '0' mockup 'b7396d693045f060' '4242424242' NULL NULL
Rx:  udp server start 1764


Tx:  ifconfig 8 add 2000::1/64
Tx:  dbfrm protect '42' '13371337' '2000::2' '2000::1' 17  NULL NULL transport
     comb none '0' mockup 'b7396d693045f060' '4242424242' NULL NULL
Tx:  udp send 2000::2 1764 "Example Payload 1234567890"
```

TUNNEL setup:

```
Rx:  ifconfig 8 add 2000::2/64
Rx:  dbfrm protect '42' '13371337' '2000::2' '2000::1' 17  NULL NULL tunnel
     comb none '0' mockup 'b7396d693045f060' '4242424242' '2000::2' '2000::1'
Rx:  udp server start 1764


Tx:  ifconfig 8 add 2000::1/64
Tx:  dbfrm protect '42' '13371337' '2000::2' '2000::1' 17  NULL NULL tunnel
     comb none '0' mockup 'b7396d693045f060' '4242424242' '2000::2' '2000::1'
Tx:  udp send 2000::2 1764 "Example Payload 1234567890"
```

List of Figures

Bibliography

[1] GNRC Network Stack. `https://riot-os.org/api/group__net__gnrc.html`. Accessed: 2019-09-16.

[2] Strongswan. `https://www.strongswan.org/`. Accessed: 2019-09-12.

[3] TLSF (Two-Level Segregate Fit). `http://www.gii.upv.es/tlsf/`. Accessed: 2019-09-15.

[4] Wireshark. `https://www.wireshark.org/`. Accessed: 2019-09-10.

[5] IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs). *IEEE Std. 802.15.4-2011*, 2011.

[6] Ericsson Mobility Report - June 2018, 2018.

[7] E. Baccelli, C. Gueundogan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, , and M. Waehlisch. RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT. *IEEE Internet of Things Journal*, 2018.

[8] E. Baccelli, O. Hahm, M. Gueunes, M. Waehlisch, and T. C. Schmidt. RIOT OS: Towards an OS for the Internet of Things. *INFOCOM'2013 Demo/Poster Session*, 2013.

[9] C. Bormann, M. Ersue, and A. Keraenen. Terminology for Constrained-Node Networks. RFC 7228, May 2014.

[10] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, Nov 2004.

[11] W. Engelbrecht. Group Key Management with Strongswan, 2018.

[12] O. Garcia-Morchon, S. Kumar, and M. Sethi. Internet of Things (IoT) Security: State of the Art and Challenges. RFC 8576, Apr. 2019.

[13] T. Guggemos. Diet-ESP: Applying IP Layer Security in Constrained Environments, 2014.

[14] T. Heider. Minimal G-IKEv2 implementation for RIOT OS, 2017.

[15] V. Jutvik. IPsec and IKEv2 for the Contiki Operating System. June 2014.

[16] C. Kaufman, P. E. Hoffman, Y. Nir, P. Eronen, and T. Kivinen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 7296, Oct. 2014.

[17] S. Kent. IP Encapsulating Security Payload (ESP). RFC 4303, Dec. 2005.

[18] B. Martinez, M. Monton, I. Vilajosana, and J. D. Prades. The power of models: Modeling power consumption for iot devices. *IEEE Sensors Journal*, 15(10):5777–5789, Oct 2015.

[19] C. Metz, D. L. McDonald, and B. Phan. PF_KEY Key Management API, Version 2. RFC 2367, July 1998.

[20] D. Migault, T. Guggemos, C. Bormann, and D. Schinazi. ESP Header Compression and Diet-ESP. Internet-Draft draft-mglt-ipsecme-diet-esp-07, Internet Engineering Task Force, Mar. 2019. Work in Progress.

[21] D. Migault, T. Guggemos, and Y. Nir. Implicit IV for Counter-based Ciphers in Encapsulating Security Payload (ESP). Internet-Draft draft-ietf-ipsecme-implicit-iv-07, Internet Engineering Task Force, Apr. 2019. Work in Progress.

[22] Y. Nir. ChaCha20, Poly1305, and Their Use in the Internet Key Exchange Protocol (IKE) and IPsec. RFC 7634, Aug. 2015.

[23] F. G. of the United States.

[24] S. Raza, S. Duquennoy, and G. Selander. Compression of IPsec AH and ESP Headers for Constrained Environments. Internet-Draft draft-raza-6lowpan-ipsec-01, Internet Engineering Task Force, Sept. 2013. Work in Progress.

[25] E. Rescorla. Diffie-Hellman Key Agreement Method. RFC 2631, June 1999.

[26] K. Seo and S. Kent. Security Architecture for the Internet Protocol. RFC 4301, Dec. 2005.

[27] P. Thubert, C. Bormann, L. Toutain, and R. Cragie. IPv6 over Low-Power Wireless Personal Area Network (6LoWPAN) Routing Header. RFC 8138, Apr. 2017.

[28] P. Thubert and J. Hui. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. RFC 6282, Sept. 2011.

[29] B. Weis and V. Smyslov. Group Key Management using IKEv2. Internet-Draft draft-yeung-g-ikev2-16, Internet Engineering Task Force, July 2019. Work in Progress.

[30] P. Wouters, D. Migault, J. Mattsson, Y. Nir, and T. Kivinen. Cryptographic Algorithm Implementation Requirements and Usage Guidance for Encapsulating Security Payload (ESP) and Authentication Header (AH). RFC 8221, Oct. 2017.