INSTITUT FÜR INFORMATIK

DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master Thesis

# Diet–ESP

Applying IP–Layer Security in Constrained Environments

Tobias Guggemos

INSTITUT FÜR INFORMATIK

DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master Thesis

# Diet–ESP

Applying IP–Layer Security in Constrained Environments

Tobias Guggemos

| | |
|---|---|
| Evaluator: | Prof. Dr. Dieter Kranzlmüller |
| Supervisor: | Daniel Migault PhD (Orange Security Labs Paris) |
| | Dr. Michael Schiffers |
| Submission Date: | August 11, 2014 |

I assure the single handed composition of this master's thesis only supported by declared resources.

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

Munich, August 11, 2014 ..........................................................

*(Tobias Guggemos)*

# Abstract

The Internet of Things (IoT) is a research topic the significance of which is continuously increasing. The devices taking part in the IoT are significantly more resource constrained than the devices used in today's networks. In order to cope with these restrictions, the protocols used in the Internet have to be reviewed and adapted to the requirements of these devices. One of these constraints is the combination of limited power and high cost for networking. To satisfy this requirement, many IoT protocols use compression techniques to limit the protocol overhead. The security protocols in use today already put a lot of strain on the large and complex infrastructures they have been designed for and are therefore too costly for IoT devices. The IPsec protocol suite is one such protocol, which is mainly designed for securely interconnecting large networks, but not for constrained devices. However, the proper design of the IPsec architecture, together with the separation of key exchange and data security to different protocols, would provide proven security features for IoT. In order to combine the security features of IPsec with the constraints of IoT devices, this thesis introduces Diet−ESP, which is an adaption of the ESP protocol. Diet−ESP considers the requirements of constrained devices but sustains the security features initially provided by IPsec. By introducing a compression context which can be directly accessed by the IPsec implementation, Diet−ESP allows high compression rates. Additionally, it is flexible and easy to implement. The results show that Diet−ESP is able to reduce the energy consumption by 40% which can nearly double the lifetime of a sensor. With the start of the standardization process of Diet−ESP at the IETF, it has the potential to be included to the ongoing development of the Internet of Things.

# Contents

# 1 Introduction

Microcontrollers and sensors in the Internet of Things (IoT) will significantly grow up in the near future. A few years ago, sensor networks were exclusively used for research or monitoring of infrastructure projects. One famous example is the Golden Gate Bridge in San Francisco, with its own sensor network, monitoring the oxidizing of the bridge's metal components [38]. Such devices are bound to find their way into everyone's everyday life, be it through automated intelligent emergency systems, home automation or even Smart Cities. Vendors of infrastructure, like telecommunication companies, and manufacturers of such devices are spearheading this development together with public institutions like city governments. Especially metropolises like Paris are supporting this development in order to improve traffic flow, tourist movements and public internet access sometimes together with international governments like the European Union [52].

These devices and networks can be used for all kinds of small applications and some of them may require secure communications. They can be life critical (like a fire alarm), security critical (like home theft alarms) or made for home automation. Smart grid [18] is one application where electricity is supplied by using information each home delivers. Similarly, home temperature might be determined by servo–controls based on information provided by temperature sensors.

IoT devices come with specific requirements for security protocols. First of all, many sensors do not have an attached power supply why, therefore they need to operate with a battery, which limits the life time of the device. The power is needed for computation within the CPU and the network interface which sends and receives data. Most sensors have highly optimized processors for minimal power consumption. Although the network interfaces are also optimized, their power consumption is, on average, 10–100 times higher than for processor instructions (section 2.1). One of the reasons is that sensors often rely on wireless communication in order to make them more flexible. The higher the radius of the wireless interface the higher the energy consumption. In fact, the energy consumption increases exponentially to the distance between sender and receiver. Saving energy, many devices are configured as sleepy nodes, shutting down in idle mode and waking up periodically to receive messages or for sending their information. Therefore the IEEE 802.15.4 [3] standard is designed for IoT communications supporting sleepy nodes. The standard also defines a low Maximum Transmission Unit (MTU) of 127 bytes in order to take the higher costs for network operations into account.

The expected increasing number of IoT devices in the future is one of the reasons why internet providers adopt IPv6 [11] inside their network infrastructures. It ensures a unique IP address for every device, but for IoT it comes with a new problem, the increasing packet size due to the bigger IP addresses of 16 bytes. The IP header itself needs 40 bytes and therefore nearly one third of the available 127 bytes of IEEE 802.15.4. Including the MAC layer headers of maximum 25 bytes, there are only 62 bytes left for Transport and Application Layer. Because of this issue 6LoWPAN [17] (IPv6 for Low Powered Area Networks) defines header compressions, mainly for IPv6 and UDP [60], in order to reduce the size of the header sizes. This standard is designed for sensors behind a gateway, which connects them to the Internet. The gateway decompresses all compressed headers, ergo the connection endpoint always receives a regular IP packet structure.

This works perfectly, as long as the connection does not need to be confidential. Therefore IEEE 802.15.4 specifies Link−Layer−Security by using AES−CCM [82] requiring another 21 bytes but offering only Hop−by−Hop Security without the possibility of End−to−End−Security. For securing the connection from endpoint to endpoint, one may use TLS [7], DTLS [39] or IPsec [37] providing approved protocols for securing internet communications. But since compressing encrypted payload cannot be done securely, the compression within other layer has to be discussed. The compression itself may be done with already existing protocols like 6LoWPAN or ROHC (RObust Header Compressions [3]).

Since TLS is widely spread over the Internet, there is a lot of research interest in porting TLS or DTLS on constrained devices, but IPsec would provide some advantages for securing End−to−End communications, such as:

- IPsec secures application communications transparently as security is handled at the IP layer. As a consequence, applications do not need to be modified to be secured.
- IPsec does not depend on the transport layer which leads to the security framework remaining the same for all transport protocols, like UDP or TCP [62].
- IPsec is well designed for sleepy nodes as there are no sessions. Information about the secured communication are stored in the Security Policy Database (SPD) which can be seen like firewall rules. The security relevant data for one connection like cipher keys are stored in the Security Association Database (SAD). This databases are set up once for a connection and can be stored on the device even if it shuts down.
- IPsec defines security rules for the whole device, which outsources the device security to a designated area. Therefore IPsec can be seen as a tiny firewall securing all communications for an IoT device.

IPsec needs access to the devices IP stack that is why it is mostly implemented in the Kernel, whereas application are in the user space. This turns out to be a common disadvantage of IPsec, because application developers want to control the security which isn’t feasible when using IPsec. Additionally, an application developer has to trust the operating system and has to rely on the correct setup of the IPsec connection. On the other hand, since there are no real distinctions between these two spaces in IoT devices, being mostly designed for a specific and unique task, this may not be an issue anymore. At the time of designing IPsec, IoT constraints have not been considered and IPsec has been mainly designed for secure infrastructure.

Even though IPsec is a requirement for IP communication and every IP implementation has to provide it, there is always room for improvements. Some researchers picked up the idea of using IPsec in IoT environments using 6LoWPAN or other compression protocols to reduce the protocol overhead. But all of them stuck in compressing the encrypted ESP [36] payload. The most common idea is a second, modified ESP−stack handling the compressions before the encryption occurs. Assuming that the endpoint has the same modification of the ESP−stack for decompression, this may work fine, but it looks more like a hack of ESP than a proper designed protocol.

This thesis describes a new ESP extension called Diet−ESP, enabling a flexible compression of all ESP protocol fields, additional cipher data and Transport Layer protocols like UDP or TCP, but also allows future extensions for other protocols and layers. Diet−ESP takes advantage of the already existing IPsec databases storing many data that can be used for compressions. IP addresses, transport layer protocol and ports are the most common of them.

The compression mechanisms defined in this work are based on ROHC and ROHCoverIPsec [15–17]. However, ROHC has been designed for flexible and robust bandwidth optimization, but not necessarily for constrained devices. Especially the robustness of the protocol works against IoT requirements defined section 4. In order to enable maximum robust compression, ROHC defines a set of signaling messages, compression states and complex compression algorithm. Such an implementation, together with the increasing networking overhead makes ROHC not useable in constrained environments.

In order to achieve the specific goals for IPsec in sensor networks, this document describes the Diet−ESP context. The context contains all parameters necessary to compress an ESP packet by using the mechanism defined for ROHC frameworks. More detailed, it describes how to derive a ROHC compression context from the Diet−ESP context and the IPsec databases. The advantage of using ROHC is that compression behavior follows a standardized compression framework. On the other hand, deriving profiles and methods from the Diet−ESP context makes the use of ROHC and ROHCoverIPsec dependent on the implementation, and any implementation that behaves in a similar way will be interoperable. This is what makes light implementation for IoT devices achievable.

The context can be synchronized between the security endpoints during the Key Exchange, which usually sets up the IPsec databases. The context is stored as in the SAD where the ESP stack has access to its information and it will survive even if the device shuts down.

This way of compression enables removing all protocol specific fields from an IP packet, leaving IP header compressions to 6LoWPAN. In best case scenarios the IP payload can be compressed by 80 bytes which significantly increases the sensor lifetime (see section 7.3)

This thesis is structured as follows:

Section 2 describes the background necessary for this work. It lists all information necessary for sensor networks, including their functionality. Moreover the specific standards like IEEE 802.15.4 or 6LoWPAN are defined. In section 2.2 the IPsec protocols are explained with a detailed overview over the specific packet formats and the differences between the IPsec modes.

Security Management in section 2.2.3 clarifies how security information like cipher keys can be exchanged over a not trustable environment like the internet. Due to the proper design of the IKEv2 [34] protocol it became part of the IPsec protocol suite [37] and the most common used key exchange protocol in IPsec environments, but the basic ideas of this protocol are the same for TLS or DTLS. In the further thesis, section 3 describes the possible use cases of secured IoT connections where Diet−ESP should be used in. This use case lead to a couple of IoT specific requirements for IPsec and security protocols in general which are listed in section 4. Like mentioned in the beginning there is some work related to IPsec in constrained environments and other security protocols. This work is analyzed in section 5. At last section 6 describes the design of Diet−ESP.

At the beginning, the Diet−ESP context, which is based on the ROHC context explained in section 6.1, is defined in section 6.2. It specifies the compression context, which is later stored in the Security Association to a specific IPsec connection, enabling the compression of all ESP fields. Like already mentioned, Diet−ESP allows extensions of the Diet−ESP context. This thesis defines two Diet−ESP context extensions, the first one in section 6.3 for the compression of encrypted upper layer protocols like UDP or TCP. The second extension in section 6.4 describes how the AES Initialization Vector for AES−CTR can be compressed. Section 6.6 describes a minimal ESP implementation, in order to provide the basic ideas of ESP. This is necessary to understand the add−on of Diet−ESP to a regular ESP implementation and the Diet−ESP protocol itself, described in section 6.7.

Section 7 describes two implementations of Diet−ESP. They provide results for the evaluation. First of all they prove the possibility to implement Diet−ESP in an easy and straight forward way. In addition they show that Diet−ESP is configurable to communicate with already existing ESP implementation like the Linux specific *ip xfrm*. The implementation for Contiki [8] (section 7.2) shows a "prove of concept", how to implement Diet−ESP for a sensor operating system and inside an already existing IPsec implementation as an add−on.

Section 8 discusses Diet−ESP, before section 9 concludes the thesis and shows possible future work.

# 2 Background

This section points out the background related to this thesis. First, it describes how sensor networks are usually set up. Since this work describes a modification of ESP, which is an IPsec protocol, IPsec itself is described later, together with security management through untrusted networks with IKEv2 as an example.

## 2.1 Sensor Networks

Sensors can be used in different scenarios, for example to observe buildings, cities or natural environment to specific characteristics, like the temperature or to control attached devices, like garage doors. Of course, a sensor can be used for both scenarios in the same time as well. Home Automation is an area where sensors measure information and control other attached devices according to the measured data. Most likely they do not make the decisions or evaluate results. This is the reason why they are usually connected to another, more powerful device for collecting information or making decisions.

A sensor network is a union of sensors, like shown in Figure 2.1. Usually, a couple of sensors are defined to interact with each other and connected to one defined gateway. This gateway represents the access point to the attached sensor network and sometimes it is even the same device like the connected sensors, but with other functionalities. If one sensor within the network communicates with the outer world, the connection is provided by this gateway.

If one application is security critical, there are several potential attacking vectors, some of them are pointed out from Granjal et al. [22] and Jain et al. [30]. Generally speaking, the attack can be within the sensor network, if the sensor environment is not secured. This can be the case if the sensors are communicating with a wireless connection. Therefore IEEE 802.15.4 [3] introduces link−layer security, which provides hop−to−hop security for the data. The connection between the gateway and the endpoint of the connection could be not trustable as well, for example if the connection is established through a public network, like the Internet. Sometimes, the gateway can provide a secure communication to the endpoint with TLS, DTLS or IPsec, but only if it is a not constrained device. If the gateway is a sensor, it cannot carry that complexity in the current state of research. However, such a configuration can never provide real End−to−End−Security between sensor and the endpoint, which may be required in some setups. Granjal et al. [22] clarify that IPsec would be a valuable option for providing End−to−End−Security. They focused on the energy costs for encryption and authentication used for securing IPsec communications.

The devices inside a sensor network are usually constrained in energy since they are often connected to a battery instead of a power supply. This is related to their location, for example inside a water pipe, but also to make them independent from external power supply which for example could be essential for fire alarms. Having a look on Table 2.1, one can say that networking is 10−100 times more expensive than computation. This causes the huge interest in reducing the network overhead by protocol compressions like done with 6LoWPAN.

| | Power consumption |
|---|---|
| **Networking** | |
| Low–power radios < 10 mW (CC1020 [78]) | (100 nJ – 1 μJ) / bit |
| **Computation** | |
| Energy–efficient microprocessors (CoolRisc [59] or MSP430 [77]) | 0.5 nJ / instruction |
| High–performance microprocessors (ARM9) | 200 nJ / instruction |

**Table 2.1 Energy Consumption for Networking and Computation in Constrained Environments.**

### 2.1.1 Wireless Standard IEEE 802.15.4

Low–Rate Wireless Personal Area Networks (LoWPAN) uses a standard defined in IEEE 802.15.4 [3]. It describes the MAC–Layer protocol, Low–Powered Devices in the Internet of Things can communicate with each other which is especially designed to work well with sleepy nodes. The specified Maximum Transmission Unit (MTU) of 127 Bytes is very small compared to the regular Ethernet MTU of 1500 Bytes [1] or the minimum MTU of 1280 Bytes, IPv6 [11] requires. In the worst case, the MAC layer header of maximum 25 bytes leaves only 102 bytes for the payload. If AES–CCM–128 is used for protecting these messages, there are only 81 octets for upper layers left. If no compression is used for the IP and UDP headers, hence another 40 plus 8 bytes are needed and only 33 Bytes are left for the application layer data.

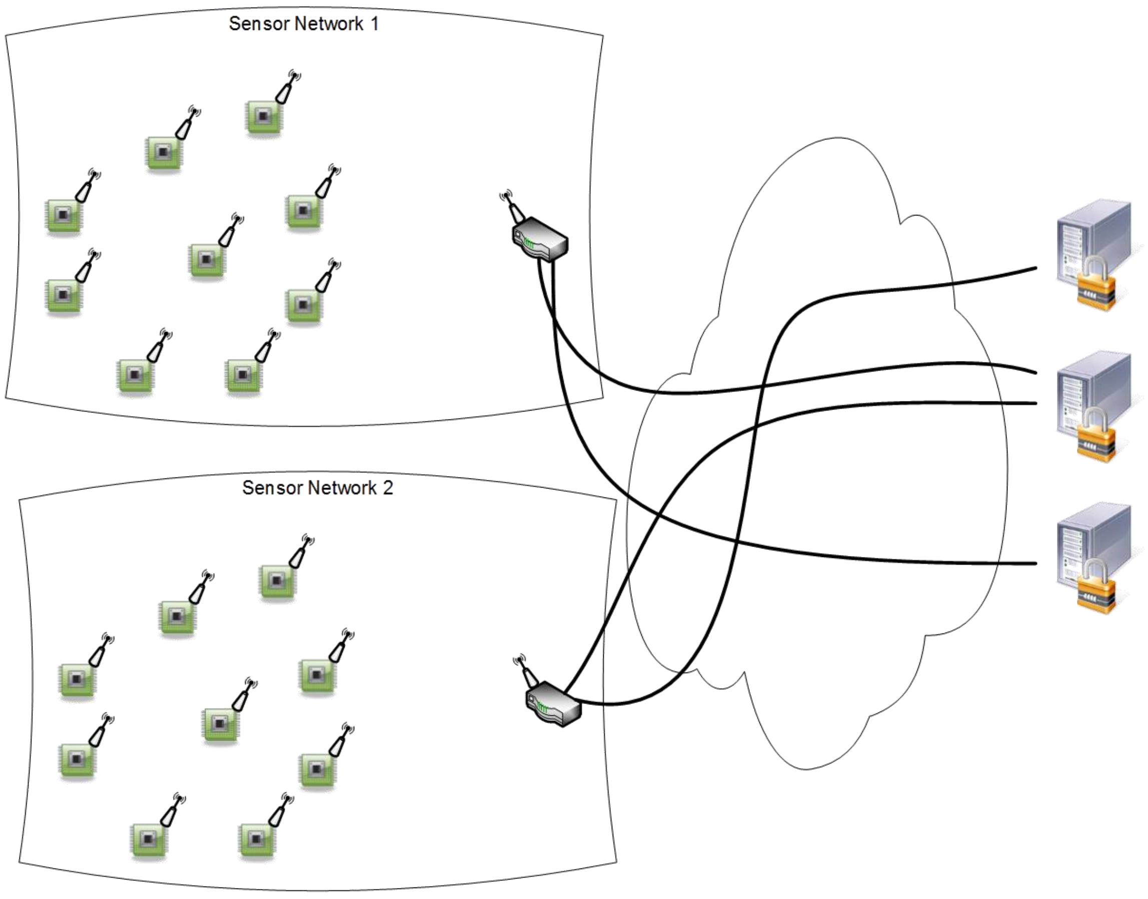


**Figure 2.1 Exemplary structure of a sensor platform.**

IEEE 802.15.4 uses AES–CCM* for link layer security, which relies on upper layer key management. The IEEE 802.15.9 working group defines how to wrap existing key management protocols like IKEv2 [34], HIP [50] or 802.1X [2] to existing 802.15.4 frames. The specification of AES–CCM for Link–Layer Security coaxed sensor producers to implement AES in hardware most common inside the wireless module. Hence, all AES modes can benefit from this hardware acceleration which improves the costs of security of AES significantly.

### 2.1.2 IPv6 Compression

IPv6 is the standard for IoT communications, but due to the larger IP addresses of 16 bytes, IPv6 has an essential disadvantage in IEEE 802.15.4 connections. 6LoWPAN [26] defines a compression standard for IPv6 packets which allows the reduction of the standard 40 bytes IPv6 header by using a 16 bit compression header (see Figure 2.2). Additionally, it allows compression of IPv6 extension headers and transport layer headers by using another set of compression headers. There are several RFCs or drafts for compressing protocols like UDP [26], IPsec [66], (D)TLS [68] or even generic headers [5]. 6LoWPAN can be used in environments like the one depicted in Figure 2.1, in which a gateway is used to compress and decompress. The compression only affects the expensive wireless communication within the sensor network, but not the communication to the endpoint of the connection.

Compression is mainly done by reducing large header fields. Some IPv6 fields for example, are compressed into one or two bit fields by defining most useful or well–known values in a few bits. Values that cannot be compressed within the 6LoWPAN header are added within the packet payload in the same order the protocol standard describes them.

```
  0                                       1
  0   1   2   3   4   5   6   7   8   9   0   1   2   3   4   5
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 1 |   TF  |NH | HLIM  |CID|SAC|  SAM  | M |DAC|  DAM  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

TF:   Traffic Class, Flow Label   NH:  Next Header
HLIM: Hop Limit                   CID: Context Identifier Extension
SAC:  Source Address Compression  SAM: Source Address Mode
M:    Multicast Compression       DAM: Destination Address Mode
```

**Figure 2.2 LOWPAN_IPHC Header, describing the fields for the compressed IPv6 format [26: p. 5].**

The Hop Limit serves as a good example to clarify the idea of 6LoWPAN compression. In IPv6 this field is defined as an 8 bit field. 6LoWPAN offers a compression to 2 bit (see Figure 2.3) holding three common used values (1, 64 and 255). If the value "00" is used, the whole 8 bit Hop Limit is placed after the compression header.

00: The Hop Limit field is carried in–line. (no compression is used)
01: The Hop Limit field is compressed and the hop limit is 1.
10: The Hop Limit field is compressed and the hop limit is 64.
11: The Hop Limit field is compressed and the hop limit is 255.

**Figure 2.3 Hop Limit Compressions in 6LoWPAN.**

Compressing the IP addresses is the main advantage of 6LoWPAN, because the source and destination address inside the IPv6 header require 32 bytes (256 bits) together. The Source

Address can be compressed to 64, 16 and 0 bits using a combination of the fields SAC (compression algorithm) and SAM (compression mode) of the LOWPAN_IPHC header. The Destination Address is a bit more complex to compress, since it carries the more important information for delivering the packet. 6LoWPAN defines a reduction to 64, 48, 32, 16 and 8 bits. Similarly to the Source Address, the DAC and DAM defines the level of compression, but additionally the M field specifies if the address is a multicast address or not.

All possible combinations of these fields are specified in RFC6282 [26].

### 2.1.3 Application Protocols

There are some well–known application protocols especially designed to manage the requirements of low powered devices. One of them is Constrained Application Protocol (CoAP) [7], an UDP based protocol providing similar functionality as HTTP but being optimized for sensors and stateless. The idea is to provide services over a stateless connection, which is handled within the application. It is widely spread for IoT communication, since UDP is the common used Transport Layer Protocol. CoAP defines four kinds of communication messages:

- **CON:** Sending a message with needed confirmation.
- **NON:** Sending a message without needed confirmation.
- **ACK:** Acknowledgement of received message.
- **RES:** Reset a connection.

For setting up an unreliable transmission, the client only has to send a NON packet covering the normal case in an UDP connection (see Figure 2.4). Reliable transmission, the client sends a CON packet to the server and waits for receiving the ACK packet (see Figure 2.5).

```
Client              Server
   |                  |
   |   NON [0x01a0]   |
   +----------------->|
   |                  |
```

**Figure 2.4 CoAP Unreliable message transmission [73: p. 11].**

```
Client              Server
   |                  |
   |   CON [0x7d34]   |
   +----------------->|
   |                  |
   |   ACK [0x7d34]   |
   |<-----------------+
   |                  |
```

**Figure 2.5 CoAP Reliable Message Transmission [73: p. 11].**

## 2.2 IPsec

IPsec is a security protocol suite that proposes to secure any IP based traffic by providing security services for both IPv4 and IPv6 and is described in RFC4301 [37]. The protocols of the suite are Authentication Header (AH, described in RFC4302 [35]) for authenticating and Encapsulating Security Payload (ESP, described in RFC4303 [36]) for data encryption and authentication. These two protocols can be used in two different modes. The Transport Mode is mainly designed for End–To–End communication whereas the Tunnel Mode is used for connecting networks, often called Virtual Private Network (VPN).

Both protocols use Security Associations (SA) as simple data structures holding the security parameters (e.g. encryption and authentication keys) for one connection. One SA is identified by the Security Policy Index (SPI). The set of SAs is stored in the Security Association Database (SAD), offering the security services for every incoming or outgoing traffic it carries. For a bidirectional connection, there are two SAs to protect both directions of the communication. The data for defining appropriate behavior facing incoming and outgoing traffic is stored within the Security Policy Database (SPD). The Internet Key Exchange protocol version 2 (IKEv2 [34]) is designed for a secure exchange of these information.

### 2.2.1 IPsec Modes

IPsec can be deployed in different scenarios. Depending on the concerned topology, the administrator has the choice between the Transport Mode and Tunnel Mode. Both modes use the same techniques and algorithms for encryption/decryption, but there are differences in the encapsulation and building of the packet.

Both protocols AH and ESP can be used in Transport and Tunnel Mode in the same way. In the Transport Mode, designed for End–to–End–Security (see Figure 2.6), the original IP header is modified. The Next Header field is changed from its original value (e.g. UDP) to the new value AH or ESP. Both protocols include an own Next Header field in the AH header respectively in the ESP trailer. This field stores the original Next Header value of the IP packet whereby the protocol stack stays complete. Figure 2.10 and Figure 2.12 show the structure of AH and ESP packets in Transport Mode.

In contrast, the Tunnel Mode is designed for Endpoint–to–Gateway (Figure 2.8) and Gateway–to–Gateway–Security (see Figure 2.9), but can be used for End–to–End–Security, too (Figure 2.7). The original IP header, processed by the IPsec stack, is not modified by any means. The AH and ESP protocols are constructed around the original IP packet and a new IP header is generated. This new IP header contains the Next Header value for AH respectively ESP. The Next Header field within the AH header respectively in the ESP trailer holds the value IP (resp. IPv6). With this processing, the whole original IP header is secured, whereas it is only partial secured in Transport mode. In ESP Transport mode the original IP header is not secured at all. Figure 2.11 and Figure 2.13 shows the structure of AH and ESP packets in Tunnel Mode.

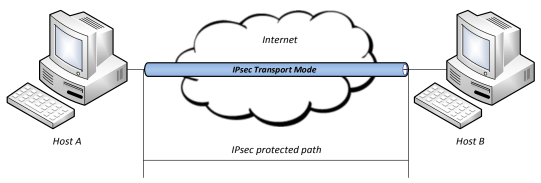


**Figure 2.6 Scenario: Transport Mode.**

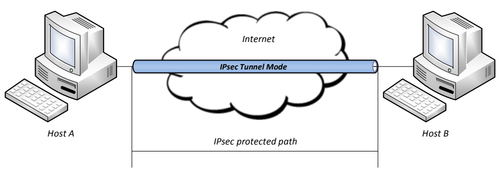


**Figure 2.7 Scenario: Endpoint to Endpoint Tunnel.**

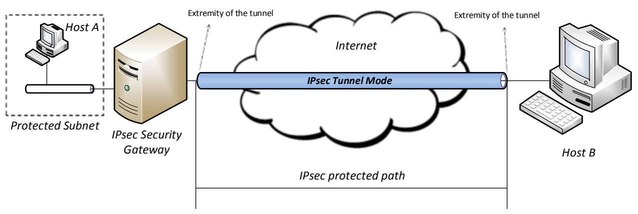


**Figure 2.8 Scenario: Endpoint to Gateway Tunnel.**

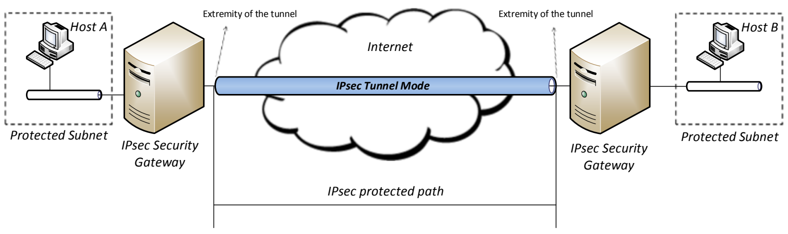


**Figure 2.9 Scenario: Gateway to Gateway Tunnel.**

There is another IPsec Mode, which is not standardized yet. The "Bound End–To–End–Tunnel" (BEET) [47] is specialized for the scenario in Figure 2.7 but in contrast to the regular Tunnel Mode the inner IP header is not sent on the wire. Especially for IPv6 this reduces the ESP payload significantly. Even not standardized, it is implemented in the XFRM module of the Linux kernel [43], resulting in a "de–facto–standard".

In order to remove (resp. rebuild) the IP header, the IP addresses inside the Traffic Selector of the SPD have to be unique on both sides of the connection. The other fields of the IP header (Hop Limit, Traffic Class and Flow Label) are copied from the outer IP header. While building the outer IP header at the sending part of the connection, these values are copied from the original IP header to the new outer IP header. The payload length can be calculated while the ESP procession.

### 2.2.2 IPsec Protocols

Right now there are two protocols provided be the IPsec protocol suite, namely Authentication Header (AH) and Encapsulating Security Payload (ESP).

The Authentication Header protocol can offer integrity, data origin authentication and optional anti–replay protection. Therefore an Integrity Check Value (ICV) is calculated including the IP payload and all immutable fields of the outer IP header. The payload includes the AH header format, which defines the Next Header holding the value of the protocol within the authenticated payload, an authentication length, the SPI and the Sequence Number of the AH packet (see Figure 2.10 and Figure 2.11).

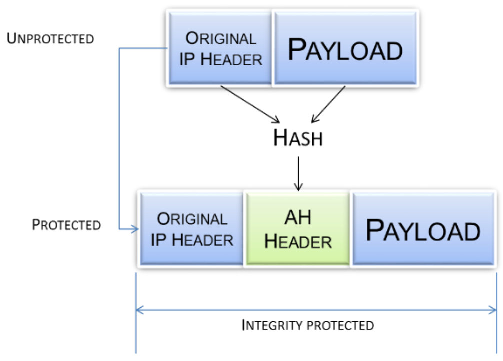


**Figure 2.10 IPsec AH in Transport Mode.**

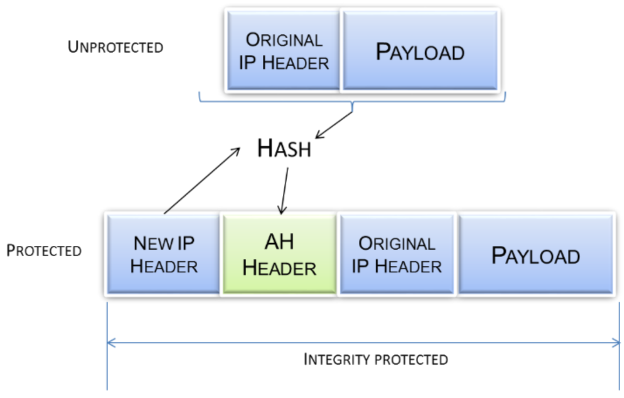


**Figure 2.11 IPsec AH in Tunnel Mode.**

The Encapsulating Security Payload (ESP) protocol offers confidentiality, data integrity and optional anti–replay protection. It encrypts the IP payload concatenated with eventual necessary padding and the ESP trailer. This trailer holds the Pad Length and Next Header field, holding the value for the protocol within the encrypted payload. Data integrity is done optional by calculating an Integrity Check Value (ICV) of the encrypted data and the ESP header holding the SPI and the Sequence Number of the ESP packet (see Figure 2.12). This value is attached to the ESP packet. ESP is able to offer data origin authentication if it is used in Tunnel Mode since the original IP header is included to the ICV calculation (see Figure 2.13). Due to this advantage of the ESP protocol, the AH protocol is defined as optional in the IPsec protocol suite [37].

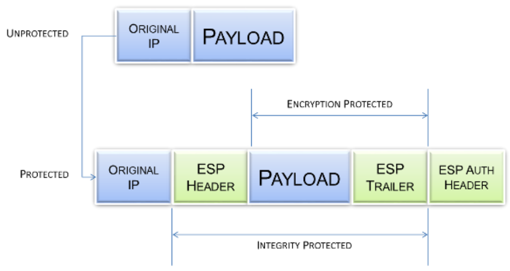


**Figure 2.12 IPsec ESP in Transport Mode.**

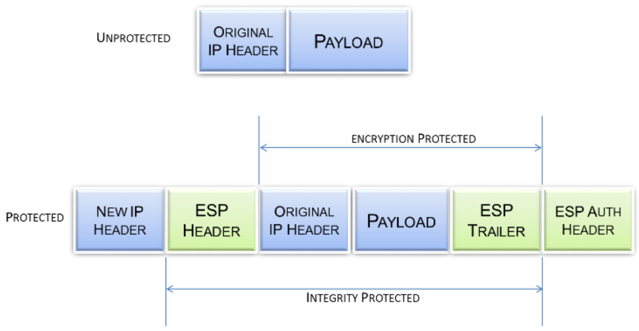


**Figure 2.13 IPsec ESP in Tunnel Mode.**

### 2.2.3 IPsec Databases

IPsec can be seen as a combination of two databases storing the necessary data shown in Figure 2.14. The Security Policy Database (SPD) holds policies about the connections to be secured. This connection is specified by a Traffic Selector including IP addresses, Transport Layer Protocols and ports. The information can be specified by using wildcards, too.

Each entry in the SPD has a link to the current used Security Association (SA) stored in the Security Association Database (SAD). One SA holds all necessary information for securing the traffic. The most common ones are the keys for encryption and authentication but there are also other information, like the lifetime of the SA or mobility information.

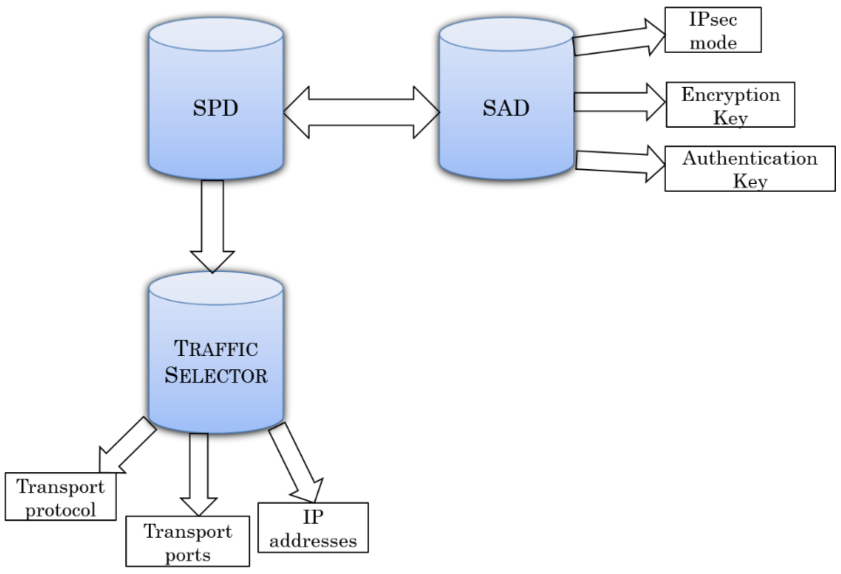


**Figure 2.14 IPsec Databases.**

Each unidirectional connection which should be secured has one entry in this database. For bidirectional connections, there is one entry for incoming and outgoing traffic. After one SA expires it is not allowed to be used again and a new entry has to be negotiated. Expiration can be caused by time, number of packets or manually e.g. due to weak keys.

## 2.3 Security Management

Using strong secrets for the different cipher algorithm is obligatory for every Security Protocol, like IPsec. Therefore a secure way of exchanging keys is absolutely necessary. The simplest idea for sharing a secret is to exchange it physically. Then, each side of the communication can determine the other side's identity. But this approach involves security risks if the exchange is done under an uncontrollable environment and in addition it is not scalable in a major infrastructure. Establishing a connection over the Internet will scale, but the Internet is a highly uncontrolled environment and not trustful at all. Therefore the Internet Key Exchange protocol in Version 2 (IKEv2) was developed and became the common way to exchange secrets for an IPsec environment. It is specified in RFC5996 [34].

IKEv2 basically defines two kinds of messages for exchanging secrets (see Figure 2.15). The first message IKE_SA_INIT establishes a secure tunnel with a Diffie−Hellman key exchange. This tunnel is used to exchange the key material within the IKE_AUTH messages. They include the supported ciphers, traffic selectors, authentication and the keying material. With that material the first IPsec SA is generated and the IPsec protocols have all the information and material they need to encrypt or authenticate the payload.

Additionally, there are two other types of messages specified in IKEv2. The CREATE_CHILD_SA is used to create new SAs for the IPsec traffic in order to generate new keys after a specified time or when the SA expires. INFORMATIONAL is used to exchange additional information. Manually removing a SA from the SAD from the other peer is one of them, but it can also be used for other informational content.

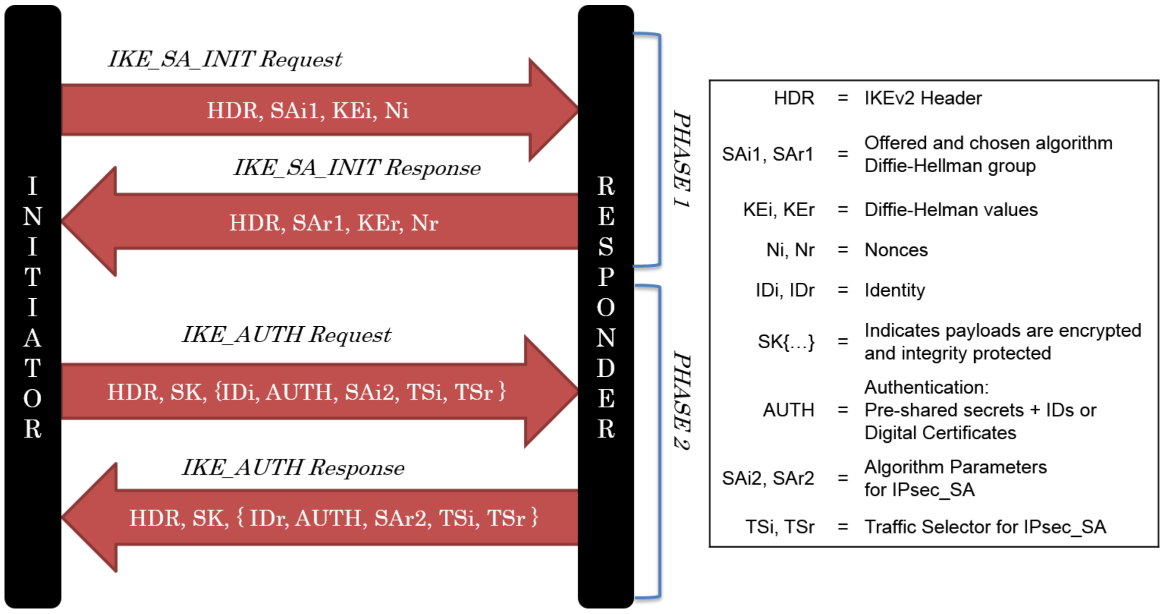

INITIATOR
RESPONDER
IKE_SA_INIT Request
HDR, SAi1, KEi, Ni
IKE_SA_INIT Response
HDR, SAr1, KEr, Nr
IKE_AUTH Request
HDR, SK, {IDi, AUTH, SAi2, TSi, TSr }
IKE_AUTH Response
HDR, SK, { IDr, AUTH, SAr2, TSi, TSr }
PHASE 1
PHASE 2
HDR = IKEv2 Header
SAi1, SAr1 = Offered and chosen algorithm Diffie-Hellman group
KEi, KEr = Diffie-Helman values
Ni, Nr = Nonces
IDi, IDr = Identity
SK{...} = Indicates payloads are encrypted and integrity protected
AUTH = Authentication: Pre-shared secrets + IDs or Digital Certificates
SAi2, SAr2 = Algorithm Parameters for IPsec_SA
TSi, TSr = Traffic Selector for IPsec_SA

**Figure 2.15 IKE INIT and IKE AUTH messages for the essential key exchange over IKEv2.**

# 3 Use Cases

Sensor networks have to deal various use cases and one network is usually designed to fulfill exactly one task. However, in case of networking the task is not relevant and it is possible to extract four basic use cases for networking in sensor networks. These are the use cases, every network protocol should be able to handle and therefore necessary to describe for Diet–ESP.

## 3.1 Use Case 1: Sending Only

One basic use case for sensors is sending data to one specific gateway (see Figure 3.1). One expected scenario this can be useful for, is a sensor sending measured data (e.g. temperature) to a collector or gateway in a fixed interval (e.g. every minute), taking computations with these data.

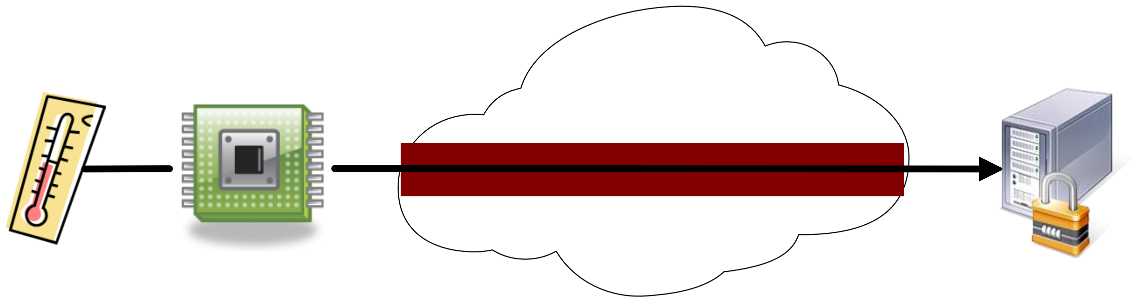


Figure 3.1 Use Case 1: Unidirectional connection to a sending uniquely identified gateway.

## 3.2 Use Case 2: Receiving Only

The second basic use case is a sensor, only receiving data from one specified endpoint (see Figure 3.2). In this case it is required, that this endpoint is never changed and always alive and reachable. One expected scenario is a sensor, which is able to control a plugged device, receiving remote signals, for example a garage door.

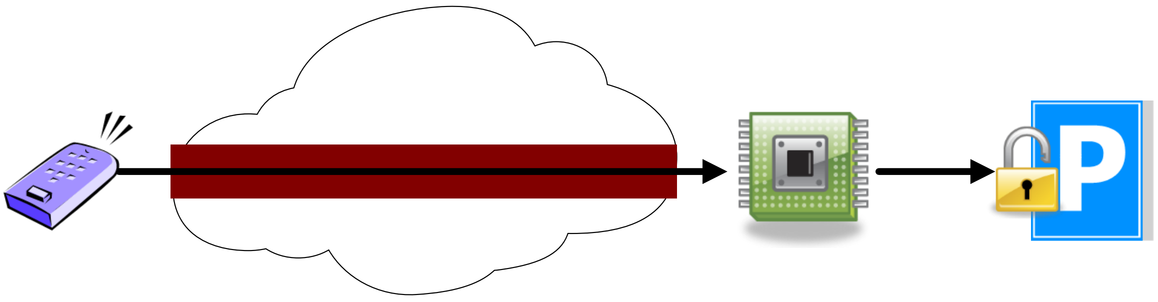


Figure 3.2 Use Case 2: Unidirectional connection receiving data from a uniquely identified endpoint.

## 3.3 Use Case 3: Sending and Receiving

This use case combines the two previous ones (see Figure 3.3). There is exactly one endpoint for the communication, but the connection is bidirectional. An expected scenario could be a microcontroller, attached to a temperature sensor and a heating system. The microcontroller sends the measured temperature data to the gateway, which is computing the best settings for the heating system. Due to this computation the gateway sends control information to the microcontroller which regulates the heating system.

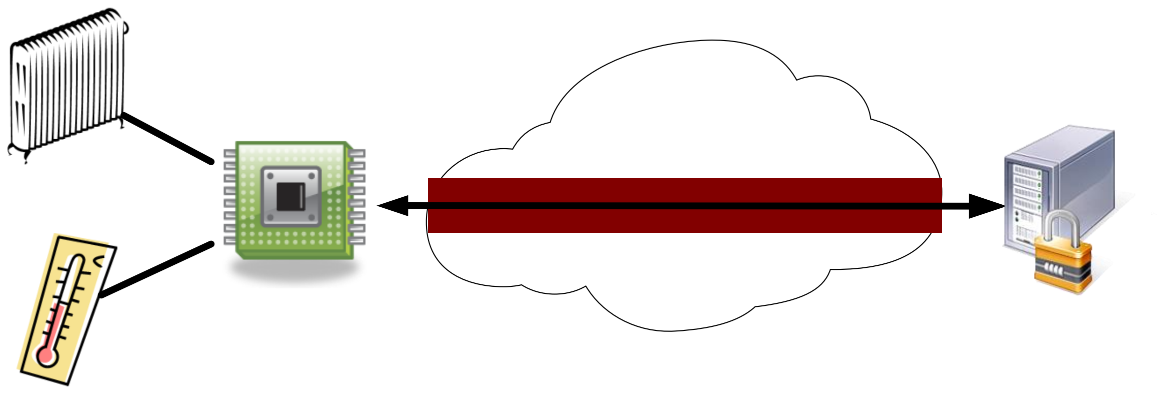


Figure 3.3 Use Case 3: Bidirectional connection to a uniquely identified endpoint.

## 3.4 Use Case 4: Multiple Connections

This use case is used if none of the previous fits the requirements and combines all previously defined (see Figure 3.4). There are two main reasons, this use case is designed for, which can be combined as well.

The first is a sensor connecting to multiple endpoints and sending and receiving data. One expected scenario is the same as described in use case 3, but the information is send to multiple devices. There could be an additional dedicated control device, like a remote controller, which is only able to send control data but does not receive the measured values. Additionally, there could be a second endpoint like a smartphone, which provides the data to the user who is able to control the heating system with the smartphone.

The second reason is more complex. If there is more than one connection to a specified gateway, working with different transport protocols or application protocols on different ports. In this case, each connection is handled like a connection to a different gateway. One scenario could be a microcontroller, connected to different measuring sensors (e.g. temperature, air pressure, humidity, etc.). The sensor sends the information to one specific gateway, but there is a dedicated application with a dedicated port working with the measured values. Of course there may be a bidirectional connection for control information as well.

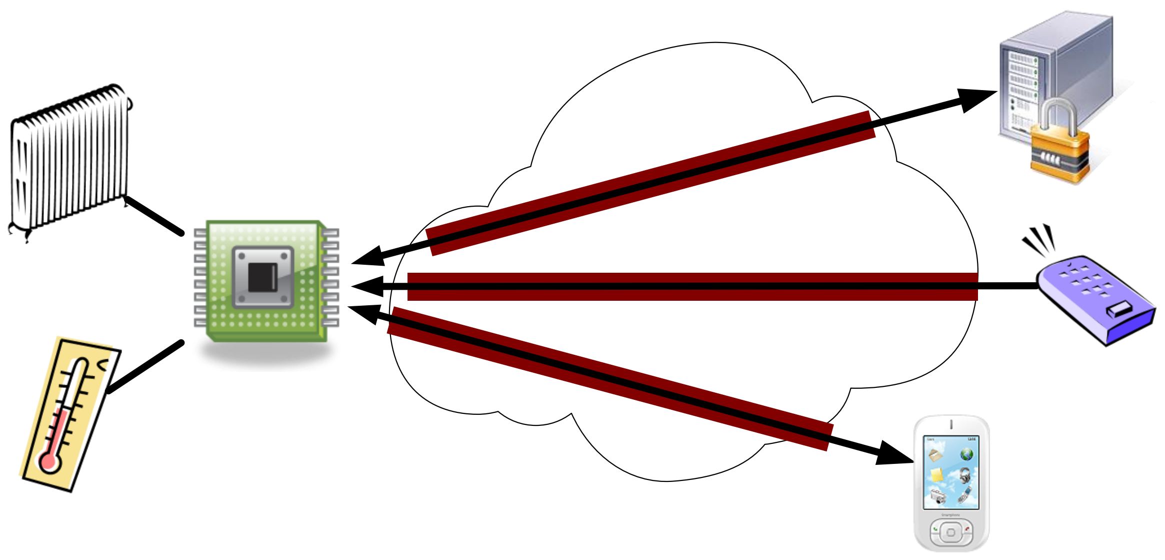


Figure 3.4 Uni– and bidirectional connections to multiple endpoints.

# 4 Requirements

The use cases show the networking functionality of a sensor network from the view point of a single sensor device. IPsec can be used to secure all of these further represented use cases. Since IPsec defines the two database SAD and SPD storing all related information for a unidirectional communication (use case 1 and 2), there are no issues with bidirectional (use case 3) or multidirectional connections (use case 4). In fact every unidirectional connection will have its one entry in the two databases, why a bidirectional connection will have two entries in each of the databases. Granjal et al [22] support this theory, stating that IPsec is a viable option to secure all kind of communication within the use cases.

In the remaining of this thesis, ESP is considered as the chosen security protocol for sensor, and investigated for IoT specific improvements. This section points out the requirements for the design of the ESP protocol for low powered devices and networks called Diet−ESP.

In IoT environments, networking itself comes with a lot of requirements for protocols, especially for security protocols. There are some hardware specifics, a security protocol should be able to deal with. Special ciphers (section 4.1) is an example, but also costs for networking which leads to requirements for alignment (section 4.2) and compression (section 4.3). Above all, a sensor is a very constrained device, why a protocol should not be too complex (section 4.5), but flexible (section 4.4), easy to use (section 4.6) and compatible to other sensor specific protocols (section 4.7). This section describes all of these requirements the Diet−ESP protocol should be able to fulfill.

## 4.1 Cipher Requirements

Due to its good security features, AES is the common used cipher algorithm for most security protocols, like ESP, AH or TLS [13: p. 74, 46]. A comparison of the different ciphers and modes can be found in [58].

AES can be used with different key length in 128, 192 and 256 bit and in four different modes. These modes are Cipher Block Chaining Mode (CBC [20]), Counter Mode (CTR [24]), Galois/Counter Mode (GCM [80]) and Counter with CBC−MAC (CCM [25]).

CBC and CTR are the common used modes. While CBC is only able to encrypt and decrypt fixed block length of 128 bit, the output produced by CTR has the same size as the input. Because of the fixed block length, CBC requires padding of the plaintext to exactly 128 bits. The two additional modes CCM and GCM use the same definition including an additional integrity check for the encrypted data. Both run encryption in Counter Mode (CTR), but CCM defines a CBC−MAC for the integrity check whereas GCM uses Galois MAC (GMAC).

In order to find the optimal cipher algorithm for an ESP implementation there are three different arguments. The first is the interoperability, which is ensured by implementing the ciphers defined in [46].

The second argument is an IoT specific one, where it is highly recommended to reduce the packet size. Since there is no valuable information included, padding is unnecessary data, sent on the wire. Table 4.1 compares AES in counter and block mode, but its results can be adapted to any other cipher algorithms. Any cipher running in block mode defines a block size for the plaintext. Input data without this block size has to be extended to it by using padding. In case of AES this block size is 16 bytes. Padding may also be necessary to fit the IP required alignment of 32 bits [61] (resp. 64 bits in IPv6 [11]). Usually the counter mode of a cipher algorithm works like a stream cipher and without the need a specific input size. Therefore all stream ciphers and ciphers in counter mode will have the same behavior as the one of AES−CTR illustrated in Table 4.1.

Interpreting the values in Table 4.1 one will see that CTR mode requires less or at least equal size of padding in every case. The packet size is less in every case, which is specific to AES−CTR because it defines an Initialization Vector (IV) of 8 byte whereas AES−CBC requires a 16 byte IV. However, even without considering the 8 byte less IV the packet size of AES−CTR is equal to AES−CBC in any case. Taking into account that AES−CTR requires an integrity check in form of the ICV one may argue that the overall packet size of AES−CTR plus a minimum ICV of 96 is higher than AES−CBC without ICV. But in that case AES−CTR delivers better security features than AES−CBC, because using NULL−ICV is not recommended.

The last argument for choosing the best cipher algorithm is the performance. Hardware implementation usually improves the performances compared to software implementation, as it reduces the number of operation to be done inside the processor. AES−NI [23] for example leads to only 4 processor instructions for every round of the AES algorithm.

IEEE 802.15.4 defines AES−CCM, for link layer security with upper layer key−management, thus it is often supported by hardware acceleration.

**REQ 1** In order to benefit from this hardware support, security protocols for sensors MUST support AES ciphers be able to take advantage of AES−CCM hardware acceleration.

**REQ 2** If encryption and authentication is enabled, a security protocol for sensors SHOULD be able to use AES−CCM as it is defined in IEEE 802.15.4 taking advantage of hardware acceleration for encryption and authentication.

| UDP Payload (Bytes) | Padding (Bytes) | | Packet Size IPv6/ESP/UDP (Bytes) | | | |
|---|---|---|---|---|---|---|
| | AES−CTR | AES−CBC | AES−CTR | AES−CBC | AES−CTR SHA1−96 | AES−CBC SHA1−96 |
| **1** | 1 | 5 | 68 | 80 | 80 | 96 |
| **2** | 0 | 4 | 68 | 80 | 80 | 96 |
| **3** | 3 | 3 | 72 | 80 | 84 | 96 |
| **4** | 2 | 2 | 72 | 80 | 84 | 96 |
| **6** | 0 | 0 | 72 | 80 | 84 | 96 |
| **8** | 2 | 14 | 76 | 96 | 88 | 112 |
| **16** | 2 | 6 | 84 | 96 | 96 | 112 |

**Table 4.1 Differences in Padding and Packet Size of an ESP packet sent with IPv6 when using AES in Block or Counter Mode.**

## 4.2 Alignment Requirements

IP extension headers need to have 32 bit Byte−Alignment in IPv4 (section 3.1 of [61] − Padding description) and a 64 bit Byte−Alignment in IPv6 (section 4 of [11]). As ESP [36] is such an extension header, padding is mandatory to meet the alignment constraint. This alignment is mostly caused by compiler and OS optimizations dealing with a 32 or 64 Bit processor. In the world of IoT, processors and compilers are highly specialized and alignment is often not necessary 32 Bit, but 16 or 8 bit. Second, ESP will often be the last protocol inside the IP header, because everything below will be encrypted and not changeable before the other peer decrypts the payload. In this case padding of the header is not necessary, since the IP−length defines the end of the ESP payload. The alignment may also be relevant if Block−Ciphers like AES−CBC needs an aligned payload to perform the encryption.

The RFC defines the Padding as a field with a variable length. RFC4835 [46] defines the mandatory encryption algorithm that has to be provided by an ESP implementation. This list defines the block cipher AES 128 as absolutely necessary, which means completely removing the size of Padding is only possible by sending exact 128 bits of application data. Padding is also necessary to fit the right alignment of the ESP trailer.

**Is it possible to use a fixed value?**
The RFC4303 defines a padding pattern, but that one is not mandatory. Therefore it would be possible to define a fixed padding value for a specific size of padding.

**Is it possible to reduce the Padding?**
The padding field cannot be reduced to a fixed size, because different ciphers or packet alignments need different paddings. On the other hand side, clever alignment of the payload data reduces this field itself. By limiting the used ciphers, the maximum size of this field is the maximum block size of the used block ciphers. The padding may be reduced by specifying special alignment restrictions for one communication.

**Is it possible to remove the Padding?**
Due to the wide range of the Padding field (0 – 255 bytes) removing seems to be ideal in order to limit the packet size. But the problem remains that different ciphers need different fixed block size to encrypt data and the right alignment of the ESP trailer has to be secured. In order to remove the padding either the alignment has to be specified as unnecessary or the application size must have the size of the alignment.

**REQ 3** Since networking is extremely expensive in IoT communications, padding MUST be prevented whenever it is possible. Therefore security protocols SHOULD support Byte−Alignment that are different from 32 bits or 64 bits to prevent unnecessary padding.

**REQ 4** In order to agree on the used alignment, each peer SHOULD be able to advertise and negotiate the Byte−Alignment, used for Diet−ESP. This could be done for example during the IKEv2 exchange.

## 4.3 ESP Compression

In order to understand how ESP related fields can be compressed, one has to understand the effect of the different ESP protocol fields. This section starts inspecting all fields described in RFC4303 [36]. Every field is inspected for potential fixed values, reducing or even removing the field from the protocol. With this information is used to deduce the requirements for ESP compression in section 4.3.7.

### 4.3.1 Security Parameter Index (SPI) (32 bits)

The SPI indexes the SA for incoming packets and is negotiated by the two peers (e.g. via IKEv2 or manually). It remains the same during the session and is chosen by the receiver. According to the RFC, the SPI is a mandatory 32 bits field and is not allowed to be removed. However, the SPI is arbitrary, thus it would be possible to use the IP or MAC address or a fixed value, except 1–255 because this values are reserved. The value 0 is only allowed to be used internal and it must not be sent on the wire.

*The SPI is an arbitrary 32–bit value that is used by a receiver to identify the SA to which an incoming packet is bound. The SPI field is mandatory. For a unicast SA, the SPI can be used by itself to specify an SA, or it may be used in conjunction with the IPsec protocol type (in this case ESP). Because the SPI value is generated by the receiver for a unicast SA, whether the value is sufficient to identify an SA by itself or whether it must be used in conjunction with the IPsec protocol value is a local matter. This mechanism for mapping inbound traffic to unicast SAs MUST be supported by all ESP implementations. [36: p. 9]*

**Is it possible to use a fixed value?**
Fixing the SPI means there is only one possible connection to each client. On the other hand side using a fixed value may be problematic if the number is given on the other side of the connection. One may chose the IP address of the receiver for the SPI value, concerning that only one SA can be identified using this functionality.

**Is it possible to reduce the SPI?**
Reducing the SPI means to reduce the possible maximum number of connections the receiver is able to deal with. As a consequence the security endpoint may be vulnerable to Denial of Service. When every possible SPI number is given, no new connection can be set up any more.

**Is it possible to remove the SPI?**
By removing the SPI a Security Association could be set up by exclusively using the IP address. Therefore the Traffic Selector inside the SPD database must be unique in order to identify the correct Security Association for incoming packets. That means there is only one possible connection to each client and IP mobility cannot be handled.

### 4.3.2 Sequence Number (SN) (32 bits)

The SN is used for anti–replay protection and is modified in every packet. In default cases, the ESP Sequence Number will be incremented by one for each packet sent. The field wants to be present in the packet, regardless if the receiver decides to use it for anti–replay or not. Additionally it is possible to use an Extended Sequence Number (ESN) of 64 bits. However the additional 32 bit are not sent on the wire and only incremented and stored within the SA.

*This unsigned 32–bit field contains a counter value that increases by one for each packet sent, i.e., a per–SA packet sequence number. For a unicast SA or a single–sender multicast SA, the sender MUST increment* *this field for every transmitted packet. Sharing an SA among multiple senders is permitted, though generally not recommended. [...] The field is mandatory and MUST always be present even if the receiver does not elect to enable the anti–replay service for a specific SA. [36: p. 11]*

**Is it possible to use a fixed value?**

Knowing that the receiver does not use anti–replay mechanism, fixing this value will prevent computing time. With a fixed Sequence Number, there is no protection for replay attacks in the IP layer. Concomitant with this, every packet has to be decrypted before a replayed packet can be recognized if the transport layer offers replay protection. This produces high computation overhead, especially on low powered devices like sensors.

According to the RFC the SN has not to be incremented by one necessarily. Therefore it would be possible to use another, always increasing and already existing value like the time as a Sequence Number.

**Is it possible to reduce the SN?**

As a consequence of reducing the SN the SA expires earlier. Therefore re–keying has to be done more frequently, what may produce computing overhead especially if many packets are sent for one connection. Mechanism like the Extended Sequence Number can be used to reduce the SN sent on the wire preventing reducing the SN stored on the receiver.

**Is it possible to remove the SN?**

Removing the Sequence Number has the same problems like fixing it. Additionally it prevents the storing of the Sequence Number on the sender.

### 4.3.3 Pad Length (8 bits)

Pad Length indicates the length of the Padding field and is computed on a per–packet basis. The RFC defines Pad Length as a mandatory 8 bit value which means it can neither be reduced nor removed.

*The Pad Length field indicates the number of pad bytes immediately preceding it in the Padding field. The range of valid values is 0 to 255, where a value of zero indicates that no Padding bytes are present. As noted above, this does not include any TFC padding bytes. The Pad Length field is mandatory. [36: p. 14]*

**Is it possible to use a fixed value?**

If the application payload is fixed sized, the padding is fixed and therefore the Pad Length, can be fixed, too.

**Is it possible to reduce the PL?**

The Pad Length cannot be reduced to a fixed size, because different ciphers or packet alignments need different paddings.

**Is it possible to remove the PL?**

The Pad Length is unnecessary to be sent in each packet as long as the size of padding is fixed or padding itself is not necessary. This fits to the sensor use cases because a fixed number of bytes is sent by the sensor in general. So the padding size is fixed by the application and it is unnecessary to send it on the wire.

### 4.3.4 Next Header (8 bits)

The Next Header indicates the protocol of the next layer inside the ESP Payload. The RFC defines the Next Header as a mandatory field of 8 bits.

*The Next Header is a mandatory, 8–bit field that identifies the type of data contained in the Payload Data field, e.g., an IPv4 or IPv6 packet, or a next layer header and data. [...] the protocol value 59 (which means "no next header") MUST be used to designate a "dummy" packet. A transmitter MUST be capable of generating dummy packets marked with this value in the next protocol field, and a receiver MUST be prepared to discard such packets, without indicating an error. [36: p. 15]*

**Is it possible to use a fixed value?**
A fixed Next Header assumes that it becomes impossible to have different protocols on the transport layer, which does not mean that the number of possible application protocols is reduced. Because sensor assume to use UDP instead of TCP this might be possible. On the other hand side the Tunnel Mode becomes impossible or the only used mode, because therefore the Next Header has to be IP.

**Is it possible to reduce the NH?**
On sensors the number of protocols is much less than on computer networks nowadays. Therefore reducing the Next Header to a lower value may be a possibility to reduce packet size.

**Is it possible to remove the NH?**
Removing the Next Header carries the same issues like a fixed value. In order to use different next headers the protocol inside ESP has to be negotiated uniquely otherwise, e.g. during the Key Exchange and stored inside the Traffic Selector.

### 4.3.5 Initialization Vector (IV)

The Initialization Vector is used to initialize the encryption cipher algorithm with the correct value what is necessary to decrypt correctly. The length and value of the Initialization Vector is defined in specific RFCs defining the Cipher algorithms for ESP. In case of AES there are different lengths defined for the use of AES in Block and Counter Mode [20, 24, 25, 80]. For CBC the IV inside the packet must be 16 bytes, which equates the size of the Initialization Vector for AES. For CTR the IV sent in the packet is only 8 byte. The remaining 8 byte for AES are built from the AES key and a counter, starting with 4 bytes of 0. In both cases the IV has to be unpredictable and unique for one key.

**Is it possible to use a fixed value?**
No, the IV must be unpredictable and unique for one communication key.

**Is it possible to reduce the IV?**
If uniqueness and unpredictability remains ensured, sending only an offset is possible. However the function calculating the IV on sender and receiver must always return the same value for one packet, whereas it needs accurate initialization and synchronization mechanism in case of lost or miss ordered packets.

**Is it possible to remove the IV?**
If uniqueness and unpredictability remains ensured, removing is possible, with the same assumption made for reducing the field.

### 4.3.6 Integrity Check Value (ICV)

The ICV secures the integrity of the ESP packet. It includes the ESP header (SPI, SN), the ESP payload (IV, original IP payload) and the ESP trailer (Padding, Pad Length and Next Header). Note that the ICV is built after the encryption occurred whereas the original IP payload and the ESP trailer are encrypted. According to the RFC the ICV is an optional value and its length is specified by the integrity algorithm/function which is used for a connection.

> *The Integrity Check Value is a variable–length field computed over the ESP header, Payload, and ESP trailer fields. Implicit ESP trailer fields (integrity padding and high–order ESN bits, if applicable) are included in the ICV computation. The ICV field is optional. It is present only if the integrity service is selected and is provided by either a separate integrity algorithm or a combined mode algorithm that uses an ICV. The length of the field is specified by the integrity algorithm selected and associated with the SA. The integrity algorithm specification MUST specify the length of the ICV and the comparison rules and processing steps for validation. [36: p. 16–17]*

However, the ICV can be mandatory in some cases, e.g. the encryption algorithm can require an authenticated IV value, which is the case for AES in Counter Mode.

> *Since it is trivial to construct a forgery AES–CTR ciphertext from a valid AES–CTR ciphertext, AES–CTR implementations MUST employ a non–NULL ESP authentication method. [24: p. 5]*

**Is it possible to use a fixed value?**
No, a fixed Integrity Check Value does not make any sense at all.

**Is it possible to reduce the ICV?**
The ICV can be reduced by using short size authentication algorithm, which may impact there security features. If one want to reduce the ICV by an external functionality, he has to ensure the correct rebuilding and point out security considerations due to the compression.

**Is it possible to remove the ICV?**
Without an integrity check, the encrypted data can be modified by interferences or attacks. Considering this fact it can be removed regardless if it is not required, by negotiating “NULL” as the ESP authentication method.

### 4.3.7 Compression Requirements

Sending data is very expensive regarding to power consumption, as illustrated in section 2.1. Compression can be performed at different layers but usually it is performed in the MAC layer. Since 6LoWPAN is a common standard to compress IP packets in the IoT that one should be used. Therefore Diet−ESP focuses on the encrypted ESP payload, which cannot be compressed inside the MAC layer.

**REQ 5** Diet−ESP SHOULD be able to reduce or remove all fields, directly related to the ESP protocol.

- **REQ 5.1** Diet−ESP SHOULD be able to remove the SPI if there is only one connection. If more than one connections is possible, Diet−ESP SHOULD be able to reduce the SPI to an appropriate value.
- **REQ 5.2** According to the frequency of sent packets, Diet−ESP SHOULD be able to remove the Sequence Number or reduce it to an appropriate value.
- **REQ 5.3** Padding SHOULD be absence according to REQ 3. In that case, the Pad Length field MUST be removable by Diet−ESP.
- **REQ 5.4** In many IoT scenarios UDP will be the only used protocol. Even if another transport layer protocol is used, it is usually unique at least for a specific connection. Therefore Diet−ESP SHOULD be able to remove the Next Header field from the ESP trailer.
- **REQ 5.5** Diet−ESP SHOULD be able to define a reduced size of the ICV independent of the used algorithm. Diet−ESP MUST NOT support completely removing the ICV, as NULL cipher is defined in RFC4303 for that case.

**REQ 6** Diet−ESP SHOULD allow compressions of upper layer protocols, e.g. protocols of the transport− or application layer.

**REQ 7** Diet−ESP SHOULD NOT allow compressed fields, not aligned to 1 byte in order to prevent alignment complexity.

**REQ 8** If a block cipher with an Initialization Vector (IV) is used (like AES) Diet−ESP SHOULD be able to minimize the value sent on the wire. If so, it MUST define a proper algorithm to remain uniqueness and unpredictability of the IV.

## 4.4 Flexibility

Diet−ESP can compress some of the ESP fields as Diet−ESP is optimized for IoT. Which fields may be compressed or not, highly depends on the scenario and all the current and future scenarios cannot been foreseen. Diet−ESP and ESP differs in the following point: ESP has been designed so that any ESP secured communication on any device is able to communicate with another one. This means that ESP has been designed to work for large Security Gateway under thousands of connections, as well as devices with a single ESP communication. Because, ESP has been designed not to introduce any protocol limitations, counters and identifiers may become oversized in an IoT context.

**REQ 9** The developer SHOULD be able to specify the maximum level of compression.

**REQ 10** Diet−ESP SHOULD be able to compress any field independent from another one.

**REQ 11** Diet−ESP SHOULD be able to define different compression method, when appropriated.

**REQ 12** Each peer SHOULD be able to announce and negotiate the different compressed fields as well as the used method.

## 4.5 Complexity

IoT devices have limited space for memory and storage.

**REQ 13** Diet−ESP SHOULD be able to be implemented with minimal complexity considering small implementation that implement only a subset of all Diet−ESP capabilities without requiring involving standard ESP, specific compressors and decompressors.

## 4.6 Usability

Application Developer usually do not want to take care about the underlying protocols and security. Standard IPsec addresses the goal by providing a framework that secures communication in any circumstances. Although application developers for IoT are expected to pay more attention to the device security and system requirements, they are not expected to be security developers. As a result, some default parameters that provides a standard secure framework for most cases should be provided. This is of course performed at the expense of some optimization, but it makes possible for application developers to have "standard" security and standard Diet−ESP compression by setting a single "DIET−ESP" flag. More advanced developers will be able to tune the security and compression parameters for their needs.

**REQ 14** Diet−ESP SHOULD provide default configurations, which can be easily set up by a developer.

## 4.7 Compatibility

IPsec/ESP is widely deployed by different vendors on different machines, why IoT devices may communicate with Standard ESP implementations.

**REQ 15** Diet−ESP SHOULD be able to interact with Standard ESP implementations on a single platform, without the need of implementing a second ESP stack inside the device.

**REQ 16** Diet−ESP SHOULD be able to communicate with Standard ESP gateways, by producing RFC4303 conform output.

6LoWPAN is the common used standard for compression in IoT connections and ROHC defines compression for mobile communications, like Voice over IP (VoIP). These compressions appear within the MAC layer enabling higher compression levels, but they have disadvantages if the payload is encrypted, like it is done in ESP. This behavior will be similar in all future compression methods within the MAC−Layer.

**REQ 17** In order to keep compatibility to other compression protocols appearing at the MAC layer, Diet−ESP SHOULD be able to interact with IP compression protocols, without the need of modifying them.

# 5 Related Work

Other work already defines compression of security protocols. ROHC and 6LoWPAN are able to reduce the overhead of the ESP protocol and parts of the encrypted payload. Even though, the compression rate is not really satisfying and leaves a lot of space for improvements, they give some good practices which can be adopted to Diet−ESP.

TLS and DTLS is another security protocol than ESP, but even there compression takes place and it shows that there is a high interest in compressing security protocols. In IoT environments, there is no actual difference between transport and IP layer, as the packet sizes are that small. However, TLS has to be developed for each application protocol, whereas IPsec can be used like a firewall securing all communication.

## 5.1 TLS and DTLS

There are several ideas implementing security on low−powered devices. In contrast to IPsec discussed in this thesis, there are a couple of investigations leading to TLS [13] (former known as SSL) securing the transport layer. As may be imagined, there are aspiration porting TLS on low−powered devices, too. Especially DTLS [71] is widely discussed providing best features to become the standard in securing the Internet of Things. In their paper "6LoWPAN Compressed DTLS for CoAP" [68, 70], Raza et al. discuss compressing DTLS with the already described 6LoWPAN header. The compression is especially defined for CoAP and they suggest four new header extensions for 6LoWPAN:

- **LOWPAN_NHC_R** compresses the *record header*, which is used in each packet secured with DTLS (see Figure 5.1). This header allows deletion of DTLS version. It also describes compression of the fields *Epoch* (16 to 8 bit) and *Sequence Number* (32 to 16 bit).
- **LOWPAN_NHC_RHS** compresses the TLS record header with additional information about the initial DTLS Handshake, usually described in the handshake protocol (see Figure 5.1). It uses the same compressions as *Record Only* with additional removing of the *fragment_offset* and *fragment_length* fields.
- **LOWPAN_NHC_CH** compresses the *ClientHello* header, used by the client to establish a connection to the server (see Figure 5.2). This header allows deletion of Session *Id, Cookie, Cipher Suite* and *Compression Method.* The DTLS version field is always deleted and the default DTLSv1.0 is used. If the server does not support the default version, it sends his version in the *ServerHello* and the client can decide if it is supported.
- **LOWPAN_NHC_SH** compresses the *ServerHello* header, used by the server to establish the connection (see Figure 5.3). This header allows deletion of the DTLS header fields *DTLS version*, *Session Id*, *Cipher Suite* and *Compression Method.*

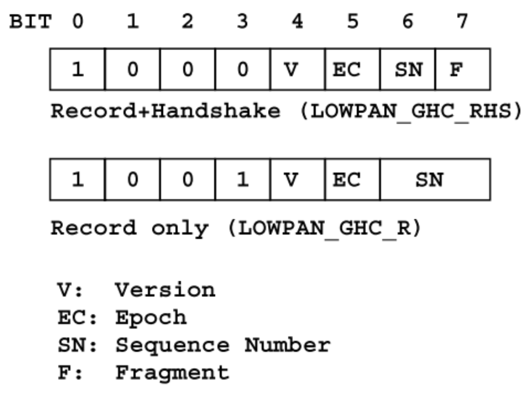


**Figure 5.1 LOWPAN_GHC_R(HS): 6LoWPAN compression for the TLS Record Header in optional combination with the Handshake header [70: p. 288].**

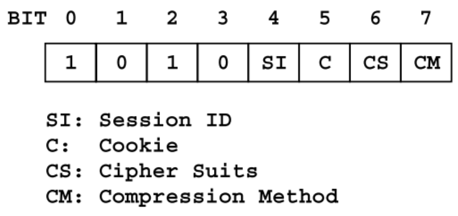


**Figure 5.2 LOWPAN_GHC_CH: 6LoWPAN compression for the TLS Client Hello header [70: p. 289].**

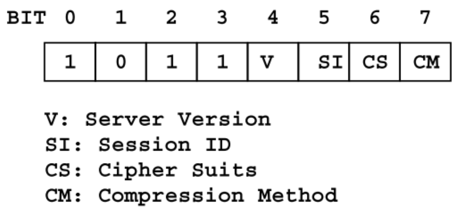


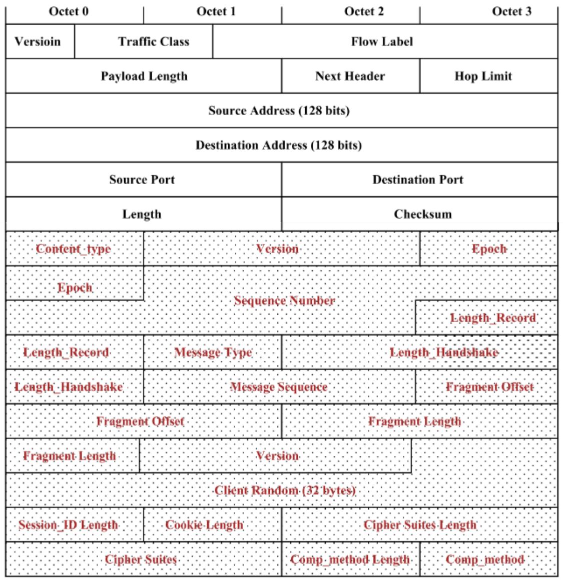
**Figure 5.3 LOWPAN_GHC_SH: 6LoWPAN compression for the TLS Server Hello header [70: p. 289].**

These compressions can extremely reduce the packet overhead because they reduce the record header, added to every packet from 104 to 40 bit (63% reduction). The handshake is reduced from 96 to 24 bit (75%), the *ClientHello* from 336 to 264 bit (23%) and the *ServerHello* from 305 to 264 bit (14%).

Using the compressions, the CoAP protocol can be secured with DTLS later called CoAPs or Lithe [69], including the compression mechanism. For setting up the DTLS connection Raza et al. use a lightDTLS implementation [51] which provides the mostly known ciphers. They set up their implementation on Contiki and show the possible maximum compression with their protocol as seen in Figure 5.4 and Figure 5.5.

Another idea for securing IoT communication with DTLS is simplifying the protocol without using compression, like discussed in [40]. By implementing the protocol in a light version, it is easy to communicate with traditional hard– and software because no adaptations are needed on the other side of the communication. Kothmayr et al. use X.509 certificates based on RSA to perform a two–way authentication. But it is exceptional because certificate based authentication needs additional computation. Additionally, asymmetric cipher–suites are more complex than symmetric ciphers. They performed it on sensors by adding special trusted notes to the network which acts as a proxy between usual certificates from the internet and sensor certificates.

However, there is a common disadvantage of DTLS against IPsec. The application protocol needs to implement and set up the security. Hence, every new application protocol appearing in IoT environments has to be specified for the use of DTLS resp. TLS. One can imagine that this leads to a lot of effort for the developers and may tempt them not implementing security. Additionally, the implementer of an IoT application needs to specify how the application should be secured. This does not fit to the usability requirement in section 4.6 defining that the application developer should be able to set a flag security and a standard security is provided for the whole device for all connections.



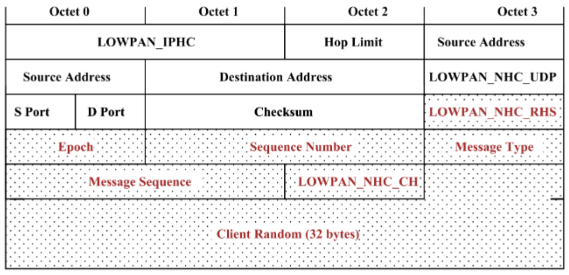


**Figure 5.4 UDP DTLS secured packet, without any compression [69: p. 3716].**

**Figure 5.5 with Lithe DTLS secured and compressed UDP message [69: p. 3716].**

## 5.2 RObust Header Compression

RObust Header Compression (ROHC [6] and ROHCv2 [56]) enables the compression of different protocols of all layers. It is designed as a framework, in which new protocol compressions appear as profiles. Usually ROHC compressions take place between IP and MAC layer, in which it does not take care on eventual IP alignment.

The general idea of ROHC is to classify the different protocol fields. According to the classification, they can be removed in general, removed after sent ones or reduced to some bits. Compressor and decompressor are holding a context defining the way of compression for a specific connection, enabling safe and robust recovering. The first packet to be sent is usually not or only minor compressed as it establishes this context. Following packets can be highly compressed. In order to negotiate the level of compression, ROHC defines different compressor states, messages and modes.

- **States:**
    - **Initialization and Request (IR) State:** In the IR–State, the static parts of the context are initialized or recovered at the decompressor context. The compressor sends complete header information, including all static and non–static fields together with some additional information.
    - **First Order (FO) State:** In the FO–State, irregularities in the packet stream are communicated between the two peers. The decompressor updates its context due to the information received in this state.

  - **Second Order (SO) State:** In the SO−State the compression is optimal. The decompressor enters this state, if the compressed header is completely predictable. This means that the context is filled with all needed information to decompress the packet without any other information. If an update of dynamic fields occurs, the decompressor switches back to the FO−State.
- **Modes:**
  - **Unidirectional (U) Mode:** Packets are sent in one direction only, from the compressor to the decompressor. Context changes are performed when the packet changes irregular, due to timeouts or periodically.
  - **Bidirectional Optimistic (O) Mode:** It is similar to the U−Mode, but it defines a feedback channel for error recovery between receiver and sender. On the other hand, periodically changes are not performed in the O−Mode.
  - **Bidirectional Reliable (R) Mode:** The R−Mode defines a more intensive use of the feedback channel and a stricter logic at compressor and decompressor. This can prevent the loss of packet synchronization and therefore maximizing the robustness of the compression.
- **Messages:**
  - **IR−Header:** It associates a Context−ID (CID) with a specific profile and usually initializes the context. It can also update the context.
  - **IR−Dyn−Header:** In contrast to the IR−Header it can never initialize an uninitialized context, but update the profile or initialize or update the dynamic parts of the context.

Since ESP may contain encrypted data, it is complex to define compressions for the encrypted payload. Therefore ROHC defines different compressions of the ESP protocol (see Figure 5.6). Regular ROHC can compress the ESP header only. If the packet is not encrypted, the rate of compression is extremely high because the whole packet including padding can be compressed with the regular ROHC stack as well. However, ESP is designed for confidentiality, whereas encryption is a common use case. ROHCoverIPsec [15–17] defines a method to compress the ESP payload before it is going to be encrypted. This leads to a second ESP stack, in which another ROHC compressor (resp. decompressor) works. Excluding the first packet which initializes the ROHC context, this enables a high compression rate. But it leads to two different compression and IPsec implementations, which is usually heavy for sensors as they may have restricted memory. In addition ROHC is a very complex compression protocol, not suitable for the constrained devices in IoT. A developer may choose to simplify the ROHC and ROHCoverIPsec implementation. But, if one decides to implement only one of the two ROHC stacks he will lose the compression of the other stack. If he decided to implement only one IPsec stack, he will lose compatibility to not ROHCoverIPsec compliant gateways, which makes the protocol not flexible but complex and this should be avoided according to the requirements (see section 4.5 and REQ 7).

Unfortunately, ROHCoverIPsec also limits the compression of the ESP protocol as it is asked in the requirements (see section 4.3.7), because of the IP restrictions. Padding remains necessary as IPsec is part of the IP stack which requires a 32 bits (resp. 64 bits for IPv6) aligned packet.

With some modifications, ROHC would be able to compress even encrypted packets without a second ESP stack. Encrypted fields could be removed at the sender and rebuilt, encrypted and reinserted by the receiver. This could be achieved by clever context definitions at the beginning of the communication, but it has some restrictions. First of all, erasure can only be done in fixed block size, a cipher algorithm may be required (e.g. 16 Byte in AES–CBC). Additionally, it leads to a high effort, without guaranteeing the compression to work safely. Since the receiver has to encrypt the compressed information and add them to the packet before it is able to check the integrity of the packet, one can easily construct a DoS attack. Flooding the receiver with invalid packet causes the receiver to perform the complex encryption and authentication algorithm for each packet, what could overcharge him.

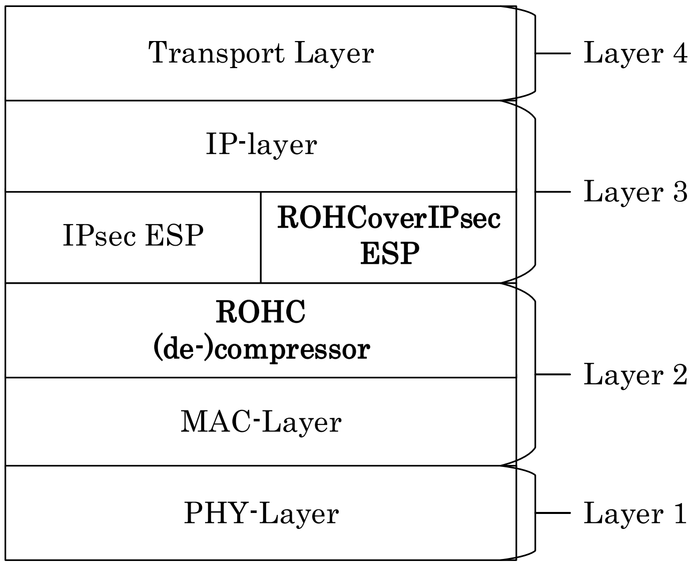


**Figure 5.6 The two different ROHC layers in the OSI model.**

## 5.3 6LoWPAN

The 6LoWPAN standard described in section 2.1.2 can be extended for compression of other protocols than IPv6 like ESP, AH. Raza et al. [66, 67] and Rantos et al. [64] describe such an extension for the ESP and AH header (see Figure 5.7 and Figure 5.8). Using all compression the AH header can be reduced from minimum 24 bytes to 18 bytes (see Figure 5.9) and the ESP header can be decreased by 6 bytes (see Figure 5.10). Referencing Raza et al., the packet size overhead of compressed IPsec to the 6LoWPAN link layer security is between 3 and 9 Bytes depending on the required security parameters (authentication, encryption, cipher algorithm, authentication + encryption). In addition to Raza et al., Rantos et al. explicitly describe the use of AES–CCM* like requested in the requirements.

```
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 1 | 1 | 0 |SPI| AI| SN| PD| NH|
+---+---+---+---+---+---+---+---+
```

**Figure 5.7 LOWPAN_NHC_ESP: The ESP Header extension for 6LowPan [64: p. 61].**

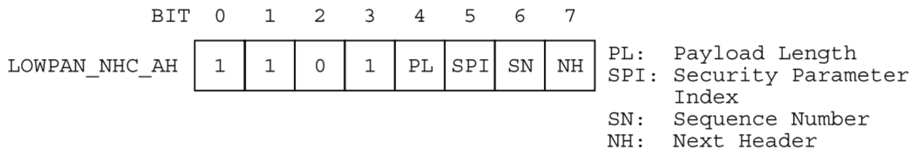


**Figure 5.8 LOWPAN_NHC_AH the extension header for 6LoWPAN for compressing IPsec Authentication Header protocol [66: p. 4].**

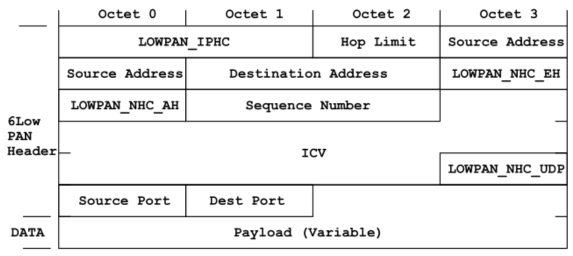


**Figure 5.9 LOWPAN_AH: IPsec Authentication Header secured UDP Payload with 6LoWPAN compression [65: p. 15].**

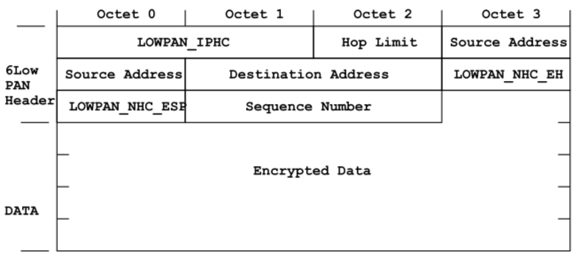


**Figure 5.10 LOWPAN_ESP: IPsec ESP secured UDP Payload using 6LoWPAN compression [65: p. 15].**

However, for ESP the same problem as with ROHC appears, that it is not possible to compress the encrypted ESP payload. Therefore Raza et al. introduce a second IPsec stack, similar to ROHCoverIPsec, for the compression of the inner protocols like UDP or IP in IPsec Tunnel mode. That makes 6LoWPANoverIPsec inflexible, incompatible to regular ESP and complex to implement, which is asked to be prevented in the requirements.

Of course one can develop similar functionality as described in the ROHC section above to compress encrypted payload, but he has to deal with the same restrictions.

## 5.4 Comparison with requirements

The two protocols ROHC and 6LoWPAN are able to compress not encrypted IP packets between the MAC and IP layer including the not encrypted ESP header. With specific extension they can also compress the encrypted IP, Transport and Application headers inside the ESP packet, by implementing a second or modified ESP procession. This enables high compression of the headers, but not of the ESP trailer and the encryption information, like the AES Initialization Vector. Table 5.1 shows which requirements are fulfilled by these two protocols.

One can see that the alignment, IV and ICV compression requirements cannot be fulfilled, even by combining two protocols (e.g. ROHC and ROHCoverIPsec or 6LoWPAN and 6LoWPANoverIPsec). Additionally, both protocols need to be implemented according to a special use case, which makes them not easy to be configured (REQ 14). As both ESP payload compressions need a second compression layer together with a second ESP stack, they cannot be implemented in a light way (REQ 13). Moreover, the ROHC framework includes that much compression negotiations that it cannot be implemented in a light way at all. Therefore, ROHC and ROHCoverIPsec can be seen as not useable for IoT communications. 6LoWPAN may be a good choice for basic compressions, but it is not able to fulfill the complete set of compression requirements (REQ 5.1 – REQ 5.5) and upper layer compressions are quite complex (REQ 6).

| Requirement | ROHC | 6LoWPAN | ROHCover IPsec | 6LoWPAN overIPsec |
|---|---|---|---|---|
| REQ 1: Support AES | ✓ | ✓ | ✓ | ✓ |
| REQ 2: Support AES–CCM | ✓ | ✓ | ✓ | ✓ |
| REQ 3: Different Alignments | ✓ | ✓ | X | X |
| REQ 4: Alignment Negotiation | X | X | X | X |
| REQ 5.1: SPI compression | ✓ | ✓ | X | X |
| REQ 5.2: SN compression | ✓ | ✓ | X | X |
| REQ 5.3: PL removal | X | X | X | X |
| REQ 5.4: NH removal | X | X | X | X |
| REQ 5.5: ICV compression | X | X | X | X |
| REQ 6: Upper Layer compression | X | X | ✓ | ✓ |
| REQ 7: 1 Byte alignment | X | ✓ | X | ✓ |
| REQ 8: IV compression | X | X | X | X |
| REQ 9: Negotiation of compression level | ✓ | ✓ | ✓ | ✓ |
| REQ 10: Independence of field compressions | ✓ | ✓ | ✓ | ✓ |
| REQ 11: Different compression methods | ✓ | ✓ | ✓ | ✓ |
| REQ 12: Negotiation of compressed fields | ✓ | X | ✓ | X |
| REQ 13: Minimal implementation | X | ✓ | X | X |
| REQ 14: Default configuration | X | X | X | X |
| REQ 15: Implementable as ESP–Add–On | X | X | X | X |
| REQ 16: Supports Standard ESP compatible compression level | ✓ | ✓ | ✓ | ✓ |
| REQ 17: Compatible with other MAC–Layer compressions | ✓ | ✓ | ✓ | ✓ |

Table 5.1 Comparison of existing ESP compression protocols due to the requirements.

# 6 Diet−ESP Protocol Design

This section points out the design of Diet−ESP. It provides instruments to compress fields of the standard ESP. Section 6.2 and 6.6 will describe the Diet−ESP context and how the protocol works. Section 6.3 addresses the compression of the Clear Text Data. In fact, Clear Text Data usually includes application protocols, transport protocols like UDP/TCP and sometimes IP headers when the IPsec Tunnel mode is used. Some of their parameters could be compressed as they are negotiated out of band, and are stored in the IPsec databases. Other parameters may also be compressed by using specific methods like ROHC. Section 6.4 addresses how the IV can be computed on both peers, thus reducing the need to send the complete IV in each packet. Section 6.5 defines how all necessary parameters for Diet−ESP can be negotiated using IKEv2. The requirements ask Diet−ESP to be able to be integrated to an existing ESP implementation in an easy way. Section 6.7 points out the special characteristics of the Diet−ESP protocol and shows how it can be integrated to a Minimal−ESP implementation as described in section 6.6. How Diet−ESP interacts with other compression protocols is outlined in section 6.8.

## 6.1 The use of the ROHC context

The compression mechanisms of Diet−ESP are based on ROHC and ROHCoverIPsec. ROHC defines mechanisms to compress and decompress fields of an IP packet. They have been designed for bandwidth optimization, but not necessarily for constrained devices. As a result, defining ROHC and ROHCoverIPsec profiles is not sufficient to fulfill the complete set of Diet−ESP requirements. Diet−ESP will result in a light implementation that does not require implementation of the full ROHC and ROHCoverIPsec.

In order to achieve this goal, the Diet−ESP Context contains all necessary parameters to compress an ESP packet and the ROHC and ROHCoverIPsec framework is used to compress the ESP packet. More specifically, it describes how the ROHC context and profile can be derived from the Diet−ESP Context to proceed to the compression. The advantage of using ROHC and ROHCoverIPsec is that compression behavior follows a standardized compression framework. On the other hand, deriving profiles and methods from the Diet−ESP Context makes the use of ROHC and ROHCoverIPsec implementation dependent, and any implementation that behaves in a similar way will be interoperable. This makes light implementation for Diet−ESP for constrained devices achievable.

## 6.2 Diet−ESP Context

The Diet−ESP context provides the necessary parameters for the compressor and decompressor to perform the appropriate compression and decompression of the ESP packet. Table 6.1 shows and section 6.2.1 describes the different parameters. These parameters provide the generic Diet−ESP Context. Section 6.2.2 provides the Diet−ESP Context that makes Diet−ESP compliant with ESP. Finally Section 6.2.3 provides the default Diet−ESP Context. These are the expected values when the Diet−ESP Context is not specified. The goal of the default Diet−ESP Context is to simplify the use of Diet−ESP as a secure framework for IoT communications.

### 6.2.1 Context Description

The Diet−ESP context is shown in Table 6.1. In the following, the fields of the context are described and their characteristics are discussed.

| Context Field Name | Overview |
|---|---|
| ALIGN | Necessary Alignment for the specific device. |
| SPI_SIZE | Size in bytes of the SPI field in the transmitted packet. |
| SN_SIZE | Size in bytes of the SN field in the transmitted packet. |
| NH | Presence of the Next Header field in the ESP Trailer. |
| PAD | Presence of the Pad Length field in the ESP Trailer. |
| Diet−ESP_ICV_SIZE | Size of the Diet−ESP ICV in the transmitted packet. |

**Table 6.1 Diet−ESP Context.**

#### 6.2.1.1 ALIGN

Alignment is the minimum alignment accepted by the hardware. Constraints may come from various reasons (see section 4.2). Diet−ESP reduces the ALIGN value from 32 bits for IPv4 or 64 bits for IPv6 to 8, 16, 32 and 64 bit alignment.

The motivation to do so is to remove the padding and other mandatory fields of the ESP packet. Then, many IoT devices embed small 8 or 16 bit CPUs. Finally, even though ESP is an extension header, it is often the last extension header of a header−only IP packet. The ESP header is only read by the real receiver and is uninteresting for other devices like routers, placed between the two peers. As a result, there seems to be no real impact on the system if ESP extension header is not aligned.

Note that the benefices of ALIGN also depends on the used cryptographic mode. More specifically AES−CTR has an 8 bit block whereas AES−CBC has a 128 bit block. As a result the use of AES−CBC with small Clear Text Data results in large Encrypted Data with embedded padding. In other words, the alignment for one packet is always $MAX(CIPHER\_BLOCK\_SIZE, ALIGN)$.

#### 6.2.1.2 SPI_SIZE

ESP Security Policy Index is 4 byte long to identify the SAD−entry for incoming traffic. Diet−ESP omits, leaves unchanged or reduces the SPI sent on the wire to the 0, 1, 2, 3 or 4 LSB (Least Significant Bytes).

Compressing the SPI can have security impact, since the inbound SAD−lookup has to be changed. In some variations, this lookup may result in Denial of Service (DoS) attacks, if the device is not configured properly. Therefore this number should be guided by the number of simultaneous inbound SA the device is expected to handle and reliability of the IP addresses in order to identify the proper SA for incoming packets. A sensor with a single connection to a Security Gateway may bind incoming packets to the proper SA based only on its IP addresses. In that case the SPI is not used for SAD−lookup but for integrity check. Other scenarios may consider using the SPI to index the SAs or having multiple ESP channels with the same host from a single host. In that case a reduced length for the SPI may be chosen. Note also that the value 0 for the SPI is note allowed to be sent on the wire as described in [36], since it is used to separate ESP and IKEv2 traffic within NAT environments.

#### 6.2.1.3 SN_SIZE

ESP Sequence Number is 32 bit and extended SN is 64 bit long and used for anti−replay protection. Diet−ESP omits, leaves unchanged or reduces SN sent on the wire to 0, 1, 2, 3 or 4 LSB.

Decompressing the SN at the receiver is guided by a linear extrapolation of the expected received Sequence Number and the LSB−SN sent on the wire. To avoid packet overhead, this configuration is stored within the Security Association, whereas it remains valid during its lifetime. An implementer has to avoid sending SN increasing higher than the maximum value of the LSB window.

In some cases the received SN may increase by a higher number than one e.g. if the time is used as the SN or because of a high number of packet loss. If one decides to use a mechanism like the time, one has to deal with the following restrictions:

1) The SN_SIZE is not allowed to be 0
2) If the SN_SIZE is not 4:
    2.1) The SN_SIZE must be chosen in a way that the LSB of the time is not allowed to overlap between the sending of two packets.
    2.2) The start value of the SN must be ensured to be the same between the two peers. This can be done with time synchronization for example.

Note that SN and SPI must be aligned to a multiple of the alignment value (ALIGN).

#### 6.2.1.4 NH

Next Header in ESP is used to identify the first header inside the ESP payload. Diet−ESP is able to remove the Next Header field from the ESP−Trailer.

Removing the Next Header is only possible if the underlying protocol can be derived from the Traffic Selector (TS) within the Security Association (SA). The Next Header indicates whether the encrypted ESP payload is an IP packet, a UDP packet, a TCP packet or no next header. The NH can only be removed if this has been explicitly specified in the SA or if the device has a single application.

Note that removing the Next Header impacts how encryption is performed. For example, the use of AES−CBC mode requires the last block to be padded, reaching a 128 bit alignment. In this case removing the Next Header increases the padding by the Next Header length, which is 8 bits. In this case, removing the Next Header provides a few advantages, as it does not reduce the ESP packet length. With AES−CBC, the only advantage of removing the Next Header would be for data with the last block of 15 bytes. In that case, ESP pads with 15 modulo 16 bytes, set the 1 byte pad length field to 15 and add the one byte Next Header field. This leads to 15 + 15 + 1 + 1 = 32 bytes to be sent. On the other hand, removing the Next Header would require only the concatenation of the pad length byte with a 0 value, which leads to only 16 bytes to be sent.

Other modes like AES−CTR do not have block alignment requirements. Using AES−CTR with ESP only requires the IP alignment. In fact, if an n byte alignment is required (for encryption or for packet format), data of length k * n + n − 1 bytes, where k is an integer, takes advantage of removing the Next Header and reduces the data to be sent over n bytes. In the case of sensor network it is very likely that data of fixed size k * n + n − 1 will be used. Furthermore, if IP alignment is reduced to 8 bits alignment, Next Header is always an additional unnecessary byte being sent.

#### 6.2.1.5 PAD

With ESP, all packets have a Pad Length field. This field is usually present because ESP requires IP alignment which is ensured with padding. Diet−ESP considers removing the Padding and the Pad Length fields. If PAD is present, it is computed according to ALIGN.

Some devices might use an 8 bits alignment and in that case padding is not necessary. Similarly, sensors may send application data of fixed length matching the alignment. With ESP these scenarios would result in an unnecessary Pad Length field always set to zero. Diet−ESP considers those cases with no padding and thus the Pad Length field can be omitted.

#### 6.2.1.6 Diet−ESP_ICV_SIZE

Integrity Check Value (ICV) is used to authenticate the Diet−ESP Payload. Diet−ESP considers sending the whole ICV or the first 1 byte (resp. 2, 4, 8 bytes).

ESP negotiates an authentication protocol for every SA. These protocols generate an ICV of a length defined by the authentication protocol. These definitions do not provide ways to perform weak authentication, as there is no way to reduce the size of the ICV. IoT is interested in weak authentication as it may send a small amount of bytes, and the trade−off between battery life time and security may be worth. As a result Diet−ESP indicates the number of bytes of the ICV. Note that reducing the size of the ICV may expose the system to security flaws. Note also that ICV is optional so if one chooses not to perform authentication, one must negotiate the authentication algorithm to NULL as defined in [46].

The Diet−ESP ICV value differs from the Standard ESP value since the authenticated data is not the same. In the case of the Diet−ESP ICV, the ICV is computed over the compressed Diet−ESP payload, whereas in the case of the Standard ESP ICV the ICV is computed over the uncompressed packet. This means that the decompressed Diet−ESP ICV is not expected to match the Standard ESP ICV value (see section 6.7.2 for more details).

### 6.2.2 Standard ESP compliant Diet−ESP Context

Table 6.2 defines the Diet−ESP Context that produces regular ESP packets. This makes Diet−ESP compatible with standard ESP.

| Context Field Name | Default Value |
|---|---|
| ALIGN | IP alignment (4 bytes for IPv4 and 8 bytes for IPv6) |
| SPI_SIZE | 4 bytes |
| SN_SIZE | 4 bytes |
| NH | Present |
| PAD | Present |
| Diet−ESP_ICV_SIZE | Not compressed |

**Table 6.2 Diet−ESP Context for regular ESP.**

- ALIGN (IP alignment): The alignment is 32 bit for IPv4 and 64 bit for IPv6.
- SPI_SIZE (4 bytes): This is not problematic for devices that handle a few simultaneous connections.
- SN_SIZE (4 bytes): This causes a window of $2^{32}$ lost packets for a regular incrimination of 1 for each packet.
- NH (present): Next Header remains uncompressed, as compression cannot be performed in all scenarios.
- PAD (present): Padding is computed according to the IP alignment.
- Diet−ESP_ICV_SIZE (not compressed): The Diet−ESP ICV is computed. The length is determined by the authenticating algorithm, negotiated by the SA. This value is not truncated.

### 6.2.3 Default Diet−ESP Context

This section defines a default Diet−ESP Context. IPsec/ESP has been designed to provide a secure framework that remains secure in any scenarios. As a result, specific scenarios may carry unnecessary security parameters. As compression is performed on a per−scenario basis, the Diet−ESP Context configuration for a given scenario may introduce vulnerabilities in another scenario. As a result, Diet−ESP can hardly define an optimized Diet−ESP Context that matches with any scenarios.

On the other hand, Diet−ESP is very flexible and makes it possible to compress any fields. Defining which field can be compressed without introducing vulnerabilities requires specific security knowledge not all application developer are expected to have. Instead, an application developer should be able to simply mention that a given application needs to be secured with Diet−ESP without specifying any parameters. In that case, a Default Diet−ESP Context will be considered. This Diet−ESP Context is probably not optimized for the given scenario, but at least it does not introduce vulnerabilities. The values for the Default Diet−ESP Context are specified in Table 6.3.

| Field Name | Default Value |
|---|---|
| ALIGN | 8 bit Alignment |
| SPI_SIZE | 2 bytes |
| SN_SIZE | 2 bytes |
| NH | Present |
| PAD | Removed |
| ESP_ICV_SIZE | Not compressed |

**Table 6.3 Diet−ESP default context.**

- ALIGN (8 bit): As most IoT devices CPUs are likely to deal with 8 bit alignment.
- SPI_SIZE (2 bytes): The SPI is reduced to 2 bytes. This is not problematic for devices that handle a few simultaneous connections.
- SN_SIZE (2 bytes): The SN reduced to 2 bytes which causes a window of 512 lost packets for a regular incrimination of 1 for each packet.
- NH (Present): Next Header remains present in the packet, as compression cannot be performed in all scenarios.
- PAD (Removed): Padding is removed as AES−CTR (AES−CCM*) is widely deployed for IoT, and most IoT devices can deal with 8 bit alignment. If AES−CBC is used, the padding is performed by the encryption mode itself.

- Diet–ESP_ICV_SIZE (not compressed): The Diet–ESP ICV is computed. The length is determined by the authenticating algorithm negotiated by the SA. This value is not truncated in order to remain the security provided by the algorithm.

## 6.3 Diet–ESP Extension: Upper Layer Compressions

The ESP payload includes transport layer (TCP [62], UDP [60], UDP–Lite [41]) and application layer header (HTTP [19], RTP [72], etc.). Inside the Diet–ESP framework, these upper layer headers can be compressed by any compression protocol, like 6LoWPAN or ROHC. However, both of them need additional information to be stored on the device (e.g. ROHC context) or information sent on the wire (6LoWPAN header or ROHC signaling). Therefore, this section introduces a Diet–ESP context extension for compression of transport layer headers.

The main propose of transport protocols is the definition of the port where the packet should be sent to, identifying the program at the endpoint which is used for working with this packet. For (de–) compressions, one has to know the source and destination ports together with the transport layer protocol. All these information can already be found in the IPsec Security Association (SA), if it is established uniquely. If IPsec is working in Tunnel Mode, the ESP payload includes an IP [11, 61] header. Like for transport layer protocols, most information for (de–) compression of this header, can be found inside the SA or in the original IP header by applying mechanism like the ones used in the IPsec BEET mode described in section 2.2.1.

### 6.3.1 Diet–ESP Context Extension

This section provides the extension for the Diet–ESP context to enable inner header compressions shown in Table 6.4.

| Context Field Name | Overview |
|---|---|
| **USE_IPSEC_DATABASE** | Defines the use of the Traffic Selector for (de–) compression. |
| **INNER_ROHC_COMPRESSION_PROFILE** | Defines the ROHC profile used for compression. |
| **CHECKSUM_LSB** | LSB of the UDP, UDP–Lite or TCP checksum |

**Table 6.4 Diet–ESP context extension for upper layer header compressions.**

**USE_IPSEC_DATABASE:**

The IPsec SAD holds some values which can be used for compressing a packet, but only if the values are static. If a developer chooses to use the context field USE_IPSEC_DATABASE, he must ensure that all necessary values for all profiles specified in INNER_ROHC_COMPRESSION_PROFILE are unique.

**INNER_ROHC_COMPRESSION_PROFILE:**

ROHC defines different profiles for protocols and protocol combinations [27]. The Diet–ESP context parameter INNER_ROHC_COMPRESSION_PROFILE specifies the profile(s) used for one session associated with one IPsec security association. Table 6.5 shows an overview over the existing ROHC profiles during the time of writing this thesis.

**CHECKSUM_LSB:**

If an inner header provides a checksum this can be compressed by the LSB mechanism. The way the checksum is compressed is specified by the related profiles, e.g. UDP Section 6.2, UDP−Lite Section 6.3 and TCP Section 6.4.

| Profile Number | ROHC version | Protocol | RFC |
|---|---|---|---|
| 0x0000 | ROHC | uncompressed IP | RFC3095[6] |
| 0x0001 | ROHC | RTP/UDP/IP | RFC3095[6] |
| 0x1001 | ROHCv2 | RTP/UDP/IP | RFC5225[56] |
| 0x0002 | ROHC | UDP/IP | RFC3095[6] |
| 0x1002 | ROHCv2 | UDP/IP | RFC5225[56] |
| 0x0003 | ROHC | ESP/IP | RFC3095[6] |
| 0x1003 | ROHCv2 | ESP/IP | RFC5225[56] |
| 0x0004 | ROHC | IP | RFC3843[31] |
| 0x1004 | ROHCv2 | IP | RFC5225[56] |
| 0x0006 | ROHC | TCP/IP | RFC6846[57] |
| 0x0007 | ROHC | RTP/UDP−Lite/IP | RFC4019[55] |
| 0x1007 | ROHCv2 | RTP/UDP−Lite/IP | RFC5225[56] |
| 0x0008 | ROHC | UDP−Lite/IP | RFC4019[55] |
| 0x1008 | ROHCv2 | UDP−Lite/IP | RFC5225[56] |

**Table 6.5 Overview over currently existing ROHC profiles.**

### 6.3.2 Overview

The Diet−ESP context extensions defined in section 4 have specific requirements for the values stored in the Security Association. Roughly speaking, Diet−ESP is able to remove all header fields which have unique values inside the Security Association. Most probably they are stored in the Traffic Selector, which defines the traffic to be secured with IPsec. Table 6.6 shows some header fields which can be adopted from the Traffic Selector. Table 6.7 shows the compressed headers for the specific profiles, currently supported by Diet−ESP. Negotiating ROHC profile 0x0000 provides the same functionality as negotiating USE_IPSEC_DATABASE with “no”.

| Field | Protocol | ROHC class |
|---|---|---|
| IP version | IP/IPv6 | STATIC−KNOWN |
| Source Address | IP/IPv6 | STATIC−DEF |
| Destination Address | IP/IPv6 | STATIC−DEF |
| Next Header | IP/IPv6 | STATIC |
| Source PORT | UPD/TCP | STATIC−DEF |
| Destination PORT | UPD/TCP | STATIC−DEF |

**Table 6.6 Required fields inside the IPsec Security Association.**

Diet−ESP considers none of the ROHC messages to establish the context between the two peers. The ROHC context is defined during the Key Exchange, which establishes the Diet−ESP context, before the first packet is sent. If the USE_IPSEC_DATABASE parameter is set to 'yes', all relevant values defined by the profile specified in INNER_ROHC_COMPRESSION_PROFILE are linked with the correlating values in the Security Association.

| Profile Number | Protocol | Diet-ESP compression |
|---|---|---|
| 0x0000 | uncompressed IP | no compression |
| 0x0001 | RTP/UDP/IP | not used |
| 0x1001 | RTP/UDP/IP | not used |
| 0x0002 | UDP/IP | UDP and IP in Tunnel Mode |
| 0x1002 | UDP/IP | UDP and IP in Tunnel Mode |
| 0x0003 | ESP/IP | not used |
| 0x1003 | ESP/IP | not used |
| 0x0004 | IP | IP in Tunnel Mode |
| 0x1004 | IP | IP in Tunnel Mode |
| 0x0006 | TCP/IP | TCP and IP in Tunnel Mode |
| 0x0007 | RTP/UDP-Lite/IP | not used |
| 0x1007 | RTP/UDP-Lite/IP | not used |
| 0x0008 | UDP-Lite/IP | UDP-Lite and IP in Tunnel Mode |
| 0x1008 | UDP-Lite/IP | UDP-Lite and IP in Tunnel Mode |

**Table 6.7 Interaction of ROHC profiles with Diet−ESP compression.**

### 6.3.3 Upper Layer Header Compression Details

This section describes the details, how the different protocols can be compressed using Diet−ESP upper layer compression. If compression occurs, one of the ROHC profile number listed in [27] is specified in INNER_ROHC_COMPRESSION_PROFILE. Note that IP compression only takes place if IPsec Tunnel mode is used for the Security Association.

Later, this section points out the compression methods for all protocol headers which can be compressed by using ROHC profiles and how Diet−ESP uses them.

#### 6.3.3.1 IP (profile 0x0001 − 0x1008)

This section shows the compression of ESP payload for all ROHC profiles including an IP header. This is only relevant if IPsec is used in Tunnel Mode and it only affects the inner IP header. How the different header fields are compressed is provided in Table 6.8. Note that in IoT usually IPv6 is used instead of IPv4, therefore this section focuses only on IPv6, but the adoptions can be made on IPv4 as well.

Outer IP specifies values, which can be read from the IP packet, holding the ESP header. This one is not compressed by Diet−ESP and Diet−ESP must have access to uncompressed values. This mechanism is proven to work in IPsec BEET mode [47].

| Field | Class | Compression Method | Diet–ESP ROHC class | Data origin |
|---|---|---|---|---|
| Version | STATIC | removed | STATIC | TS |
| Traffic Class | CHANGING | removed | INFERRED | outer IP |
| Flow Label | STATIC–DEF | removed | INFERRED | outer IP |
| Payload Length | INFERRED | removed | INFERRED | outer IP |
| Next Header | STATIC | removed | STATIC | TS |
| Hop Limit | RACH | removed | INFERRED | outer IP |
| Source Address | STATIC–DEF | removed | STATIC–DEF | TS |
| Destination Address | STATIC–DEF | removed | STATIC–DEF | TS |

**Table 6.8 Header classification for IPv6.**

**Version:**
The IP version is specified in the SA and can be copied to the ROHC context, before the first packet is sent/received.

**Traffic Class:**
Traffic Class can be read from the outer IP header. Therefore the classification is changed to INFERRED.

**Flow Label:**
Flow Label can be read from the outer IP header. Therefore the classification is changed to INFERRED.

**Next Header:**
The Next Header is stored in the protocol of the Traffic Selector and is fixed. It can be copied to the ROHC context, before the first packet is sent/received.

**Hop Limit:**
The Hop Limit can be read from the outer IP header. Therefore the classification is changed to INFERRED.

**Source Address:**
The Source Address is fixed in the SA and can be copied to the ROHC context, before the first packet is sent/received.

**Destination Address:**
The Destination Address is fixed in the SA and can be copied to the ROHC context, before the first packet is sent/received.

#### 6.3.3.2 UDP (profile 0x0001, 0x1001, 0x0002, 0x1002)

This section shows the compression of ESP payload for all ROHC profiles including an UDP header listed in Table 6.9.

| Field | Class | Compression Method | Diet−ESP ROHC class | Data origin |
|---|---|---|---|---|
| **Source Port** | STATIC−DEF | removed | STATIC−DEF | TS |
| **Destination Port** | STATIC−DEF | removed | STATIC−DEF | TS |
| **Length** | INFERRED | removed | INFERRED | IP payload length |
| **Checksum** | IRREGULAR | LSB | INFERRED | calc. |

**Table 6.9 Header classification for UDP.**

**Source Port:**
The Source Port is fixed in the SA and can be copied to the ROHC context, before the first packet is sent/received.

**Destination Port:**
The Destination Port is fixed in the SA and can be copied to the ROHC context, before the first packet is sent/received.

**Length:**
The length of the UDP header can be calculated like: IP header − IP header length. Therefore there is no need to send it on the wire and it is defined as INFERRED.

**Checksum:**
The checksum can be calculated by Diet−ESP and proved by comparing the LSB sent on the wire. The number of bytes sent on the wire can be 0, 1 and 2 stored in CHECKSUM_LSB. If 0 LSB is chosen, the checksum MUST be decompressed with the value 0. If the UDP implementation of the sender chose to disable the UDP checksum by setting the checksum to 0 Diet−ESP SHOULD be used with CHECKSUM_LSB = 0.

#### 6.3.3.3 UDP−Lite (profile 0x0007, 0x1007, 0x0008, 0x1008)

This section shows the compression of ESP payload for all ROHC profiles including an UDP−Lite header listed in Table 6.10.

| Field | Class | Compression Method | Diet−ESP ROHC class | Data origin |
|---|---|---|---|---|
| **Source Port** | STATIC−DEF | removed | STATIC−DEF | TS |
| **Destination Port** | STATIC−DEF | removed | STATIC−DEF | TS |
| **Checksum Coverage** | IRREGULAR | LSB | IRREGULAR | calc. |
| **Checksum** | IRREGULAR | LSB | INFERRED | calc. |

**Table 6.10 Header classification for UDP−Lite.**

**Source Port:**
The Source Port is fixed in the SA and can be copied to the ROHC context, before the first packet is sent/received.

**Destination Port:**
The Destination Port is fixed in the SA and can be copied to the ROHC context, before the first packet is sent/received.

**Checksum Coverage:**
The Checksum specifies the number of octets carried by the UDP–Lite checksum. It can have the same value as the UDP length (0 or UDP length) or any value between 8 and UDP length. This field is compressed with CHECKSUM_LSB of 0, 1 or 2 bytes. If 0 or 1 LSB is chosen, the field MUST be decompressed with the UDP length. If 2 LSB is chosen, the checksum has to carry this behavior.

**Checksum:**
The checksum can be calculated by Diet–ESP and proved by comparing the LSB sent on the wire. The number of bytes sent on the wire can be 0, 1 and 2 stored in CHECKSUM_LSB. If 0 LSB is chosen, the checksum MUST be decompressed with the value 0. If an UDP–lite implementation of the sender chose to disable the UDP checksum by setting the checksum to 0 Diet–ESP SHOULD be used with CHECKSUM_LSB = 0.

#### 6.3.3.4 TCP (profile 0x0006)

This section shows the compression of ESP payload for all ROHC profiles including a TCP header listed in Table 6.11. The ROHC context is partly filled while the Diet–ESP context exchange and some values can be removed. Since TCP is not stateless only fields with the compression methods 'removed' and 'LSB' are allowed to be compressed, the other fields have to be sent on the wire uncompressed.

| Field | Class | Compression Method | Diet–ESP ROHC class | Data origin |
|---|---|---|---|---|
| **Source Port** | STATIC–DEF | removed | STATIC–DEF | TS |
| **Destination Port** | STATIC–DEF | removed | STATIC–DEF | TS |
| **Sequence Number** | CHANGING | N/A | CHANGING | |
| **Acknowledgement Num** | INFERRED | N/A | INFERRED | |
| **Data Offset** | CHANGING | N/A | CHANGING | |
| **Reserved** | CHANGING | N/A | CHANGING | |
| **CWR flag** | CHANGING | N/A | CHANGING | |
| **ECE flag** | CHANGING | N/A | CHANGING | |
| **URG flag** | CHANGING | N/A | CHANGING | |
| **ACK flag** | CHANGING | N/A | CHANGING | |
| **PSH flag** | CHANGING | N/A | CHANGING | |
| **RST flag** | CHANGING | N/A | CHANGING | |
| **SYN flag** | CHANGING | N/A | CHANGING | |
| **FIN flag** | CHANGING | N/A | CHANGING | |
| **Window** | CHANGING | N/A | CHANGING | |
| **Checksum** | IRREGULAR | LSB | INFERRED | calc. |
| **Urgent Pointer** | CHANGING | N/A | CHANGING | |
| **Options** | CHANGING | N/A | CHANGING | |

**Table 6.11 Header classification for TCP.**

**Source Port:**
The Source Port is fixed in the SA and can be copied to the ROHC context, before the first packet is sent/received.

**Destination Port:**
The Destination Port is fixed in the SA and can be copied to the ROHC context, before the first packet is sent/received.

**Checksum:**
The checksum can be calculated by Diet−ESP and proved by comparing the LSB sent on the wire. The number of bytes sent on the wire can be 0, 1 and 2 stored in CHECKSUM_LSB. If 0 LSB is chosen, the checksum has to be decompressed with the value 0. If a TCP implementation of the sender chose to disable the TCP checksum by setting the checksum to 0 Diet−ESP should be used with CHECKSUM_LSB = 0.

## 6.4 Diet−ESP Extension: IV compression

The Initialization Vector (IV) is defined as an input to the encryption function AES. It is the same value to encrypt and decrypt, whereas it is usually sent on the wire. Standard IPsec/ESP does not define the IV, but it leaves its definition to the encryption function (see RFC3602 [20] for AES−CBC and RFC3686 [24] for AES−CTR). The IV has to be unique for every packet secured by one encryption key and unpredictable for an attacker.

As the IV is an implicit overhead for every packet (8 Byte in CTR, 16 Byte in CBC), this document introduces a Pseudo Random Function (PRF) to generate the IV explicit on both sides of the security communication. This enables the compression of the value sent on the wire, as it is predictable by the receiver but not for an attacker.

This section defines a Diet−ESP extension using a Pseudo Random Function for safely generating the same IV on both peers in order to remove it from the ESP payload.

### 6.4.1 Pseudo Random Function

PBKDF2 standardized in RFC2898 [33] defines a PRF, which can be used with any HMAC algorithm like CBC−HMAC, AES−XCBC−HMAC, SHA1, SHA−256 or SHA3. Figure 6.1 shows parameters and the output of PBKDF2.

```
+-----------------------------------------------------------------+
|PBKDF2 (P, S, c, dkLen)                                          |
|                                                                 |
|   Options:        PRF        underlying pseudorandom function (hLen |
|                              denotes the length in octets of the |
|                              pseudorandom function output)       |
|                                                                 |
|   Input:          P          password, an octet string           |
|                   S          salt, an octet string               |
|                   c          iteration count, a positive integer |
|                   dkLen      intended length in octets of the derived|
|                              key, a positive integer, at most    |
|                              (2^32 - 1) * hLen                   |
|                                                                 |
|   Output:         DK         derived key, a dkLen-octet string   |
+-----------------------------------------------------------------+
```

**Figure 6.1 PBKDF2 function parameters [33].**

For the explicit generation of the IV this extension considers the following parameters:

- **PRF:** The PRF is negotiated with the IV_PRFT value in the Diet−ESP context.
- **P:** The password is the encryption key.
- **S:** The SN of the packet.
- **C:** The number of iterations can be small, as the password is really strong (at least 128 Bit for AES). While it is not negotiated otherwise, the value 1 is used.
- **dkLen:** The output length is defined by the size of the implicit IV of the encryption algorithm (e.g. 8 Bytes for AES−CTR, 16 Bytes for AES−CBC).

### 6.4.2 Diet−ESP Context Extension

To enable the compression of the IV, the Diet−ESP context is extended with three values (see Table 6.12).

| Context Field Name | Overview |
|---|---|
| IV_COMPRESSION | Defines if IV compression is enabled. |
| IV_PRFT | Defines the hash function used as the algorithm for the Pseudo−Random−Function. |
| USE_IV_PBKDF2 | Defines the use of PBKDF2 as the algorithm for generating the Pseudo Random values. |

**Table 6.12 Diet−ESP context extension for compression of the IV.**

**IV_COMPRESSION:**
Specifies if the IV is sent on the wire or not. If it is set, the IV is removed completely. If it is unset the IV is sent like specified in the encryption algorithm. This value can only be used in combination with a pseudo random function, for example by specifying USE_IV_PBKDF2 in order to take advantage of PBKDF2:

**IV_PRFT:**
Defines the Pseudo Random Function Transform used for the Pseudo Random Function. The usable IDs are defined in [28], by default the algorithm of the authentication function is used. If NULL authentication is provided and the algorithm of IV_PRFT is not useable, the IV compression cannot be used.

**USE_IV_PBKDF2:**
Defines if the PBKDF2 is used as Pseudo Random function to generate the IV or not. The algorithm used to derive a Pseudo Random value is specified by IV_PRFT.

In regular ESP packets, the IV is included to the ICV generation, since it is placed directly after the ESP header. Therefore, if authentication is enabled, the IV must be included to the ICV. The IV has to be built and attached to the ESP payload before the ICV is going to be calculated. If IV_COMPRESSION is enabled, the IV is removed together with the ESP header, which is shown in Figure 6.2. In this case, the receiver must restore the IV before he calculates the ICV to ensure the integrity of the packet.

If further developments want to define another PRF function instead of PBKDF2, one should add a USE_IV_[NEW_PRF_FUNCTION] to the Diet−ESP context.

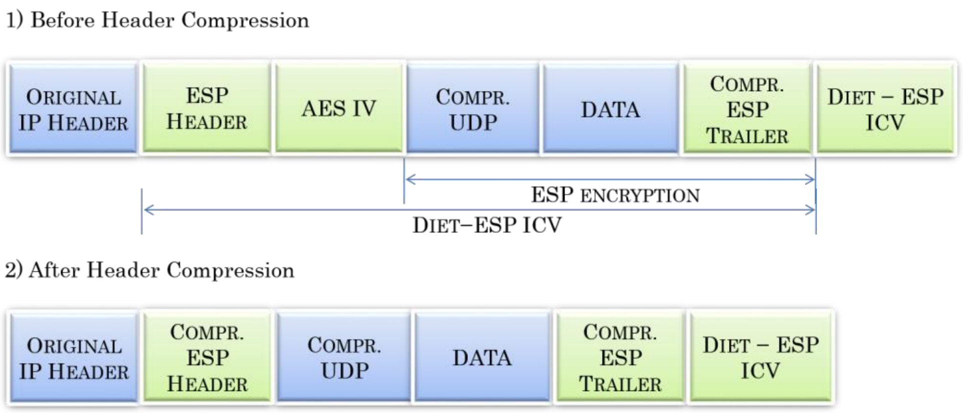


**Figure 6.2 IV before and after compressions.**

## 6.5 Diet–ESP Context Negotiation with IKEv2

Usually the IPsec databases SPD and SAD (see section 2.2.3) are set up by an IKE daemon. Since IKEv2 is the most proper protocol for this set up, this section describes the negotiation of the Diet–ESP context between the peers with IKEv2. The context packet format is shown in Figure 6.3. This 38 bits context has to be stored in exactly that order in the Security Association of both peers. One or more of these configurations are included in the IKE_AUTH messages inside a DIET_ESP_SUPPORT Notify Payload, as it is shown in Figure 6.4. The initiator proposes some specific contexts he wants to support. The responder selects a context by answering with another Notify Payload including the chosen context. It has to be one of the set, sent by the initiator. The acronym ANY says that the responder is able to support all Diet–ESP contexts the responder may chose. If the responder does not return a DIET_ESP_SUPPORT Notify Payload, the initiator knows that the responder does not support Diet–ESP. Subsequently, the initiator can chose either to decline the connection or to use the "Standard ESP compliant Diet–ESP Context" (see section 6.2.2) without any extensions. For inner header compression, the receiver has to decide if the negotiated traffic selector can be used to decompress the encrypted ESP payload. If it is not unique, the USE_SA bit needs to be set to 0. Unique means, that there is no range of IP addresses, ports or protocols inside the TS.

```
    0
    0          1          2          3          4          5          6          7
   +----------+----------+----------+----------+----------+----------+----------+----------+
 0|       ALIGN         |            SPI_SIZE            |            SN_SIZE             |
   +----------+----------+----------+----------+----------+----------+----------+----------+
 8|    NH    |   PAD    |      Diet-ESP_ICV_SIZE         |  USE_SA  | INNER_ROHC_PROFILE  |
   +----------+----------+----------+----------+----------+----------+----------+----------+
16|                                  INNER_ROHC_PROFILE                                   |
   +----------+----------+----------+----------+----------+----------+----------+----------+
24|                      INNER_ROHC_PROFILE                         |    CHECKSUM_LSB     |
   +----------+----------+----------+----------+----------+----------+----------+----------+
32| COMPR_IV |                  IV_PRFT                  |USE_PBKDF2|      RESERVED       |
   +----------+----------+----------+----------+----------+----------+----------+----------+
```

**Figure 6.3 Diet–ESP Context packet format for IKEv2.**

In the following, all contents of this context are described and assigned to their specific values.

ALIGN: (2 bits): specifies the alignment for padding as follows:

- 00: indicates an 8 bit alignment. There is no need for padding in that case.
- 01: indicates a 16 bit alignment.
- 10: indicates a 32 bit alignment.
- 11: indicates a 64 bit alignment.

SPI SIZE (3 bits): specifies the size of the SPI field length of the Diet−ESP header in byte. Values can be from 0 to 4. A zero value means the SPI does not appear in the Diet−ESP packet. The size depends on the use case, the connection should be used for.

- 000: indicates a 0 bit SPI. The SPI is removed from the packet.
- 001: indicates an 8 bit SPI in each Diet−ESP−packet.
- 010: indicates a 16 bit SPI in each Diet−ESP−packet.
- 011: indicates a 24 bit SPI in each Diet−ESP−packet.
- 100: indicates a 32 bit SPI in each Diet−ESP−packet. This configuration is according to the RFC 4303.
- 101: Unassigned.
- 110: Unassigned.
- 111: Unassigned.

SN SIZE (3 bits): specifies the size of the Sequence Number field within the Diet−ESP header in byte. Values can be from 0 to 4. A zero value means the SN does not appear in the Diet−ESP packet. The size depends on the use case, the connection should be used for.

- 000: indicates a 0 bit SN. The SN is removed from the packet and anti−replay is disabled on the receiver.
- 001: indicates an 8 bit SN in each Diet−ESP−packet.
- 010: indicates a 16 bit SN in each Diet−ESP−packet.
- 011: indicates a 24 bit SN in each Diet−ESP−packet.
- 100: indicates a 32 bit SN in each Diet−ESP−packet. This configuration is according to the RFC 4303.
- 101: Unassigned.
- 110: Unassigned.
- 111: Unassigned.

NH (1 bit): specifies if the Next Header field appears in the Diet−ESP trailer. NH unset to 0 indicates the Next Header field is present and NH set to 1 indicates the Next Header is omitted.

PAD (1 bit): specifies if the Pad Length field appears in the Diet−ESP trailer. P unset to 0 indicates the Pad Length field is present and P set to 1 indicates the Pad Length is omitted.

Diet−ESP_ICV_SIZE (3 bits): specifies the transmitted number of bytes to authenticate the Diet−ESP packet. If one chooses not to perform authentication, one has to negotiate the authentication algorithm NULL as defined in RFC4835. The minimum length greater than 0 for ICV is 96 bits and can be generated with the following hash functions: HMAC−MD5−96 [44], HMAC−SHA1−96 [45], AES−CMAC−96 [74], AES−XCBC−MAC−96 [21]. As a result Diet−ESP only specifies sizes smaller than 96 bits.

- 000: ICV is left untouched as it is specified by the authentication algorithm.
- 001: Diet−ESP ICV consists of the 8 most significant bits of ESP ICV.
- 010: Diet−ESP ICV consists of the 16 most significant bits of ESP ICV.
- 011: Diet−ESP ICV consists of the 32 most significant bits of ESP ICV.
- 100: Diet−ESP ICV consists of the 64 most significant bits of ESP ICV.
- 101: Unassigned.
- 110: Unassigned.
- 111: Unassigned.

USE_SA (1 bit): specifies if the Traffic Selector of the SA should be used to compress the ESP payload like described in the Diet−ESP extension for Upper Layer Compressions (see section 6.3). The value 0 says, that no upper layer compression is used. If set to 0 the following 18 bits for INNER_ROHC_PROFILE and CHECKSUM_LSB are not used but they have to be negotiated. The value 1 means, that the upper layer headers inside ESP payload are compressed, according to the profile specified in INNER_ROHC_PROFILE.

INNER_ROHC_PROFILE (16 bits): specifies the ROHC profile used for compression and decompression the upper layer headers inside the ESP payload. The values are specified by IANA [18].

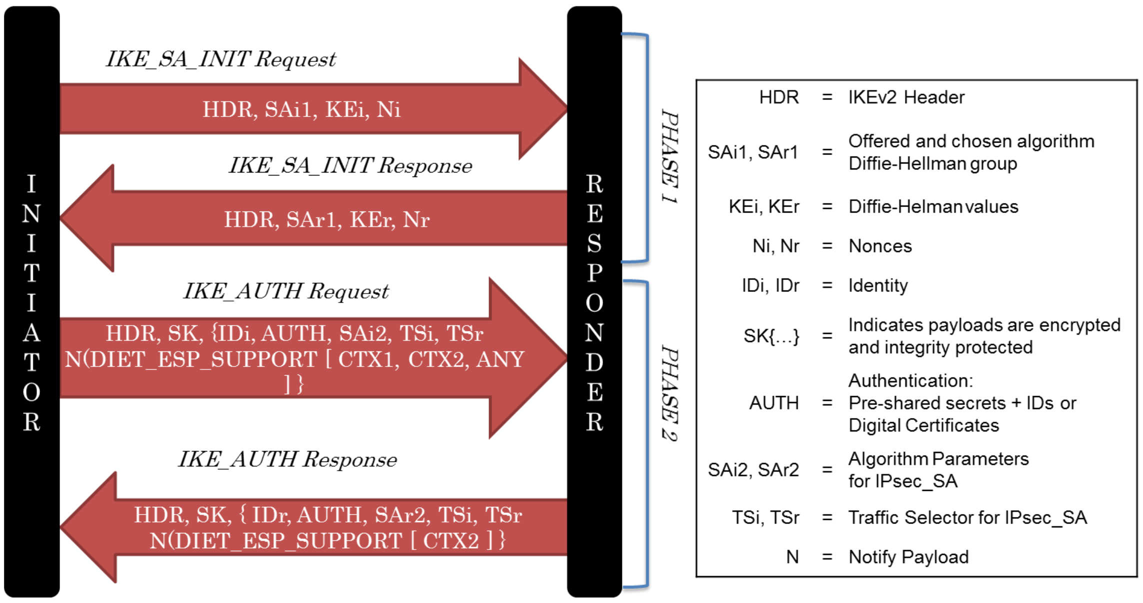
CHECKSUM_LSB (2 bits): specifies the number of bytes sent on the wire for the checksum of upper layer protocols.

- 00: the checksum of the upper layer protocols are not send on the wire and have to be regenerated by the receiver.
- 01: the checksum of upper layer protocols consists of the 1 last significant byte of the original value.
- 10: the checksum of upper layer protocols consists of the 2 last significant byte of the original value. That says the checksum is not compressed.
- 11: Unassigned.

COMPR_IV (1 bit): specifies if the AES−IV is compressed. The value 0 defines that the IV is not compressed and send on the wire like described in the cipher specific RFC. The value 1 says that the IV is removed completely.

IV_PRFT (4bit): specifies the algorithm, used as an input of the pseudo random function of, for example for PBKDF2. Currently assigned numbers are 1−8 [28]. The value 0 is reserved.

USE_PBKDF2 (1bit): specifies that the PKDF2 function is used to generate the AES−IV. The value 1 defines the generation of the IV with PKDF2, whereas 0 says that it is generated otherwise.


IKE_SA_INIT Request
HDR, SAi1, KEi, Ni
IKE_SA_INIT Response
HDR, SAr1, KEr, Nr
IKE_AUTH Request
HDR, SK, {IDi, AUTH, SAi2, TSi, TSr N(DIET_ESP_SUPPORT [ CTX1, CTX2, ANY ] }
IKE_AUTH Response
HDR, SK, { IDr, AUTH, SAr2, TSi, TSr N(DIET_ESP_SUPPORT [ CTX2 ] }
INITIATOR
RESPONDER
PHASE 1
PHASE 2

| | | |
|---|---|---|
| HDR | = | IKEv2 Header |
| SAi1, SAr1 | = | Offered and chosen algorithm Diffie-Hellman group |
| KEi, KEr | = | Diffie-Helman values |
| Ni, Nr | = | Nonces |
| IDi, IDr | = | Identity |
| SK{...} | = | Indicates payloads are encrypted and integrity protected |
| AUTH | = | Authentication: Pre-shared secrets + IDs or Digital Certificates |
| SAi2, SAr2 | = | Algorithm Parameters for IPsec_SA |
| TSi, TSr | = | Traffic Selector for IPsec_SA |
| N | = | Notify Payload |

**Figure 6.4 DIET_ESP_SUPPORT Notfiy Payload within the Key Exchange with IKEv2.**

## 6.6 Minimal–ESP implementation

This section defines a minimal implementation guideline of the ESP protocol. Therefore the packet on the wire has to be conform to RFC4303 [36]. RFC4303 defines ESP for high interoperability, mainly used by huge IPsec infrastructures for secured Virtual Private Networks. A lot of these features are not necessary for a single point–to–point connections, usually used in the Internet of Things. Since sensors are usually constrained in memory, processor and power consumption, Minimal–ESP focuses on these three aspects by reducing computation and memory usage. Minimal–ESP requires a RFC4303 conform ESP packet, whereas packet compressions are not part of it. Anyway, it provides guidance how the packet size can be minimized without the use of compressions.

First of all, the sensor should only implement the parts of IPsec respectively ESP it is going to use. If the sensor will only send data, there is no need to implement the incoming IPsec stack, and vice versa.

Additionally, most of the ESP fields are designed for high interoperability and huge infrastructures. In the following, all fields are discussed for potential simplifications. The key idea is to fix the values if possible, like described in section 4.3.1 – 4.3.6. All fields not mentioned in the following are handled like described in RFC4301 or RFC4303.

### 6.6.1 Security Parameter Index (SPI)

Since the SPI is chosen by the receiver of the packet, a Minimal–ESP implementation can only chose this value if it is the receiving part of the connection. Anyway, to prevent storage on the sensor, the receiver should chose the least significant 32 bits of the IP address if it is unique. If that is ensured the Security Association can be found by using the IP address of the IP packet holding the ESP packet.

Generally, this can be used by receiving implementations only. But if a sending implementation recognizes a known value for the SPI, it can free the memory for the storage of the SPI.

### 6.6.2 Sequence Number (SN)

The Sequence Number cannot be fixed, since it must increase in every packet. To prevent the 32 bit of storage used by the SN one may decide to use the time, as an always increasing value.

If not more than one packet every second is going to be sent, the current time in seconds may be a good choice. Using the seconds will provide an increasing value for roughly 136 years. If minutes can be used the value is much higher with around 8,165 years.

This is only for sending implementations.

### 6.6.3 Padding

The padding can be reduced by using ciphers in Counter Mode. Therefore it is recommended to use ciphers in Counter Mode for Minimal−ESP implementations.

Additionally the padding can be reduced by clever alignment of the application size to the cipher block size or IP alignment.

This is only for sending implementations.

### 6.6.4 Encryption

AES−CTR is defined as “SHOULD” in RFC4835 [46], why it is safe to assume that most IPsec implementations support AES−CTR. Additionally AES−CCM* is the cipher which has to be implemented by any IEEE 802.15.4 [3] device.

A Minimal−ESP implementation should prefer ciphers supported by hardware accelerations. In addition, ciphers in Counter Mode should be used to reduce the padding.

This is for sending and receiving implementations, but since no cipher in Counter Mode is explicit required for ESP, there may be a fallback to AES−CBC if the receiver does not support a cipher in Counter Mode. Note that it is required to support AES−CBC but not to propose it as the used cipher to the receiver, e.g. by IKEv2. Therefore a sender may only propose AES−CTR, but if the receiver does not support it, the connections will not be established.

Some use cases may not require confidentiality. In that case ESP−NULL cipher can be used as well, but in that case authentication is required.

### 6.6.5 Authentication

As long as ESP−NULL encryption is not going to be used, the ICV is an optional value with variable length. Although optional, it is strongly recommended to use the ICV. IoT devices may allow weak security by removing the ICV, and gateways wanting to connect to IoT devices should be able to deal with NULL authentication.

Like for encryption it is recommended to use ciphers supporting hardware acceleration, e.g. AES−XCBC [21] or AES CBC−MAC [25].

This is for sending and receiving implementations.

### 6.6.6 IPsec Databases

For a minimal implementation of the ESP protocol, the databases should be small in order to reduce the storage. For a RFC4301 compliant IPsec implementation, the following three policy rules should be implemented:

- The traffic IPsec should secure, for example UDP or TCP to and from a specific port.
- The last rule of the IPsec policies should discard all traffic not matched by previous rules [37: p. 60].
- If IKEv2 is used, the IKEv2 traffic should be excluded by the policies. This can be done by bypassing traffic to and from UPD port 500.

### 6.6.7 Packet Procession

A Minimal–ESP implementation does not handle all the extensions IPsec provides. It is recommended to deal only with the minimal set of extension, in most use cases none of them. One example of unnecessary extension is NAT traversal. ESP can work behind NAT gateways by sending the ESP packet inside a UDP packet with the ports 500 or 4500. In IoT scenarios this is a not essential functionality, as IoT devices usually work inside an IPv6 environment in which NAT is not common.

Figure 6.5 and Figure 6.6 show the outgoing and incoming packet procession for ESP without any extensions. Since ESP is unidirectional, there is no direct exchange between the peers after the keys are exchanged and the SA is not expired. The incoming and outgoing procession are quite similar, but in reversed order.

First, there is the SAD lookup in order to get the correct keys and values from the Security Association in both cases. For outgoing packets, the properties of the IP packet are looked up for a Security Policy matching the IP packet. The found outgoing SPD entry contains a link to the associated SAD entry. If there is no entry in the SPD, the packet is either forwarded without securing or dropped. The SPD will hold an entry which is going to be done for not found packets. For incoming packets, the ESP header contains the SPI, which is unique in the incoming SAD. Once an entry is found the IP packet is rechecked for an entry in the SPD in order to prevent packet procession for incorrect packets.

After succeeded lookup in the databases, the literal packet procession starts.

During outgoing procession the IP packet is encapsulated. The procession differs for connections in Tunnel and Transport mode. In Tunnel mode a new IP header is built, according to the information in the Security Association. The original IP header is left like it was, the padding is calculated before the ESP trailer is added and encrypted together with the original IP packet. In Transport mode, the Payload Length and Next Header fields of the original IP header are modified and the ESP header is placed between IP and transport layer header, before the creation of the ESP trailer. The encryption parameters (encryption algorithm and key) are read from the SA. In most implementations, the encapsulation is embedded in an encryption function, since the padding generation and IV attachment depends on the used algorithms. The last step is the generation of the ICV with the authentication parameters (algorithm and key) with the result of an IP packet including an ESP secured packet.

The procession of an ESP packet inside an incoming IP packet is quite similar to the encapsulation, but of course in the reverse way. After the SAD–lookup the expected ICV of the ESP packet is generated and compared with the one attached in the packet. If they do not match there may be an error or malicious modification in the payload and the packet is dropped. Now the packet is checked for a replay attack by comparing the last sequence number stored in the SA with the one transmitted in the packet. In order to prevent dropped packets due to disordered packet arrival, there is an anti–replay–window for lesser sequence numbers defined in the SA. Packets with a sequence number lower than the stored one minus the anti–replay–window are dropped. After these security checks, the ESP payload can be decrypted and de–capsulated with the same differences for Tunnel and Transport mode already shown during the encapsulation. In Transport mode, the Next Header field of the IP packet is filled with the information in the ESP trailer. The payload length is calculated by subtracting the length of ESP header, IV, ESP trailer and ICV from the one stored in the IP packet. In Tunnel mode rebuilding is a bit simpler, since the inner IP packet can be moved to the position of the outer IP header. The inner IP header holds all necessary information for a correct procession of the IP packet.

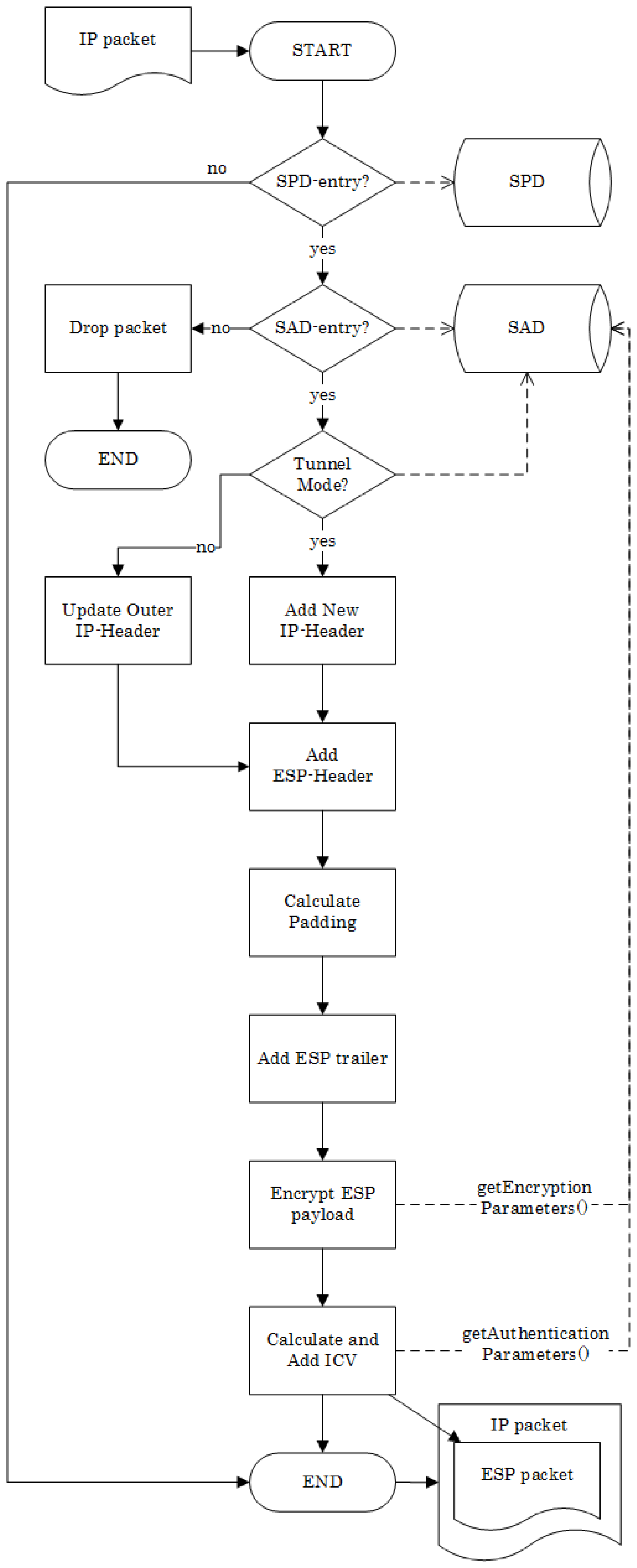


**Figure 6.5 Minimal–ESP outgoing packet procession.**

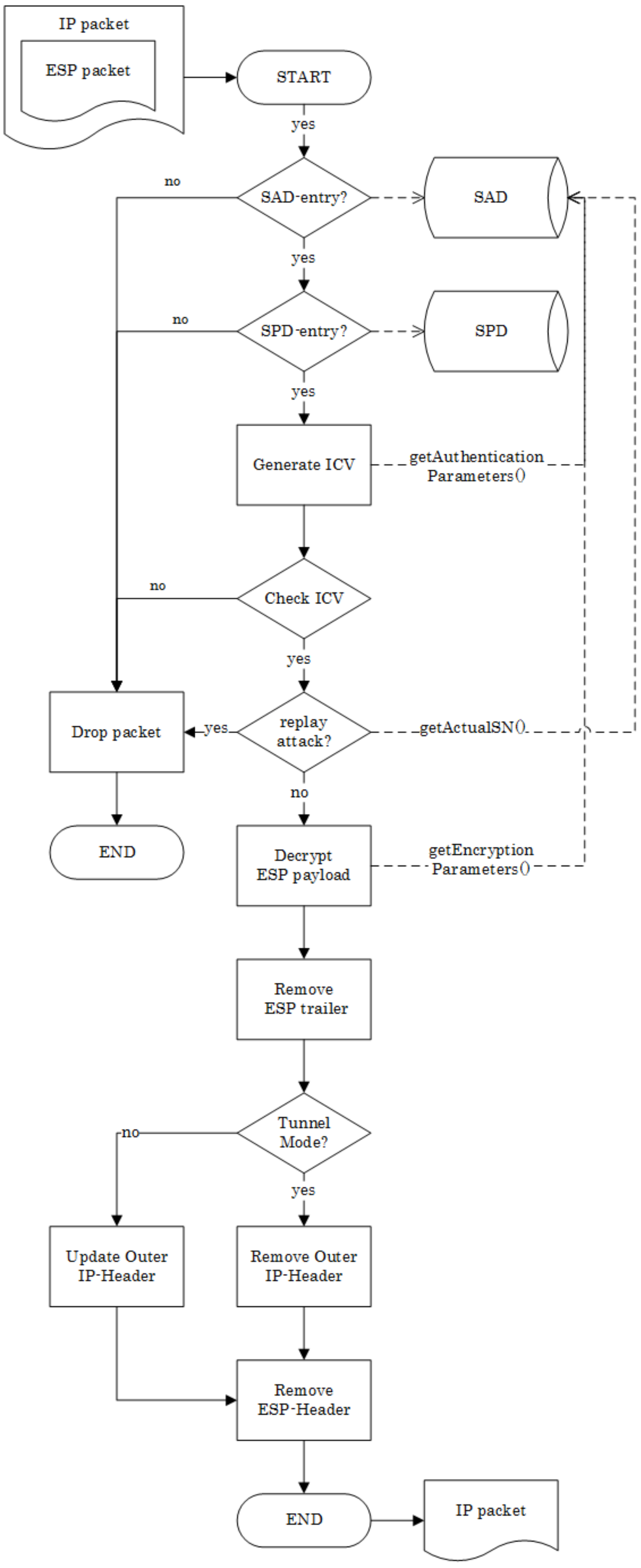


Figure 6.6 Minimal−ESP incoming packet procession.

## 6.7 Diet−ESP Protocol Description

This section defines Diet−ESP on the top of the ROHC and ROHCoverIPsec framework.

### 6.7.1 Diet−ESP ROHC framework

This section defines how the compression of all ESP fields is performed within the ROHC and ROHCoverIPsec frameworks. Fields that are in the ESP Header (i.e. the SPI and the SN) and the ICV are compressed by the ROHC framework. The other fields, that is to say those of the ESP Trailer, are compressed by the ROHCoverIPsec framework. The specific Diet−ESP ICV field is detailed in Section 6.7.2.

Diet−ESP fits in the ROHC and the ROHCoverIPsec in a very specific way.

1. Diet−ESP does not need any ROHC signaling between the peers. More specifically, ROHC Initialization and Refresh (IR), ROHC IR−DYN or ROHC Feedback packet are not considered with Diet−ESP. The first reason is, that fields are either STATIC or PATTERN and their value or profile is defined through the Diet−ESP Context agreed out−of band by the peers. Subsequently, the profiles are applied for each Security Association that is unidirectional. The SA negotiation results in two unidirectional SA's. As a result, each SA is used for one direction only, which corresponds to the unidirectional mode (*U−mode*).
2. Diet−ESP only exchange compressed data. How the compression / decompression occurs is defined by the Diet−ESP Context. Once the Diet−ESP Context has been agreed, both peers are in a *Second Order (SO) State* and exchange only compressed data.
3. Diet−ESP itself only compresses ESP packets. It may include inner packet compression as an extension, but it does not make any assumption on the IP compression for the outer IP header used for sending the data. This is made in order to make Diet−ESP interoperable with multiple IP compression protocols.
4. Diet−ESP compresses partially STATIC fields as they are used as indexes by the receiver and may not be removed completely.

Similarly like in ROHC, the fields of the ESP−header are classified, what defines the method for compression and decompression. This ensures that sender and receiver use the same method, ensuring the correct decompression at the receiver. Table 6.13 defines the classification together with the used Encoding Method and the Diet−ESP Context Parameters used for compression and decompression.

| **Field** | **ROHC class** | **Framework** | **Encoding Method** | **Diet-ESP Context Parameters** |
|---|---|---|---|---|
| SPI | STATIC-DEF | ROHC | LSB | SPI_SIZE |
| SN | PATTERN | ROHC | LSB | SN_SIZE |
| Padding | PATTERN | ROHCoverIPsec | Removed | PAD, ALIGN |
| Pad Length | PATTERN | ROHCoverIPsec | Removed | PAD, ALIGN |
| Next Header | STATIC-DEF | ROHCoverIPsec | Removed | NH |

**Table 6.13 Classification of the ESP fields.**

### 6.7.2 Diet−ESP ICV

With standard ESP the ICV is computed over the whole ESP Packet. If Diet−ESP were using the Standard ESP ICV value, then Diet−ESP would have to decompress Diet−ESP to Standard ESP packet to check the Standard ESP ICV value. First, this is not in the scope of Diet−ESP that is looking for light implementation of ESP. Then, as there is no standard way to generate the padding, this may be impossible in most cases.

The Diet−ESP ICV is defined to enable an integrity check without decompressing the Diet−ESP packet to Standard ESP before the integrity check is performed. Therefore the Diet−ESP ICV is computed over the Diet−ESP packet before the ESP Header is compressed. In other words, the Diet−ESP ICV is computed over the compressed ESP payload but before header (SPI, SN and IV) compression.

The reason of building the Diet−ESP ICV over the uncompressed header is the anti−replay protection of IPsec. If anti−replay is enabled at the receiver of the packet, the ICV ensures the integrity of the SN sent on the wire. Suppose the SN is compressed to 0 byte and the Diet−ESP ICV is built over the compressed ESP Header. In that case, the Diet−ESP ICV would not consider the SN value and thus removes the ESP anti−replay mechanism, as the SN cannot be compared with the one chosen by the sender. The problem remains if the SN is compressed to less than 4 bytes and the sequence number chosen by the sender increases out of the range of the SN_SIZE sent on the wire. For example, if the SN_SIZE is 1 byte the maximum increasing can be 255. If the received SN is increased by 300, the receiver will recognize an increase of only 45, whereby the mechanism would be corrupted.

Due to these issues the SN has to be decompressed to 32 bit SN before the Diet−ESP ICV generation takes place (see Figure 6.8). More specifically, the regular ESP header is used for the Diet−ESP ICV generation on sender and receiver. This mechanism ensures the correctness of the anti−replay mechanism and the possibility of sending Standard ESP conform packets remains. As the receiver includes the ESP header to the Diet−ESP ICV generation he always checks the whole 32 Bit SN.

The algorithm used to generate the Diet−ESP ICV is the same as the one negotiated for ESP. As this field is computed for every packet, it is classified as PATTERN. The compressed Diet−ESP ICV consists in the Diet−ESP_ICV_SIZE LSB of the Diet−ESP ICV. Diet−ESP_ICV_SIZE is provided by the Diet−ESP Context. Profile parameters are summed up in Table 6.14

| **Field** | **ROHC class** | **Framework** | **Encoding Method** | **Diet−ESP Context Parameters** |
|---|---|---|---|---|
| **Diet−ESP ICV** | PATTERN | ROHCoverIPsec/ ROHC | LSB | Diet−ESP_ICV_SIZE |

**Table 6.14 Diet−ESP ICV profile.**

The Diet−ESP ICV differs from the ROHC ICV described in section 4.2 of RFC5858 [15]. The ROHC ICV is computed over the Clear Text Data and encapsulated in the ESP payload. The goal of the ROHC ICV is to check the integrity of Clear Text Data output. It ensures that the compressed payload is not corrupted by an attacker. As a result, the ROHC ICV authenticates the Clear Text Data over the whole chain of compressor/network/decompressor. In contrast, the Diet−ESP ICV authenticates the Diet−ESP packet over the network transmission. Note that the ROHC−ICV can be disabled by negotiating the algorithm NULL in the ROHC_INTEG notify payload RFC5857 [16].

Figure 6.7 illustrates the Standard ESP ICV, ROHC ICV and Diet–ESP ICV. The ROHC ICV can be used with Diet–ESP ICV or Standard ESP ICV. Diet–ESP considers ROHC–ICV as disabled, that is to say that ROHC_INTEG algorithm is set to NULL.

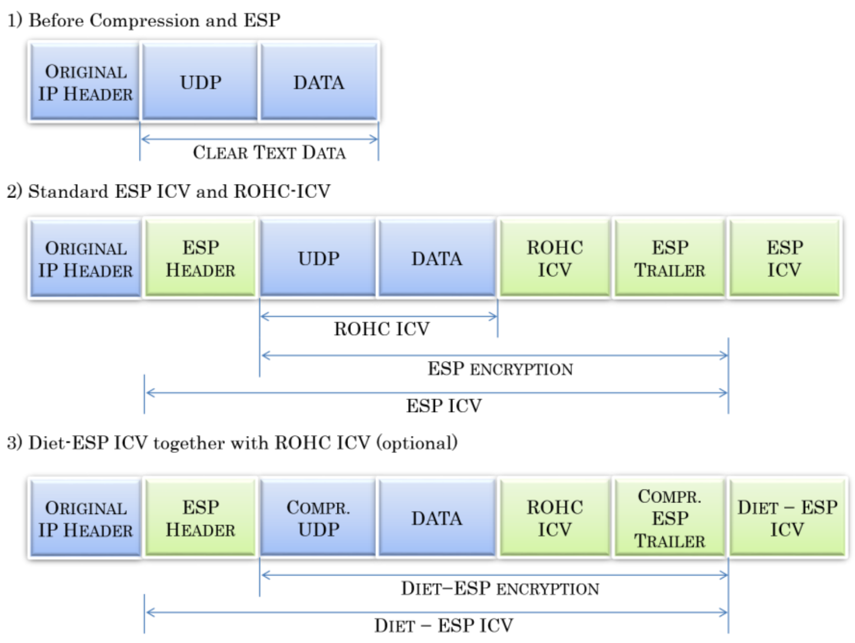


**Figure 6.7 Diet–ESP–ICV in IPsec Transport Mode**

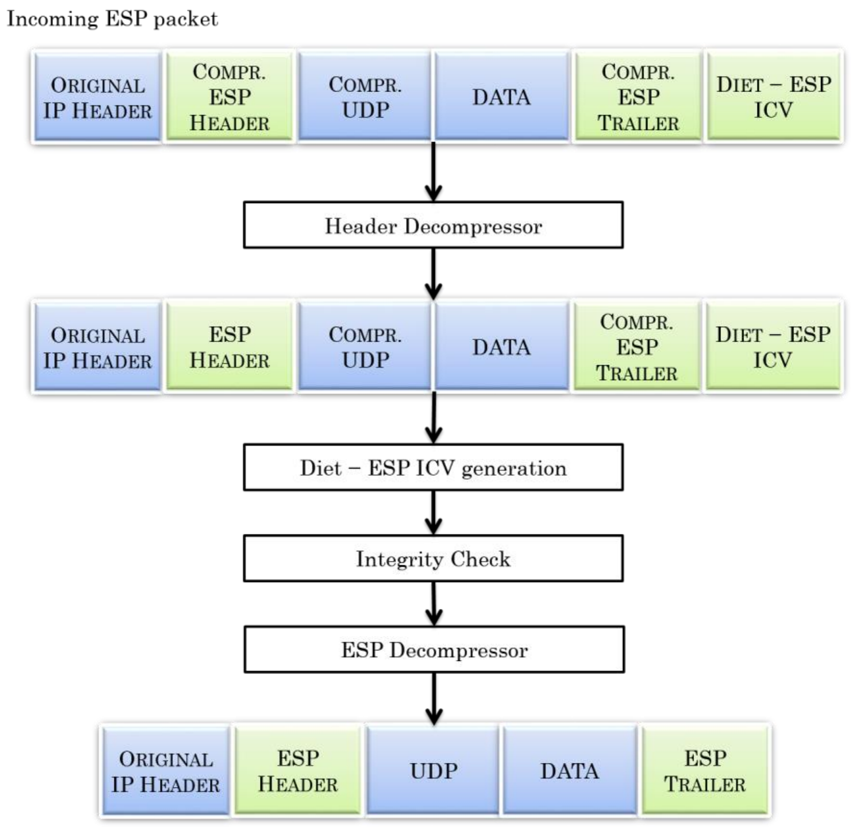


**Figure 6.8 Example of decompression of the header, before ICV generation and checking.**

### 6.7.3 Integration to Regular ESP

This section details how to use Diet–ESP for sending and receiving messages. The use of Diet–ESP is based on the IPsec architecture [37] and the ESP protocol [36], therefore only the adaptation that may be involved by Diet–ESP are listed below.

The Diet–ESP context is stored in the Security Association Database (SAD) where it is accessible for the ESP implementation as shown in Figure 6.9.

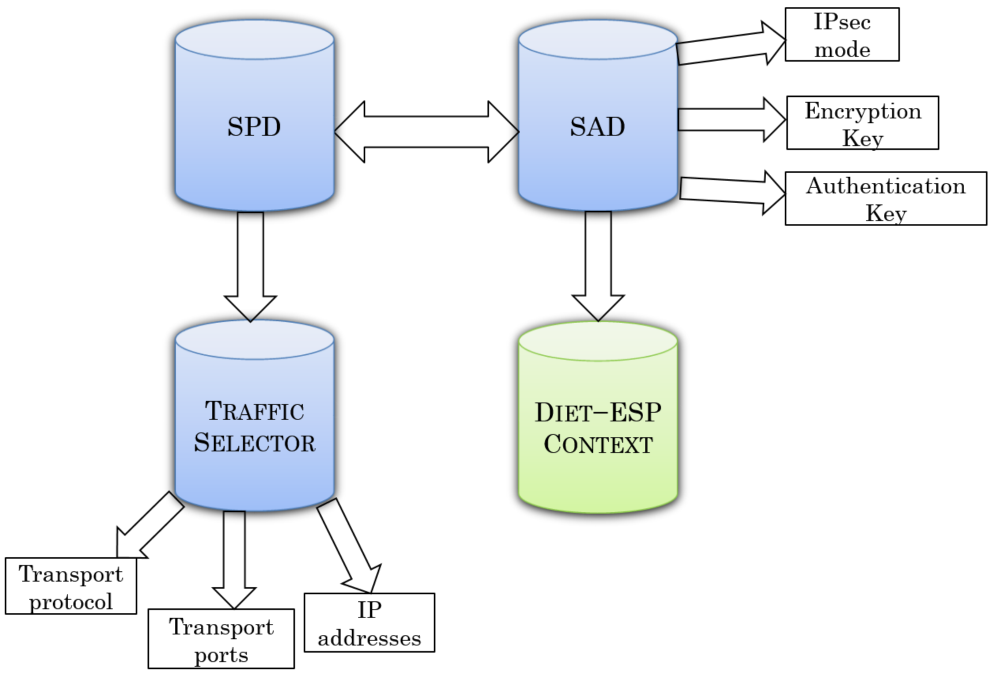


**Figure 6.9 Integration of the Diet–ESP Context to the SAD.**

#### 6.7.3.1 Packet Procession

The procession of Diet–ESP packets can be easily embedded to every already existing ESP implementation. Diet–ESP defines two different compression layers, which has to be implemented and called at the correct location inside the ESP procession. The separation of these two compression layers are necessary, since the encrypted payload should be compressed as well as the ESP header. In this context, the two layers are called “Compress ESP payload” and “Compress ESP header + ICV” for outgoing packets (see Figure 6.10) and “Decompress ESP header” and “Decompress ESP payload” for incoming packets (see Figure 6.11). An explicit decompression of the ICV is useless, since it is impossible to ensure that the ICV bits which are not sent on the wire are the same as the one of the calculation. Therefore, only the compressed Diet–ESP ICV is compared with the negotiated part of the generated Diet–ESP ICV. If the cipher algorithm using an IV, it is compressed resp. decompressed in the header compression, which is contradicted to the common wording, in which the IV is part of the ESP payload. This is necessary, as the IV is part of the authenticated data, why it has to be compressed after the Diet–ESP ICV generation.

For simplification reasons, the figures show the integration of the compression layers inside the Minimal–ESP packet procession. But there is no expected difference if a more complex ESP implementation is used, as the integration is at the end (resp. the beginning for incoming packets) of the procession. This ensures, that a full compatible ESP packet is provided to the further ESP implementation. The full expanded flowchart diagrams for in– and outgoing procession and the flowchart diagrams for the different compression layers are provided in Appendix A.

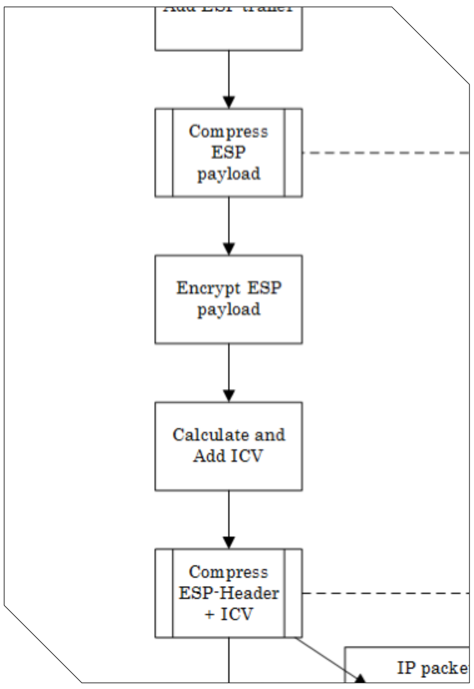


**Figure 6.10 The Diet–ESP compression layers inside the outgoing packet procession.**

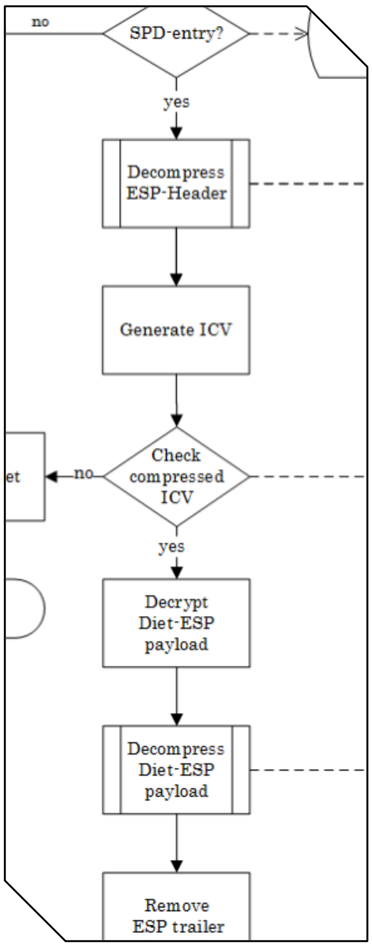


**Figure 6.11 The Diet–ESP compression layers inside the incoming packet procession.**

### 6.7.3.2 Inbound Security Association Lookup

Identifying the SA for incoming packets is one of the main reasons the SPI is sent in each packet on the wire. For regular ESP (and AH) packets, the Security Association is detected as follows:

1. Search the SAD for a match on {SPI, destination IP address, source IP address}. If an SAD entry matches, then process the inbound ESP packet with that matching SAD entry. Otherwise, proceed to step 2.
2. Search the SAD for a match on {SPI, destination IP address}. If the SAD entry matches, then process the inbound ESP packet with that matching SAD entry. Otherwise, proceed to step 3.
3. Search the SAD for a match on only {SPI} if the receiver has chosen to maintain a single SPI space for AH and ESP, or on {SPI, protocol} otherwise. If an SAD entry matches, then process the inbound ESP packet with that matching SAD entry. Otherwise, discard the packet and log an audible event.

Devices configured to work with a single SPI_SIZE value can process inbound packets as defined in RFC4301. As such, no modifications is required by Diet–ESP. Devices supporting different SPI_SIZE values, the way inbound packets are handled differs from the one described above. In fact, when an inbound packet is received, the peer does not know the SPI_SIZE used to send the packet. As a result, it does not know the SPI that applies to the incoming packet. The different values could be the 0 (resp. 1, 2, 3 and 4) first bytes of the IP payload.

Since the size of the SPI is not known for incoming packets, the detection of inbound SAs has to be redefined in a Diet–ESP environment. In order to ensure a detection of an SA the above described regular detection has to be done for each supported SPI_SIZE (in most cases 5 times), which most generally will return a unique Security Association.

If there is more than one SA matching the lookup, the authentication has to be performed for all found SAs to detect the SA with the correct key. In case there is no match, the packet has to be dropped. Of course this can lead into DoS vulnerability as an attacker recognizes an overlap of one or more IP–SPI combinations. Therefore it is highly recommended to avoid different values of the SPI_SIZE for one tuple of Source and Destination IP address. Furthermore, this recommendation becomes mandatory if NULL authentication is supported. This is easy to implement as long as the sensors are not mobile and do not change their IP address what should not be the case in IPv6 environments.

The following optimizations may be considered for sensors that are not likely to perform mobility or multi–homing features provided by MOBIKE (RFC4555 [14]) or any change of IP address during the lifetime of the SA.

#### 6.7.3.2.1 Optimization 1 – SPI_SIZE is mentioned inside the SPI

The SPI_SIZE is defined as part of the SPI sent in each packet. Therefore the receiver has to choose the most significant 2 bits of the SPI in the following way in order to recognize the right size for incoming Diet–ESP packets:

| | |
|---|---|
| 00: | SPI_SIZE of 1 byte is used. |
| 01: | SPI_SIZE of 2 byte is used. |
| 10: | SPI_SIZE of 3 byte is used. |
| 11: | SPI_SIZE of 4 byte is used. |

If the value 0 is chosen for the SPI_SIZE this option is not feasible.

#### 6.7.3.2.2 Optimization 2 – IP address based lookup

IP addressed based search is one optimization one may choose to avoid several SAD lookups. It is based on the IP address and the stored SPI_SIZE, which has to be the same value for each SA of one IP address tuple. Otherwise it can neither be ensured that one SA is found nor that the correct one is found. Note that in case of mobile IP the SPI_SIZE has to be updated for all SAs related to the new IP address which may cause in renegotiation. Figure 6.12 shows this lookup described below.

1. Search most significant SA as follows:
    1.1. Search the first SA for a match on {destination address, source address}. If an SA entry matches, then process to step 2. Otherwise, proceed to step 1.2.
    1.2. Search the first SA for a match on {source address}. If an SA entry matches, then process to step 2. Otherwise, drop the packet.

2. Identify the size of the compressed SPI for the found SA, stored in the Diet−ESP context. Note that all SAs to one IP address must have the same value for the SPI_SIZE. Then go to step 3.
3. If the SPI_SIZE is not zero, read the SPI_SIZE long SPI from the packet and perform a regular SAD lookup as described in RFC4301. If the SPI_SIZE is zero, the SA from step 1 is unique and can be used.

Note that an implementation can collect all SPI's matching the IP addresses in step 2 to avoid an additional lookup over the whole SAD. This is implementation dependent.

If the sensor is likely to change its IP address, the outcome could be a given IP address associated with different SPI_SIZE values. This case can occur if one IP address has been used by a device not anymore online, but the SA has not been removed. The IP has then been provided to another device. In this case the Diet−ESP Context should not be accepted by the Security Gateway when the new Diet−ESP Context is provided to the Security Gateway. At least the Security Gateway can check if the previous peer is reachable and then delete the SA before accepting the new SA.

Another case is a sensor with two interfaces with different IP addresses, negotiating different SPI_SIZE on each interface and then use MOBIKE to move the IPsec channels from one interface to the other. In this case, the Security endpoint should not accept the update, or force a renegotiation of the SPI_SIZE for all SAs, basically by re−keying the SAs.

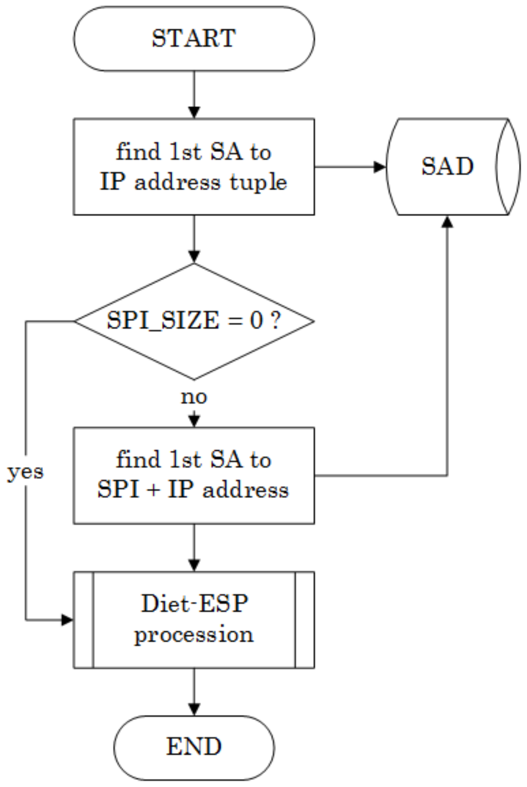


**Figure 6.12 SAD lookup for incoming packets.**

## 6.8 Interaction with other Compression Protocols

Diet−ESP exclusively defines compression for the ESP protocol as well as the ESP payload. It does not consider compression of the IP protocol. ROHC or 6LoWPAN are proper standardized protocols which may be used by a sensor to compress the IP (resp. IPv6) header. How Diet−ESP interacts with these two protocols, is specified below. Since compression usually occurs between the MAC and IP layers, there are no expected complications with this family of compression protocols.

If a compression protocol specifies the compression of the encrypted ESP payload it can only be compatible to Diet−ESP if Diet−ESP produces RFC4303 conform output (see section 6.2.2). Such a compression can only take place with a second ESP stack. In this case, the developer has to make sure that the ESP stacks are proper differed from each other.

### 6.8.1 ROHC

ROHC and ROHCoverIPsec have been used to describe Diet−ESP, although the interactions with these two protocols are described in the following:

Diet−ESP smoothly interacts with regular ROHC implementation appearing between MAC and IP layer. If the used ROHC profile enables the ESP compression (e.g. profile 0x0003 and 0x1003), the Diet−ESP context has to be negotiated with SPI_SIZE = 4 and SN_SIZE = 4. The ROHC implementation takes care of the IP and ESP header compression, even though the encrypted payload is compressed with Diet−ESP.

If ROHCoverIPsec is implemented in one of the peers, the developer has to ensure the proper separation of ROHCoverIPsec and Diet−ESP traffic to two ESP stacks.

### 6.8.2 6LoWPAN

Diet−ESP smoothly interacts with 6LoWPAN. Every 6LoWPAN compression header (NHC_EH) has an NH bit. This one is set to 1 if the following header is compressed with 6LoWPAN. Similarly, the NH bit is set to 0 if the following header is not compressed with 6LowPAN. This section details two cases. First of all, how Diet−ESP is compatible with ESP not compressed by 6LowPAN and then how Diet−ESP is compatible with ESP compressed with 6LowPAN.

Suppose 6LowPAN indicates the Next Header ESP is not compressed by 6LowPAN. If the peers have agreed to use Diet−ESP, the ESP layer on each peers receive the expected Diet−ESP packet. Diet−ESP is fully compatible with 6LowPAN ESP compression disabled.

Suppose 6LowPAN indicates the Next Header ESP is compressed by 6LowPAN. ESP compression with 6LowPAN considers the compression of the ESP Header, that is to say the compression of the SPI and SN fields. As a result 6LowPAN compression expects a 4 byte SPI and a 4 byte SN from the ESP layer. Similarly 6LowPAN decompression provides a 4 byte SPI and a 4 byte SN to the ESP layer. If the peers have agreed to use Diet−ESP and one of them uses 6LowPAN ESP compression, then the Diet−ESP must use SPI_SIZE and SN_SIZE set to 4 bytes.

If 6LoWPAN should support ESP payload compression in one of the peers, the developer has to ensure the proper separation of 6LoWPAN and Diet−ESP traffic to two separated ESP stacks.

## 6.9 Evaluation of Diet−ESP

In order to evaluate Diet−ESP, it is compared with the other mentioned compression methods ROHC and 6LoWPAN. Assuming an ESP packet in Tunnel mode and AES−CTR encryption, the maximum saving of Diet−ESP is 67 bytes in comparison with a regular ESP packet (see Table 6.15). In comparison with ROHC and 6LoWPAN (both considering the second ESP stack for ROHCoverIPsec and 6LoWPANoverIPsec) Diet−ESP still saves 11 resp. 15 bytes. For the calculation of the ESP overhead, the ESP header (8 bytes) in addition to the ESP trailer (1 byte padding for AES−CTR, Pad Length and Next Header field) is added. ROHC and 6LoWPAN can compress the header, but not the trailer. Additionally, 6LoWPAN requires the 1 byte extension header to be sent in every packet. All protocols can compress the inner IPv6 header and the UDP header. Again, 6LoWPAN requires to send the compression information in every packet.

| Protocol | Size of ESP overhead (+ICV size) | Size of Encryption Overhead (IV) | Size of Inner Pv6 overhead | Size of UDP overhead | Size of IP payload (+ICV size) |
|---|---|---|---|---|---|
| ESP | 11+12 bytes | 8 bytes | 40 bytes | 8 bytes | 68+12 bytes |
| ROHC | 3+12 bytes | 8 bytes | 0 bytes | 0 bytes | 12+12 bytes |
| 6LoWPAN | 4+12 bytes | 8 bytes | 3 bytes | 1 byte | 16+12 bytes |
| Diet−ESP | 0+12 bytes | 0 bytes | 0 bytes | 0 bytes | 1+12 bytes |

**Table 6.15 Comparison of IP payload size when sending 1 byte of application data secured with AES−CTR.**

The additional savings of Diet−ESP against the other protocols, together with the simplified integration and improved usability makes it a valuable addition to the actual IoT protocols.

The potential savings are possible because of the high flexibility of the Diet−ESP context together with the possibility of exchanging the context at the beginning of the communication and the already existing information about the connection in the IPsec databases.

Together with the savings, Diet−ESP matches the requirements developed in section 4 as shown in the following.

*REQ 1: In order to benefit from this hardware support, security protocols for sensors MUST support AES ciphers be able to take advantage of AES−CCM hardware acceleration.*

Diet−ESP supports all ciphers specified for ESP. In the moment of writing this document, the four AES modes CBC, CTR, CCM and GCM are explicitly supported in [46].

*REQ 2: If encryption and authentication is enabled, a security protocol for sensors SHOULD be able to use AES−CCM as it is defined in IEEE 802.15.4 taking advantage of hardware acceleration for encryption and authentication.*

Diet–ESP does not modify how encryption occurs. It only changes the encrypted payload, which is one of the parameters for the encryption function. Therefore Diet–ESP is able to work with any encryption defined in [46] which also includes AES–CCM [25].

Combined Mode algorithm (e.g. AES–CCM, AES–GCM) have an additional parameter, called Addition Authentication Data (AAD). This AAD requires the uncompressed ESP header that is to say the full SPI and SN. These parameters are not removed by Diet–ESP are not removed before encryption/authentication takes place, why these ciphers are fully supported.

*REQ 3: Since networking is extremely expensive in IoT communications, padding MUST be prevented whenever it is possible. Therefore security protocols SHOULD support Byte–Alignment that are different from 32 bits or 64 bits to prevent unnecessary padding.*

With Minimal–ESP explicitly mentions the use of AES in Counter Mode, whenever it is possible. In addition Diet–ESP allows the remove of Padding by specifying alignments of 16 and 8 bits if AES–CTR is going to be used.

*REQ 4: In order to agree on the used alignment, each peer SHOULD be able to advertise and negotiate the Byte–Alignment, used for Diet–ESP. This could be done for example during the IKEv2 exchange.*

The Diet–ESP context contains the ALIGN field, which is used to negotiate the alignment between the peers. With the IKEv2 exchange of the Diet–ESP context, the sender can propose different supported alignments and the receiver will choose the value he can deal with.

*REQ 5: Diet–ESP SHOULD be able to reduce or remove all fields, directly related to the ESP protocol.*

The Diet–ESP context explicitly specifies all fields of the ESP header and trailer, which makes the compression of this fields possible. Table 6.16 shows how the different fields are compressed in order to fulfill REQ 5.1 – REQ 5.5.

| Field | Encoding Method | Diet–ESP Context Parameters |
|---|---|---|
| SPI | LSB | SPI_SIZE |
| SN | LSB | SN_SIZE |
| Padding | Removed | PAD, ALIGN |
| Pad Length | Removed | PAD, ALIGN |
| Next Header | Removed | NH |

**Table 6.16 Compression of the different ESP protocol fields.**

*REQ 6: Diet–ESP SHOULD allow compressions of upper layer protocols, e.g. protocols of the transport– or application layer.*

With the Diet–ESP extension for Upper Layer Compressions (see section 6.3) it is possible to compress IP, UDP, TCP and UDP–Lite headers inside the ESP payload. As the compression is defined with ROHC profiles, all now available or further ROHC profiles can be used for the compression even of application protocols, which do not necessary need the context negotiation defined by the ROHC framework. If application layer compression takes place over the transport layer, Diet–ESP supports this compression as well.

*REQ 7: Diet–ESP SHOULD NOT allow compressed fields, not aligned to 1 byte in order to prevent alignment complexity.*

With Diet–ESP all fields all compress fields are aligned to 1 byte.

*REQ 8: If a block cipher with an Initialization Vector (IV) is used (like AES) Diet–ESP SHOULD be able to minimize the value sent on the wire. If so, it MUST define a proper algorithm to remain uniqueness and unpredictability of the IV.*

With the Diet–ESP extension for IV Compressions (see section 6.4) it is possible to remove the IV from the Diet–ESP payload. Due to the use of PBKDF2 it defines a standard way of IV generation, which is proven to be secure by standardization.

*REQ 9: The developer SHOULD be able to specify the maximum level of compression.*

An application developer, who wants to secure his communication with Diet–ESP, can specify all Diet–ESP context he is willing to support. This mechanism allows minimum and maximum compression to be specified by the developer.

*REQ 10: Diet–ESP SHOULD be able to compress any field independent from another one.*

Due to the use of the Diet–ESP context, the compression of the ESP related fields, the IV and the different upper layer protocols, can be specified independent from any other field. This provides high flexibility to the application developer.

*REQ 11: Diet–ESP SHOULD be able to define different compression method, when appropriated.*

The different fields are compressed with different compression methods, for example LSB or removal. The compression method is chosen due to the properties and usage of the different fields.

*REQ 12: Each peer SHOULD be able to announce and negotiate the different compressed fields as well as the used method.*

During the negotiation of the Diet–ESP context, the peers can announce and negotiate the different supported contexts, which includes the fields which should be compressed.

*REQ 13: Diet–ESP SHOULD be able to be implemented with minimal complexity considering small implementation that implement only a subset of all Diet–ESP capabilities without requiring involving standard ESP, specific compressors and decompressors.*

As Diet–ESP does not need special negotiation functionality like ROHC only the compression itself has to be implemented by the device. This follows a straight forward way but it also allows optimization according to special use cases. As an example, the IP header compression must not be implemented if the device does not support Tunnel mode. Similarly, a developer can chose to only implement supported Transport Layer compressions, e.g. UDP.

The (de–) compression itself is defined in to different layers, payload and header compression. The (de–) compression of the ESP trailer, for example can be minimized by not building the full ESP trailer before compression occurs, but directly compress the different fields while the ESP trailer is built.

*REQ 14: Diet−ESP SHOULD provide default configurations, which can be easily set up by a developer.*

Section 6.2.3 defines a default context all Diet−ESP implementations have to support. If an application developer does not specify a specific context, this context is going to be used.

*REQ 15: Diet−ESP SHOULD be able to interact with Standard ESP implementations on a single platform, without the need of implementing a second ESP stack inside the device.*

Diet−ESP can be integrated to an existing ESP implementation as an add−on, leaving the core functionality of ESP untouched, if Diet−ESP should not be supported.

*REQ 16: Diet−ESP SHOULD be able to communicate with Standard ESP gateways, by producing RFC4303 conform output.*

With the context described in section 6.2.2 Diet−ESP is able to produce RFC4303 conform output.

*REQ 17: In order to keep compatibility to other compression protocols appearing at the MAC layer, Diet−ESP SHOULD be able to interact with IP compression protocols, without the need of modifying them.*

Diet−ESP is compatible to all compression protocols appearing between MAC and IP layer because these protocols are not able to compress the encrypted ESP payload. This statement remains true, even if one of this protocols compresses the ESP header, as Diet−ESP can be configured to produce a full standard compatible ESP header of 8 bytes.

# 7 Implementation

Diet–ESP described in section 6 is implemented in two different programming languages. The Python implementation results in a Minimal–ESP implementation, combined with a Diet–ESP implementation. The program includes some default configurations for the IPsec databases, networking setup and an additional AES–XCBC–MAC support. This code is less than 1,000 lines of code. The code for Diet–ESP itself is around 300 lines of code, including a basic ESP implementation.

The implementation described in section 7.2 is for Contiki [8], a famous sensor operating system. It shows how Diet–ESP can be integrated into an existing IPsec implementation with small overhead. The IPsec implementation including Diet–ESP results in only 500 Bytes of memory overhead, holding a RFC4301 [37] conform setup for the Security Policies.

## 7.1 Diet–ESP demonstrator in Python

ESP procession itself can be implemented straight forward like it is described in section 6.6. The complexity of the security protocol is the encryption implementation and the interaction with the IP implementation of the kernel. Python includes solutions for both difficulties. First, the PyCrypto [42] library offers a set of implemented cryptographic function. In the following work AES is used for encryption, as well as AES–XCBC and SHA1 for authentication. The library offers a well–documented way to implement new cipher algorithms, as it is shown with the implementation of AES–XCBC which is not part of the library, but it is implemented with the AES cipher of the library and behaves like other authentication algorithms. Since Python programs run in user space, however, there is no way to intercept the IP traffic of the kernel. But it is possible to extract the raw MAC–layer traffic from the network interface in order to work with the raw IP traffic for incoming packets. For outgoing packets the shown implementation uses RAW–Sockets [12, 48], allowing the sending of IP packets without interception of the kernel.

In the following subsections, the key parts of the implementation are described.

### 7.1.1 IPsec Databases

The IPsec databases SAD and SPD are implemented in python dictionaries. This implementation uses one dictionary called “SPD” (an example is shown in Listing 7.1) having different sub–dictionaries for IPv4 and IPv6 entries. Like described in RFC4301 there is another set of sub dictionaries for outgoing and incoming packets, for each of the previous dictionaries. Each of these entries has a traffic description, consisting of the IP address as the identifier for the entry, the transport protocol and the source and destination ports of the transport protocol. Furthermore the entry includes a link to one specific SAD entry with the SPI as identifier. An example for the second dictionary “SAD” is shown in Listing 7.2. It stores the cipher keys for encryption and authentication and is prepared for the AH protocol as well. Together with the keys, the used algorithm is specified for each entry. In addition, the currently set Sequence Number, the SA lifetime, support of anti–replay–protection, the IPsec Mode and a reference to the Diet–ESP (see Listing 7.3) configuration is stored inside each SA entry. For simplification, the Diet–ESP configuration is stored in a separate dictionary to avoid code duplication for equal configuration, but of course it remains possible to store one configuration for each entry by filling the Diet−ESP directly inside the SA.

```
SPD = {
    4:{ ## IPv4
        "outgoing": {
            "10.193.160.108": {
                "sa": SAD[0],
                "transport_protocol": socket.IPPROTO_UDP,
                "source_port": 65500,
                "dest_port": 65500
            }
        },
        "incoming": {
            "10.193.160.108": {
                "proto": socket.IPPROTO_IP,
                "sa": SAD[0],
                "transport_protocol": socket.IPPROTO_UDP,
                "source_port": 65500,
                "dest_port": 65500
            }
        }
    },
    6:{ ## IPv6
        "outgoing": {
            ip_address("aaaa::200:0:0:2").exploded: {
                "proto": socket.IPPROTO_IPV6,
                "sa": SAD[0],
                "transport_protocol": socket.IPPROTO_UDP,
                "source_port": 65500,
                "dest_port": 65500
            }
        },
        "incoming": {
            ip_address("aaaa::200:0:0:2").exploded: {
                "proto": socket.IPPROTO_IPV6,
                "sa": SAD[0],
                "transport_protocol": socket.IPPROTO_UDP,
                "source_port": 65500,
                "dest_port": 65500
            }
        }
    }
}
```

Listing 7.1 Implementation of outgoing and incoming SPD entries for IPv4 and IPv6.

```
SAD = {
    0: {
        "SN": 0,
        "anti_replay": False,

        "AH_INTEG": INTEG_ALGORITHM["HMAC_SHA1_96"],
        "AH_INTEG_KEY": b'1'*16,

        "ESP_ENC": ENC_ALGORITHM["AES_CTR"],
        "ESP_ENC_KEY": b'1'*20, # 16 bytes AES-key + 4 byte IV

        "ESP_INTEG": INTEG_ALGORITHM["HMAC_SHA1_96"],
        "ESP_INTEG_KEY": b'1'*16,

        "LIFETIME": 0,
        "IPSEC_MODE": IPSEC_MODE_TRANSPORT,

        "DIET_ESP": DIET_ESP_SA["diet1"]
    }
}
```

**Listing 7.2 SAD database with one entry as an example.**

```
DIET_ESP_SA = {
    "diet1":{
        "ALIGN": 0b0,
        "SPI_SIZE": 0b000,
        "SN_SIZE": 0b000,
        "NH": 0b1,
        "PAD": 0b1,
        "Diet-ESP_ICV_SIZE": DIET_ESP_ICV_TRUNCATION[0b000],
        "USE_SA": 0b1,
        "INNER_ROHC_PROFILE": 0x0002, #IP/UDP
        "CHECKSUM_LSB": 0b0,
        "COMPR_IV": 0b1,
        "IV_PRFT": 0b0001,
        "USE_PBKDF2": 0b1
    }
}
```

**Listing 7.3 Example of a Diet–ESP entry, supporting maximum compression.**

### 7.1.2 Cryptographic functions

Depending on the chosen algorithm stored inside the SA, the PyCrypto library has to be initialized. Listing 7.4 shows this initialization for outgoing packets secured with AES–CTR. First a random IV has to be generated, which could be done with the PRF function of section 6.4 as well. Than the Counter object, consisting of the last four bytes of the AES–CTR–key and the previously generated IV has to be generated. With the AES–key and the Counter, the cipher object can be initialized. All of these cipher objects have an “encrypt” function which performs the encryption. Listing 7.5 shows the quite similar functionality for authentication of the ESP–packet with the three supported algorithms “SHA1–HMAC”, “AES–XCBC” and “NULL”. Like for encryption a cipher object has to be generated using the authentication key.

```
enc = self.sa["ESP_ENC"]
if enc["NAME"] == "AES-CTR":
    iv = os.urandom(8)
    # counter = NONCE || IV || ONE @see RFC3686
    counter = self.sa["ESP_ENC_KEY"][-4:] + iv
    ctr = Counter.new(32, prefix=counter)
    cipher = AES.new(self.sa["ESP_ENC_KEY"][:-4], AES.MODE_CTR, counter=ctr)
else:
    print("Encryption %s is not supported"%enc["NAME"])
    return
enc = cipher.encrypt(esp_packet)
```

Listing 7.4 Initialization of the cipher object for AES–CTR encryption.

```
integ = self.sa["ESP_INTEG"]
if integ["NAME"] == "HMAC-SHA1-96":
    cipher = HMAC.new(key=self.sa["ESP_INTEG_KEY"], digestmod=MD5)
    cipher.update(esp_packet)
    icv = self.truncate_icv(cipher.digest())
elif integ["NAME"] == "NULL":
    icv = b''
elif integ["NAME"] == "AES-XCBC-MAC-96":
    cipher = XCBCMAC.new(key=self.sa["ESP_INTEG_KEY"], digestmod=AES)
    cipher.update(esp_packet)
    icv = self.truncate_icv(cipher.digest())
    print("Sended ICV: ", cipher.hexdigest())
else:
    print("Authentication %s is not supported"%integ["NAME"])
    return
```

Listing 7.5 Initialization of the cipher object for different authentication algorithms.

## 7.1.3 Networking

For sending and receiving ESP or Diet–ESP packets, the implementation uses the socket functionality provided by Python, which uses the socket functionality of the operating system. Since IPsec usually runs in kernel inside the IP implementation it has access to the whole IP traffic of the operating system. With python it is impossible to intercept the IP traffic itself, thus another solution has to be found.

### Receiving

For receiving packets, a physical socket is opened, with direct access to the traffic on the network card. Therefore the socket has to be bound to the physical device in this case "eth0".

```
sock = socket.socket(socket.PF_PACKET, socket.SOCK_RAW, socket.htons(3))

sock.setsockopt(socket.SOL_SOCKET, IN.SO_BINDTODEVICE,
  struct.pack("%ds"%(len("eth0")+1,),str.encode("eth0")))
```

After the bounding, the data and the address can be extracted from the interface. This can be done inside a while loop for example:

```
data, addr = sock.recvfrom(1024) # buffer size is 1024 bytes
```

The data includes the MAC packet, therefore the MAC header should be removed and the version of the IP header can be checked:

```
data = data[14:] #remove ethernet header
ip_ihl_ver = struct.unpack('!B', data[:1])[0] #extract IP version
```

Now that the IP version is known, it is possible to extract all fields from the IP header, according to the IP version.

```
if (ip_ihl_ver >> 4) == 4:
    print("Received IPv4 packet")
    (ip_ihl_ver, ip_tos, ip_tot_len, ip_id, ip_frag_off, ip_ttl, ip_proto,
        ip_check, ip_saddr, ip_daddr) = struct.unpack('!BBHHHBBH4s4s',
        data[:20])
elif (ip_ihl_ver >> 4) == 6 :
    print("Received IPv6 packet")
    (row, ip_payload_len, ip_proto, ip_hop_limit, ip_saddr, ip_daddr) =
        struct.unpack('!LHBB16s16s', data[:40])
else:
    continue
```

The values of the IP header can be processed like described in the specific RFCs. However, at the end of the procession the last Next Header field is checked for ESP. With this information, the IP header will be removed leaving the ESP packet for encapsulation. In this demonstration, the decrypted content is print on the console and not forwarded. If one wants to forward the encapsulated packet, he has to send the ESP payload to a local socket.

```
if source_ip.version == 4:
    hdr_len = (ip_ihl_ver % 0x10) * 4 # 4 bit of IHL_VER * 32 bit
    esp_data = data[20:]# remove ip header
    esp_data = esp_data[:(ip_tot_len-hdr_len)]
elif source_ip.version == 6:
    esp_data = data[40:]
    esp_data = esp_data[:ip_payload_len]
else:
    print("unsupported ip protocol")
    continue
print("Content of ESP packet:")
print(Diet_ESP_Daemon(dest_ip,source_ip,incoming=True).process_incoming_packet
    (esp_data))
```

### Sending

Sending packets is simpler than receiving, but there is a difference between IPv4 and IPv6. The IPv4 stacks of Linux and Windows support the “socket.IP_HDRINCL” option, defining that the IP header is included in the data sent to the RAW socket. In IPv6 this option remains without effect to the resulting packet, which is why the packet has to be sent in another way.

Opening a RAW_Socket is the same for IPv6 and IPv4, with the difference of setting the further described option:

```
#IPv4
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ESP)
#IMPORTANT: enables application made ip headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

#IPv6
s = socket.socket(socket.AF_INET6, socket.SOCK_RAW, socket.IPPROTO_ESP)
```

For IPv6 the whole configuration is done with this step, which makes modifications of the IP header not feasible. In IPv4 one can construct his own IP header with all the fields he want to fill, like in the example in Listing 7.6. For IPv4 the packet to be sent includes the IP header, for IPv6 it includes only the ESP header. It can be sent to the socket with the following command.

```
s.sendto(packet, (self.dest_ip, 1))
```

```
# ip header fields
ip_ihl = 5 # IP header length
ip_ver = 4 # IP version 4
ip_tos = 0
ip_tot_len = 0  # kernel will fill the correct total length
ip_id = 54321   #Id of this packet
ip_frag_off = 0
ip_ttl = 255
ip_proto = socket.IPPROTO_ESP
ip_check = 0    # kernel will fill the correct checksum

# Transform the adresses
ip_saddr = socket.inet_aton(self.source_ip)
ip_daddr = socket.inet_aton(self.dest_ip)

# ip_ver and ip_ihl are 4 bit each, so put them together in one byte
ip_ihl_ver = (ip_ver << 4) + ip_ihl

# Build IP-Header
ip_header = struct.pack('!BBHHHBBH4s4s' , ip_ihl_ver, ip_tos, ip_tot_len,
    ip_id, ip_frag_off, ip_ttl, ip_proto, ip_check, ip_saddr, ip_daddr)

# build packet
packet = ip_header + user_data
```

**Listing 7.6 Building of a user defined IPv4 header.**

### 7.1.4 Diet–ESP procession

The relevant parts for compressing the ESP header and ESP payload are defined as two different layers. But for a minimal and efficient implementation the compression can be done instantly when the packet is built.

First the ESP Payload (IP and Transport Layer Headers) can be compressed. As an example, the UDP header can be compressed. The compression uses the information of the Diet–ESP context. First the protocol and the inner ROHC profile is checked. If this is UDP, the ports and the UDP–length can be removed. The checksum of the UDP–header is compressed, depending on the “CHECKSUM_LSB” value.

```
def compress_transport_header(self, data, diet_esp):
  if self.sp["transport_protocol"] == socket.IPPROTO_UDP and
     diet_esp["INNER_ROHC_PROFILE"] in [0x0001, 0x1001, 0x1002, 0x0002,
     0x0008, 0x1008]:
      data = data[6:]
      if diet_esp["CHECKSUM_LSB"] == 0:
          return data[2:]
      elif diet_esp["CHECKSUM_LSB"] == 1:
          return data[1:]
      else:
          return data
```

In order to build the packet, the struct.pack [63] function of Python is used. It allows constructing a string which handles the encoding of the date in the correct order (e.g. network byte order). It allows 1, 2, 4 and 8 byte values, in general numeric values, strings and padding bytes. The following code snippet shows the construction of the Diet–ESP content, Padding, PadLength and NextHeader. The payload "%ds" defines the payload with length of the app_data. The "%dx" defines pad_length padding bytes. Pad Length and Next Header are 1 Byte each ("L") and filled depending on the "PAD" and "NH" value of the Diet–ESP context.

```
packet_struct = '!%ds%dx'% (len(app_data),pad_length)
# PAD LENGTH and NEXT HEADER
if self.diet_esp_sa["PAD"] == 0b1:
    packet_struct += 'L'
if self.diet_esp_sa["NH"] == 0b0:
    packet_struct += 'L'
```

With this string one can use the struct.pack function to build the Diet–ESP payload, depending on the information of the Diet–ESP context.

```
if self.diet_esp_sa["PAD"] == 0b1 and self.diet_esp_sa["NH"] == 0b0:
    esp_packet = struct.pack(packet_struct, app_data, pad_length,
self.sp["transport_protocol"])
elif self.diet_esp_sa["PAD"] == 0b1:
    esp_packet = struct.pack(packet_struct, app_data, pad_length)
elif self.diet_esp_sa["NH"] == 0b0:
    esp_packet = struct.pack(packet_struct, app_data, self.sp["transport_pro-
tocol"])
else:
    esp_packet = struct.pack(packet_struct, app_data)
```

Now that the payload is built, it can be encrypted like shown before and the ESP header can be built. Like before, this values are compressed instantly when they are built but the ICV is built before in order to include the uncompressed header.

```
# Authentication
self.sa["SN"] = self.sa["SN"] + 1
header = struct.pack('LL', self.sa["SPI"], bytes([self.sa["SN"]]))

if integ["NAME"] == "NULL":
    icv = b''
else:
    cipher.update(header + esp_packet)
    icv = self.truncate_icv(cipher.digest())

esp_packet = esp_packet + icv

# Add ESP headers
if self.diet_esp_sa["SPI_SIZE"] > 0b0 and self.diet_esp_sa["SN_SIZE"] > 0b0:
    header = struct.pack(self.build_struct_for_esp_header(), self.sa["SPI"],
bytes([self.sa["SN"]]))
elif self.diet_esp_sa["SPI_SIZE"] > 0b0:
    header = struct.pack(self.build_struct_for_esp_header(), self.sa["SPI"])
elif self.diet_esp_sa["SN_SIZE"] > 0b0:
    header = struct.pack(self.build_struct_for_esp_header(),
bytes([self.sa["SN"]]))
else:
    header = b''

esp_packet = header + esp_packet
```

Receiving packets works in exactly the same way. The string for the struct is built in the same manner, but the *struct.unpack* function is used instead of *struct.pack*.

## 7.2 Diet–ESP Add–On for Contiki IPsec

Contiki is an operating system, especially designed for IoT devices. Compared to a regular computer operating system, one can say that it is more a collection of functionalities, which are partially compiled for the specific use cases and hardware. One reason for using Contiki in this work is, that it provides a fully compatible but reduced IP and IPv6 stack, called µIP. In his master thesis Vilhelm Jutvik [32] developed an RFC conform ESP implementation for this operating system, which is available as open source. Using such an implementation as an entry point for developing Diet–ESP has the advantage that the functionality of ESP already exists and Diet–ESP can be integrated as an Add–On. Additionally it approves the possibility to integrate Diet–ESP to an ESP implementation.

Another advantage of Contiki is that it is quite easy to extract parts to other hardware. The Orange SensOrLabs in Grenoble ported the Contiki IPsec stack to an Excelyo [10] platform[1]. For testing the Diet–ESP performance on a real platform, the following work was ported to this platform, but unfortunately this sensor has such limited storage that it was not successful. However, it was possible to reduce the code size to only 500 Bytes (see Table 7.1), including a single policy for securing one connection, which is sufficient for simple IoT scenarios. Having a look on the memory print, which is automatically generated by the compiler, one can compare ESP enabled with ESP disabled binaries. Without IPsec the binary has a size of 4032 Bytes versus 4600 Bytes with the Diet–ESP add–on enabled. Therefore the code size of IPsec + Diet–ESP is exactly 568 Bytes with space for one incoming and one outgoing static allocated SAD entry.

[1] Orange Labs in Grenoble decided to build a own sensor platform, similar as described in [54] for observing the network behavior from the periphery. The result is a fully observable platform called SensOrLabs.

| Implementation | Size in hex | Size in Bytes |
|---|---|---|
| without IPsec and Diet–ESP | 0x0fc0 | 4032 |
| with IPsec and Diet–ESP | 0x11f8 | 4600 |

**Table 7.1 Memory footprint of different implementations.**

One of the most important files for a Contiki developer is the *contiki–conf.h* file. As its name implies, it holds configurations for the compilation and was extended by some IPsec specific configuration. Usually this file is placed in the same hierarchical order as the program written by the application developer. Some example programs for basic functionalities can be found in the */examples* directory.

The Contiki IP stack is located in the *core/net* directory (see Figure 7.1). The IPsec code is placed in different files inside the *ipsec* subfolder (see Figure 7.2) and in the *uip6.c* file. This file represents the integration of IPsec into the IP stack, usually placed in the kernel. Here, every IP packet can be inspected itself. Therefore this file includes the SPD lookup for outgoing and incoming packets. If a policy matches the IP packet, ESP procession is started or the IP packet is dropped otherwise. The ESP procession in this file includes the building of the ESP header and the correct call of the encryption and authentication functionality.

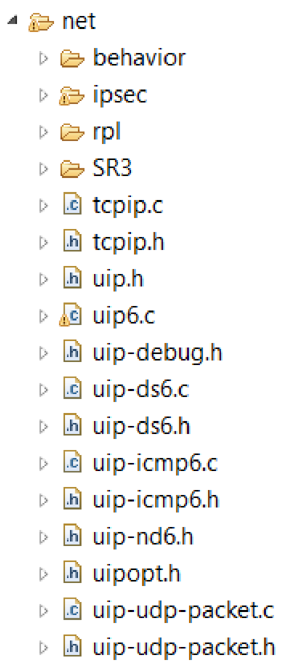


**Figure 7.1 The Contiki */net* directory structure.**

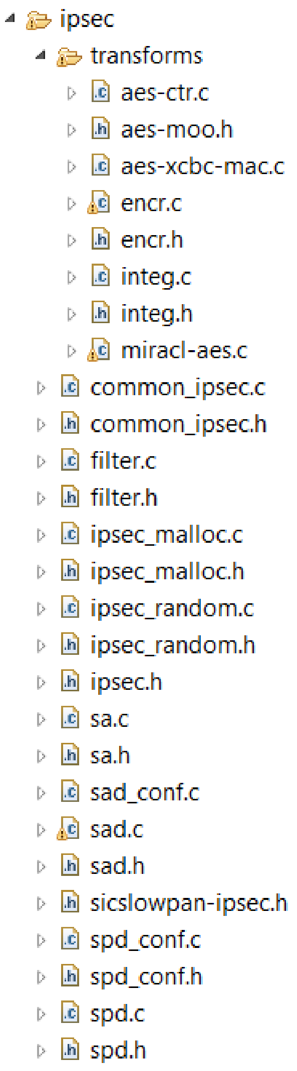
The IPsec databases are located inside the *ipsec* directory, including configuration files for static databases as well. They have the functionality to dynamically or statically allocate memory for the entries. The static allocation was added as part of this work, as it saves a lot of memory preventing standard functions like *malloc*.

The encapsulation of the ESP payload is done during the encryption, in the *transforms/encr.h*. It has a wrapper functionality for all supported ciphers, in the current version only AES–CTR. There are two possibilities of AES encryption supported. First, there is a software implementation, called “Miracle AES”. It consists of a number of pre–computed values in order to prevent high computation overhead. The other possibility is a hardware supported ciphering with the CC2420 interface [76]. It is a wireless interface including hardware acceleration for AES like already described in section 2.1.

The Diet–ESP specific implementation is included in the following parts of the IPsec implementation:

- **sad.h:** includes the definition of the Diet–ESP context and the appropriated pointer in the sad_entry to a specific Diet–ESP context.
- **sad.c:** includes new function for reading and writing the compressed ESP header, together with the changed incoming SAD lookup for compressed ESP headers.
- **encr.c:** includes the compression of the ESP payload and the functionality for removing and restoring of IP and UDP headers.
- **uip6.c:** the calls of the new functionalities (e.g. the ESP header compression) are added to this file.

- **contiki–cont.h:** This file includes the necessary configuration for ipsec, that is to say:
  - **WITH_CONF_IPSEC_ESP:** compilation of ESP support.
  - **WITH_CONF_IPSEC_AH:** compilation of AH support (not supported in the current version)
  - **WITH_CONF_IPSEC_IKE:** compilation of IKE support.
  - **WITH_CONF_MANUAL_SA:** manual SAD configuration. It can work together with IKE.
  - **WITH_CONF_IPSEC_MALLOC:** support of static or dynamic allocation of the storage for the SAD database.
  - **IPSEC_SAD_INCOMING_SIZE:** The size of the static allocated incoming SAD.
  - **IPSEC_SAD_OUTGOING_SIZE:** The size of the static allocated outgoing SAD.



```
ipsec
  transforms
    aes-ctr.c
    aes-moo.h
    aes-xcbc-mac.c
    encr.c
    encr.h
    integ.c
    integ.h
    miracl-aes.c
  common_ipsec.c
  common_ipsec.h
  filter.c
  filter.h
  ipsec_malloc.c
  ipsec_malloc.h
  ipsec_random.c
  ipsec_random.h
  ipsec.h
  sa.c
  sa.h
  sad_conf.c
  sad.c
  sad.h
  sicslowpan-ipsec.h
  spd_conf.c
  spd_conf.h
  spd.c
  spd.h
```

**Figure 7.2 The Contiki */ipsec* directory structure.**

## 7.3 Experimental Set–Up

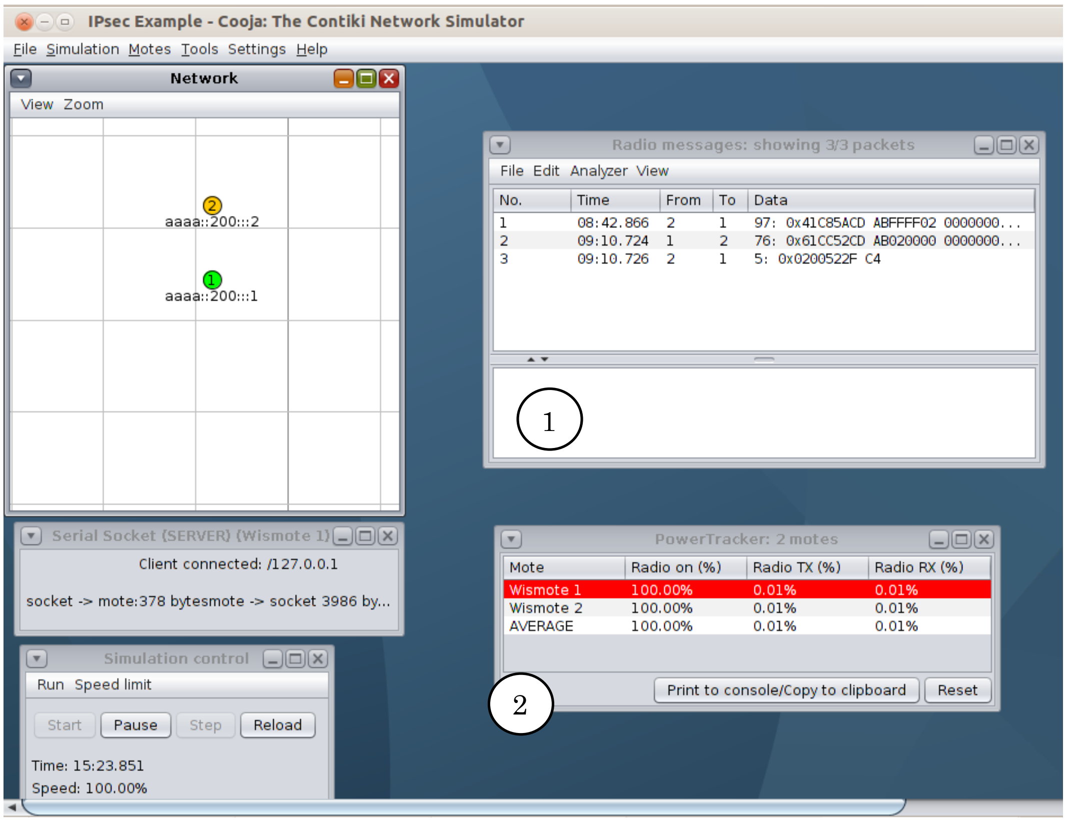
The Contiki Diet–ESP implementation was tested with the Cooja simulator, provided by Contiki. The IPsec implementation for Contiki provides a preconfigured Cooja simulation file, which can be used for the tests with Diet–ESP as well. This simulation covers two sensors, one router and one IPsec endpoint. The router's task is to enable packet forwarding between the operating system, running the simulator and the sensor network. It is a program included to every up to date Contiki system and placed in the folder *example/ipv6/rpl–border–router*. It has to be run inside the Linux system and creates a new *tun0* network interface, routing the packets to the *rpl–boarder–router* (Node 1 with IP address "aaaa::200:::1" in Figure 7.3) in the Cooja simulator. This router sensor forwards the packets to the connected sensors in the sensor network. It makes all the sensors accessible through the *tun0* interface with their given IPv6 addresses. The IPsec endpoint is an IPsec enabled sensor with an example application. It opens a UDP socket on port 1234 awaiting packets (Node 2 in Figure 7.3). If a packet is received from the sensor it increases every byte from the UDP payload by one and responses this value. For example if one is sending a UDP packet with the payload "1234" to the sensor with IP address "aaaa::200:0:0:2", it will response with "2345". If the response arrives correctly at the initiator of the connection, the transmission was successful received by the sensor and sent packet accurately. This configuration allows to run some basic tests with the Cooja simulator.

First, the interaction of Diet–ESP with regular ESP from the Linux Kernel, is tested. This provides information about the requested functionality of compatibility of Diet–ESP with a regular ESP implementation if the "Standard ESP compliant" Diet–ESP context is used. Tests are performed with both implementations of Diet–ESP. The Python implementation running on Linux and a Contiki implementation running on the same device in the Cooja simulator. Therefore the Linux Kernel IPsec databases are set up with a pre–shared secret using the *ip xfrm [43]* command and a packet is sent to the Contiki implementation with the command *nc -u aaaa::200:0:0:2 1234*. For the python implementation the same test can be performed with the IP address of the Linux system.

Second, the Diet–ESP Contiki implementation and the Diet–ESP Python implementation are interconnected. This provides indication for the proper encapsulation of the packets on both sides, and proves the functionality of Diet–ESP embedded to the regular ESP implementation of Contiki. With this second configurations it is possible to perform a basic energy measurement.

As mentioned earlier, the energy consumption for networking is highly correlated to the number of bits going to be sent, especially in wireless communications. Cooja offers two measurement methods. One is a packet sniffer (see ① in Figure 7.3), showing the contents and lengths of packets sent over the network interface of the sensors. Additionally it is possible to measure the simulated radio module, providing the time the module spent in receiving and transmission state (see ② in Figure 7.3). Comparing these two values for networking for different compression levels give some indication about the energy consumption of the network interface. Having a specific network interface for a hardware, one can pick out the specifications for energy consumption and calculate the absolute values for energy saving.

The measurements are done with the current implementation for Contiki, which is able to (de–) compress the ESP related fields and the transport layer header, but not the Initialization Vector. The other side of the Diet–ESP connection is represented by the Python implementation, which is able to provide the same level of compression as the Contiki implementation. The tests are done by sending a UPD packet with the payload "1234" (4 bytes) from the Python implementation to the sensor (Node 2 in Figure 7.3)


IPsec Example - Cooja: The Contiki Network Simulator
File Simulation Motes Tools Settings Help
Network
View Zoom
2
aaaa::200:::2
1
aaaa::200:::1
Radio messages: showing 3/3 packets
File Edit Analyzer View
1
Serial Socket (SERVER) (Wismote 1)
PowerTracker: 2 motes
2
Simulation control
Run Speed limit
Start
Pause
Step
Reload
Print to console/Copy to clipboard
Reset

| No. | Time | From | To | Data |
|---|---|---|---|---|
| 1 | 08:42.866 | 2 | 1 | 97: 0x41C85ACD ABFFFF02 0000000... |
| 2 | 09:10.724 | 1 | 2 | 76: 0x61CC52CD AB020000 0000000... |
| 3 | 09:10.726 | 2 | 1 | 5: 0x0200522F C4 |

Client connected: /127.0.0.1

socket -> mote:378 bytesmote -> socket 3986 by...

| Mote | Radio on (%) | Radio TX (%) | Radio RX (%) |
|---|---|---|---|
| Wismote 1 | 100.00% | 0.01% | 0.01% |
| Wismote 2 | 100.00% | 0.01% | 0.01% |
| AVERAGE | 100.00% | 0.01% | 0.01% |

Time: 15:23.851
Speed: 100.00%

Figure 7.3 User interface of the Contiki–Simulator "Cooja".

receiving this packet and responding with a UDP packet with the payload “2345” (4 bytes). This leads to the theoretical compression with Diet−ESP on Contiki as shown in Table 7.2. The size of the IPv6 header is read from the packet sniffer at the sensor. The received packet at Node 2 is greater than the minimum size IPv6 header size of 40 bytes, since there are some routing information added by the *rpl−border−router*. For sending from Node 2 to the Python implementation, the IPv6 header is smaller, since the routing information are added later at the *rpl−border−router* and the µIP stack provides some basic optimizations of the IPv6 addresses, which are later extracted by the *rpl−boarder−router*.

The theoretical evaluation of compression in Table 7.3 and Table 7.4 considers that time and energy are proportional to the number of byte of the packet. The measured data from the simulator prove that the reduction of the packet size is like expected. Having a look on the measured times, the device is spending for sending and receiving, there are two conclusions. The time for sending decreases to a maximum of around 21% against the theoretical saving of around 26% whereas the time for receiving is reduced by 15% against the theoretical value of 23% for maximum possible compression of 20 bytes. The maximum compression for the expected packet would be 28 bytes, if the IV would have been removed as well. Figure 7.4 and Figure 7.5 show the expected savings for sending and receiving by extending the measured values with logarithmic estimations. These graphs also show the variances between theoretical and measured values. The growing energy consumption graphs resembles a linear function, which is like the theoretical function. Even the logarithmic extension of the energy saving is evocative to a linear extension. Therefore it is possible to assume that the energy consumption will decrease linear to the number of bytes saved by compression with Diet−ESP.

The measurement results prove that Diet−ESP is able to reduce the power consumption significantly, but there remains a static part of around 20−40% (as shown in the last column in Table 7.3 and Table 7.4) which cannot be removed by compression. The static part is due to network specific overheads. One of them is the MAC layer overhead, which is neither be removed nor measured. Another reason is the time the sensor has to wait until it is getting a slot for receiving or sending. For receiving, this static part is bigger, since the device is not able to know when it should receive a packet. It will wake up periodically or will receive a notification, if there is a packet to be received. Once it woke up, it has to wait for a slot until it is allowed to receive the content.

Nevertheless it is possible to approximate the energy saving with the number of bytes removed by compression. Assuming the energy consumption decreases linear to the number of bytes removed by the compression of Diet−ESP, maximum compression enables the reduction of the energy consumption on around 40% (see Table 7.5). The table assumes a regular IPv6 header of 40 bytes holding a UDP packet with a 15 bytes payload. As AES−CBC with a necessary 16 bytes alignment is used, this payload together with the ESP trailer leads to 15 bytes of padding and an ESP payload size of 80 bytes + ICV. With Diet−ESP, the ESP payload is just the UDP payload size of 15 bytes + the Pad Length field with the value 0, in order to fill the necessary 16 bytes alignment of AES−CBC. This leads to an overall ESP payload size of 28 bytes + ICV, and a saving of 88 bytes for the whole ESP packet. From this saving of 56% the average difference between the real and theoretical energy consumption identified during the measurements is subtracted producing a saving of around 40%. With an energy saving of 40% as shown in Table 7.5, the life time of a sensor would have been increased by 80% which nearly doubles the lifetime of the device.

However, as the compression rate highly depends on the whole packet size, the values have to be seen in a very specific context. Considering the use of 6LoWPAN which compresses the IPv6 header to only two bytes instead of 40, the percentage saving is even higher, since the relation between packet size and compressed bytes will grow up significantly. As an example, the size of the uncompressed packet in Table 7.2 for receiving would be 48 bytes (84 bytes – 40 bytes (IPv6 header) + 2 bytes (6LoWPAN header)). The size of the compressed packet would be 28 bytes, resulting in a compression rate of 58%.

| Field | Uncompressed Size (bytes) | Compressed Size (bytes) | Difference |
|---|---|---|---|
| IPv6 header receiving | 42 | 42 | 0 |
| IPv6 header sending | 33 | 33 | 0 |
| ESP SPI | 4 | 0 | 4 |
| ESP SN | 4 | 0 | 4 |
| AES–CTR IV | 8 | 8 | 0 |
| UDP source port | 2 | 0 | 2 |
| UDP destination port | 2 | 0 | 2 |
| UDP length | 2 | 0 | 2 |
| UDP checksum | 2 | 0 | 2 |
| UDP payload | 4 | 4 | 0 |
| ESP Padding | 2 | 0 | 2 |
| ESP Pad Length | 1 | 0 | 1 |
| ESP Next Header | 1 | 0 | 1 |
| ICV | 12 | 12 | 0 |
| Σ (receiving) | 86 | 66 | 20 (23.26 %) |
| Σ (sending) | 77 | 57 | 20 (25.97 %) |

**Table 7.2 Theoretical compression with Diet–ESP in the test bed.**

| Compressed Bytes | Packet Size (bytes) | Saving (bytes) | | Sending Time (µs) | Saving (µs) | | Diff. |
|---|---|---|---|---|---|---|---|
| 0 | 77 | 0 | 0,00% | 3391 | 0 | 0,00% | |
| 8 | 69 | 8 | 10,39% | 3072 | 319 | 9,41% | 9,46% |
| 12 | 65 | 12 | 15,58% | 2944 | 447 | 13,18% | 15,42% |
| 20 | 57 | 20 | 25,97% | 2686 | 705 | 20,79% | 19,96% |

**Table 7.3 Energy Savings for sending.**

| Compressed Bytes | Packet Size (bytes) | Saving (bytes) | | Receiving Time (µs) | Saving (µs) | | Diff. |
|---|---|---|---|---|---|---|---|
| 0 | 86 | 0 | 0,00% | 9049 | 0 | 0,00% | |
| 8 | 78 | 8 | 9,30% | 8473 | 576 | 6,37% | 31,57% |
| 12 | 74 | 12 | 13,95% | 8281 | 768 | 8,49% | 39,18% |
| 20 | 66 | 20 | 23,26% | 7704 | 1345 | 14,86% | 36,09% |

Table 7.4 Energy Savings for receiving.

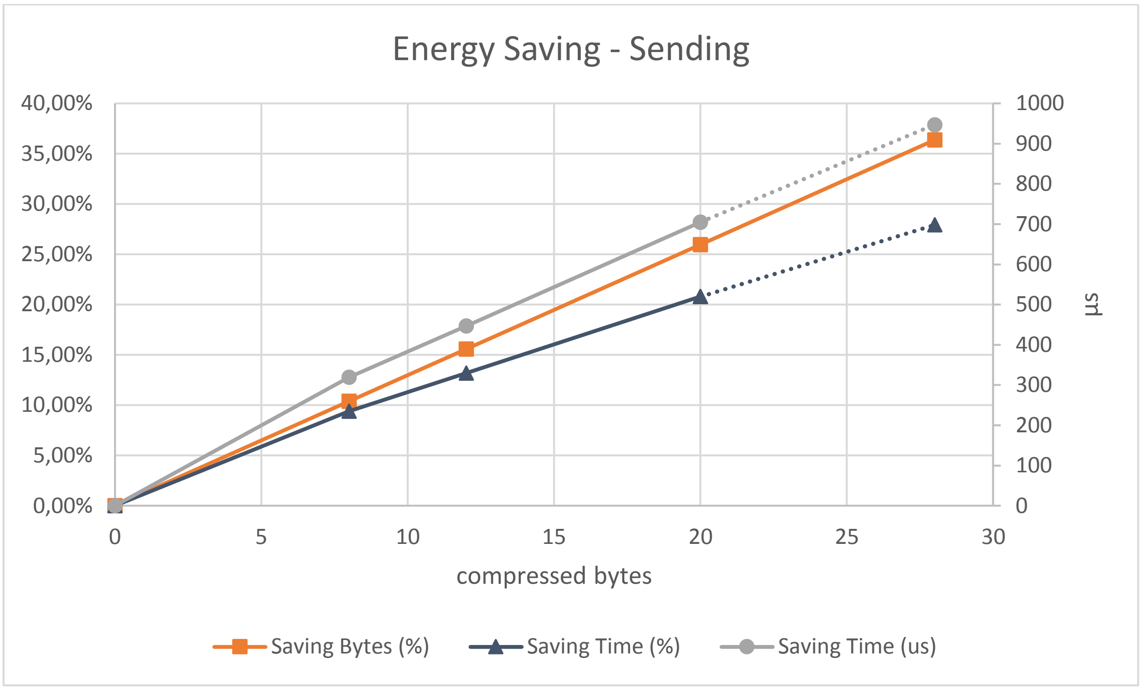


Figure 7.4 Energy consumption related to the bytes saved by compressing outgoing packets.

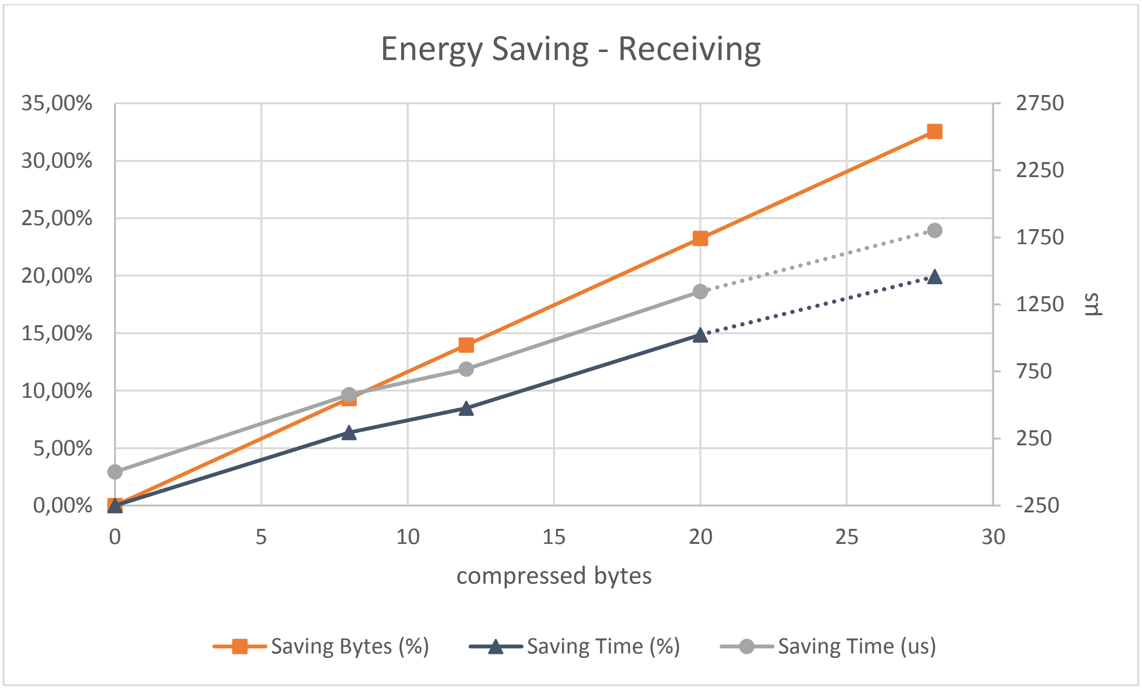


Figure 7.5 Energy consumption related to the bytes saved by compressing incoming packets.

| **Field** | **Uncompressed Size (bytes)** | **Compressed Size (bytes)** | **Difference** | **Energy Saving** |
|---|---|---|---|---|
| Minimum IPv6 header | 40 | 40 | 0 | |
| ESP header | 8 | 0 | 8 | |
| AES-CBC IV | 16 | 0 | 16 | |
| Tunnel IPv6 header | 40 | 0 | 40 | |
| UDP header | 8 | 0 | 8 | |
| UDP payload | 15 | 15 | 0 | |
| ESP Padding | 15 | 0 | 15 | |
| ESP trailer | 2 | 1 | 1 | |
| ICV | 12 | 12 | 0 | |
| Σ | 156 | 68 | 80 | 56.41 % |
| −Static part (~30%) | | | | 39.49 % |

**Table 7.5 Theoretical maximum compression with Diet-ESP.**

# 8 Discussion

Securing IoT communication is a widely discussed topic. There were a couple of investigations, exploring the possibility of the use of security protocols for low powered devices. [30] and [22] focused on the encryption overhead, while [49] where focusing on the differences between the distinct protocol solutions in combination with different cipher algorithms. Migault et al. [49] measured the performances of different Homenet devices, starting with a low powered ARM processor over an energy efficient Intel® Atom to a high performance Intel® i5. The measurements include several IPsec and TLS configurations within a closed configuration environment. The results were made by collecting user relevant data like download time or CPU occupancy. Especially the results for the ARM processor are promising, since it is applicable IoT devices needing high computation performances.

One important assertion from this paper is the performance differences between the AES modes for ESP. The results show that AES−CTR and AES−CBC have quite similar performances. This supports the statements of this thesis, saying that the Counter Mode should be used whenever it is possible in order to reduce the packet overhead caused by the fixed block size of CBC mode. The requirements for Diet−ESP defines the use of AES due to the possible hardware acceleration in microcontrollers. The measurements of the Intel® i5 processor, also supporting AES hardware acceleration, show that there is an advantage with the factor 1.5 to 2 relating to the time and CPU consumption, which can be directly applied to sensors. IoT devices usually try to use the sleep mode, when they idle. Every minute saved in computation will directly extend the lifetime of the device. Although, Migault's investigations also show the significant cryptographic overhead. The analyses include a configuration in which not encrypted packets (ESP NULL encryption) are compared with the AES−CBC encrypted packets. The evaluation shows the overhead between encryption and ESP packet procession, meaning that the encryption is 3 times more time and computation expensive. Another result supporting this hypothesis, is the networking overhead of ESP, which does not produce significant overhead in comparison to not encrypted but authenticated packets. The measurement is done by comparing not encrypted ESP packets with clear text HTTP packets.

The most important conclusion of Migault et al. relative to this thesis, is the comparison of ESP and TLS. Considering the different design of TLS and ESP, TLS should provide much better performance than ESP. This is due to the different network layers the two protocols are located in. Using TLS, the application data is secured, for example the payload of an HTTP packet. Even with bigger payloads, the whole data is encrypted and decrypted only once. In contrast, ESP is securing each IP payload, whose size is maximum $2^{16}$ bytes long (ca. 64 KB), as the Payload Length field of IP has 16 bits. In IPv6 there is an extension, called "Jumbograms" [4] supporting an extended size of $2^{32}$ bytes (ca. 4 GB). Nevertheless, this has to be supported by the transport layer, otherwise the transport layer will fragment the application payload to $2^{16}$ bytes as well. Overall one can say that the ESP payload has usually a maximum size of around 64 KB, whereas in ESP the encryption and decryption has to be performed much more often than in TLS, if the application data is bigger than 64KB. However, the measurements show that there is no significant performance improvement of TLS, especially on the low powered ARM and Intel® Atom processors which are more similar to the processor in IoT devices. Only the Intel® i5 shows significant performance benefits of TLS, which is most likely caused by the highly optimized TLS implementation for this family of processor.

Summarized, this analysis shows that ESP is an adequate protocol in order to secure IoT communication, as there are no expected performance differences between TLS and ESP, especially since IoT communication most usually deal with small packet sizes, lower than 128 bytes. This eliminates even the theoretical performance advantage of TLS.

The design of Diet−ESP picks up the results of these measurements with the requirements for specific ciphers. Since the procession of the ESP packet does not need significant performance, the only assumption to be made is the minimization of the code, provided by the Minimal−ESP design. Thus, the latter optimization to be made is the optimization of the packet overhead produced by ESP and upper layer protocols. This is achieved by requiring all unnecessary protocol overhead inside the ESP packet to be removed or reduced in a maximum flexible and usable way. The design of Diet−ESP is guided by this requirement, using the ROHC context as an already existing and proven concept. Due to the reuse and modification of this context, the compression rate provided by Diet−ESP can be extremely high, even though all ROHC mechanism needed for maintaining the context are removed and exchanged only at the beginning of the communication. Reflecting the results of the tests, Diet−ESP is able to fulfill the requirements providing an easy to handle and flexible security framework for IoT devices. The location of ESP inside the IP layer can provide a firewall−like behavior enabling secure communication to all configured connections. With maximum compression, Diet−ESP is able to compress all protocol overhead, except the application protocol and the application data.

# 9 Conclusion

The more IoT devices affect the public, the more people are asking for security considerations. Since devices in IoT are small and constrained in resources, securing them is a significant challenge for researchers. A lot of different attacks breaching such networks have been already described before [22, 30]. In today's sensor networks, these breaches remain controllable, since the networks usually have exactly one functionality in a closed environment. But with the increasing number of sensor network, accompanied by their increasing spacious and interconnectivity, the networks become uncontrollable leaving an open door for attackers. However, the challenges of securing the networks are similar to the former challenges while securing the World Wide Web, which is why there is a lot of know−how among experts.

Generally speaking, security has to be ensured in different levels during the development of such sensor networks, which are equal to the ones defined for nowadays networks like the Internet. There are a lot of standards, like the OSI−Security Architecture [79], the ISO IEC 27000 series [29] or the OWASP Developer Guide [53]. All of them define similar layers of security, but differ in their detailed description and differentiation between them.

In terms of the OSI−Security Architecture, the Access Control is one basic security layer that has to be provided by the manufacturer in terms of hardware manipulation and by the users by securing the access to the location of the device. Authentication is another layer that has to be provided by the infrastructure. X.509 [9] is a format supporting a public key infrastructure (PKI). Attempts have also been made to adopt the PKI used in the WWW to sensor networks [81].

The remaining three layers of the OSI−Security Architecture are Data Confidentiality, Data Integrity and Non−Repudiation, which can be provided by the security protocols TLS, DTLS, ESP and AH (see section 2.2 and 5.1). Confidentiality is provided by encryption, integrity with checksums and Non−Repudiation with Message Authentication Codes (MAC). This thesis is supporting current investigations of porting these network protocols for sensor networks by providing an optimized version of ESP called Diet−ESP, supporting the same security features as ESP.

The major challenges for adopting network protocols for sensors are the constraints of the device, basically in terms of energy, computation and storage. This requires optimization of protocols and implementations. Implementation can be easily optimized by extracting only necessary features as done in several minimal implementation of current standard protocols (e.g. Minimal IKEv2 [39]). In contrast, the optimization of protocols require investigations of compressing the protocol without loss of information and interoperability they were designed for. There are a couple of existing ideas for compressing all of the related security protocols by using compression frameworks like ROHC or 6LoWPAN. Nevertheless, they do not provide the maximum compression levels, they are inflexible in compression handling and complicated to embed into the existing protocols. For this reason, this thesis takes another path by designing a compression add−on for ESP providing two major differences with existing compression protocols. First, it is designed for enabling simple integration to existing ESP implementation. Additionally, the compression level is negotiated out of band of the ESP protocol by defining a context which is exchanged during the always necessary key exchange. As opposed to the other compression protocols, since Diet−ESP is an ESP add−on, it has direct access on the IPsec databases, which may contain information that can be used for optimizing the compression. Diet−ESP uses this information for the compression of protocols inside the ESP payload, that is to say the transport and IP layer protocol. Proceeding like that enables high level compression by exchanging only a couple of bytes at the beginning of the transaction, conserving the compression context throughout the whole lifetime of the Security Association even across reboots or shut downs of the device.

The results for energy saving and packet reducing show that Diet−ESP is working as expected. The implementations in Python and Contiki provide examples, how to implement the compression and prove the easiness of the integration to an existing ESP implementation. The measurements provides an impression of the expected energy savings with Diet−ESP, but the percentages are only estimations with a simulator. However, since the number of compressed bytes is directly associated with the Diet−ESP context, it is easy to calculate the energy savings of Diet−ESP for a specific use case with specific hardware. The port of the Contiki implementation to the Excelyo platform shows that ESP itself together with the Diet−ESP add−on works without significant memory usage. Unfortunately, even 568B of memory usage were too much for the Excelyo sensor, thus the tests must be shifted to future development. This could contain the minimization of the current implemented features on the Excelyo platform or the deployment within a new platform.

Future investigations for the protocol should be a standardization of the Diet−ESP context at the standardization committees, like the IETF. The standardization process will clarify questions, like possible security flaws by security and protocolling experts. There is also space for improvements of the Diet−ESP context. The design is very conservative so far, as it contains all relevant information. One possible improvement could be the reducing of the ROHC profile number.

Furthermore, the number of Diet−ESP implementations should be extended. One application may be a native C implementation which can be used as the server side implementation of a Diet−ESP connection. Thus, the IP stack of the underlying operating system (e.g. Linux) does not necessarily need to be modified. The C−implementation can be run on the system awaiting Diet−ESP packets from a sensor network and forward them to the application working with the data of the sensor. Another possible investigation could be the implementation of Diet−ESP inside an IP stack of a common operating system, like the Linux Kernel. This implementation should be quite similar to the implementation in Contiki, but since the Linux Kernel provides much more features than Contiki, it should be done by developers familiar with the Kernel, in order to minimize the risks of security flaws due to implementation mistakes.

This thesis describes a basic way to include the Diet−ESP context to the IKEv2 protocol, which may be improved by experts before it is going to be implemented. Thus, the IKEv2 extension should be standardized and implemented for existing implementations, like StrongSwan [75]. With an existing implementation for exchanging the Diet−ESP context, the use of the protocol can be significantly simplified. Additionally, a proper standardization of the exchange can help minimizing the security flaws which may be caused by not security affine application developers, using the Diet−ESP context but do not take care on the security considerations.

# List of Figures

# List of Tables

## References

[1] IEEE Standard for Ethernet - Section 1. *IEEE Std 802.3 2012 (Revision to IEEE Std 802.3 2008)*.

[2] IEEE Standard for Local and Metropolitan Area Networks Port-Based Network Access Control. *IEEE Std 802.1X 2004 (Revision of IEEE Std 802.1X 2001)*, p. 1–169.

[3] 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 (Revision of IEEE Std 802.15.4 2006)*, p. 1–314.

[4] Borman, D., Deering, S., and Hinden, R. *IPv6 Jumbograms.* RFC 2675 (Proposed Standard). aug. 1999. Request for Comments. IETF, 2675. http://www.ietf.org/rfc/rfc2675.txt.

[5] Bormann, C. 2013. *6LoWPAN Generic Compression of Headers and Header-like Payloads. Internet-Draft.* http://tools.ietf.org/html/draft-bormann-6lo-ghc-00. Accessed 13 January 2014.

[6] Bormann, C., Burmeister, C., Degermark, M., Fukushima, H., Hannu, H., Jonsson, L.-E., Hakenberg, R., Koren, T., Le K, Liu, Z., Martensson, A., Miyazaki, A., Svanbro, K., Wiebke, T., Yoshimura, T., and Zheng, H. *RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed.* RFC 3095 (Proposed Standard), Updated by RFCs 3759, 4815. jul. 2001. Request for Comments. IETF, 3095. http://www.ietf.org/rfc/rfc3095.txt.

[7] Bormann, C., Castellani, A. P., and Shelby, Z. 2012. CoAP: An Application Protocol for Billions of Tiny Internet Nodes. *Internet Computing, IEEE* 16, 2, p. 62–67.

[8] ContikiOS. 2013. *Contiki: The Open Source Operating System for the Internet of Things.* http://www.contiki-os.org/. Accessed 27 November 2013.

[9] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and Polk, W. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.* RFC 5280 (Proposed Standard), Updated by RFC 6818. may. 2008. Request for Comments. IETF, 5280. http://www.ietf.org/rfc/rfc5280.txt.

[10] Coronis. 2013. *Coronis Excelyo Data Sheet.* http://www.coronis.com/downloads/Excelyo_Data_Sheet_A4_CS5.pdf. Accessed 22 June 2014.

[11] Deering, S. and Hinden, R. *Internet Protocol, Version 6 (IPv6) Specification.* RFC 2460 (Draft Standard), Updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946, 7045. dec. 1998. Request for Comments. IETF, 2460. http://www.ietf.org/rfc/rfc2460.txt.

[12] die.net. *raw(7): IPv4 raw sockets - Linux man page.* http://linux.die.net/man/7/raw. Accessed 19 June 2014.

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

[14] Eronen, P. *IKEv2 Mobility and Multihoming Protocol (MOBIKE).* RFC 4555 (Proposed Standard). jun. 2006. Request for Comments. IETF, 4555. http://www.ietf.org/rfc/rfc4555.txt.

[15] Ertekin, E., Christou, C., and Bormann, C. *IPsec Extensions to Support Robust Header Compression over IPsec.* RFC 5858 (Proposed Standard). may. 2010. Request for Comments. IETF, 5858. http://www.ietf.org/rfc/rfc5858.txt.

[16] Ertekin, E., Christou, C., Jasani, R., Kivinen, T., and Bormann, C. *IKEv2 Extensions to Support Robust Header Compression over IPsec.* RFC 5857 (Proposed Standard). may. 2010. Request for Comments. IETF, 5857. http://www.ietf.org/rfc/rfc5857.txt.

[17] Ertekin, E., Jasani, R., Christou, C., and Bormann, C. *Integration of Robust Header Compression over IPsec Security Associations.* RFC 5856 (Informational). may. 2010. Request for Comments. IETF, 5856. http://www.ietf.org/rfc/rfc5856.txt.

[18] European Technology Platform for Smart Grids. *Smart Grids.* http://www.smartgrids.eu/. Accessed 3 June 2014.

[19] Fielding, R. and Reschke, J. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing.* RFC 7230 (Proposed Standard). jun. 2014. Request for Comments. IETF, 7230. http://www.ietf.org/rfc/rfc7230.txt.

[20] Frankel, S., Glenn, R., and Kelly, S. *The AES-CBC Cipher Algorithm and Its Use with IPsec.* RFC 3602 (Proposed Standard). sep. 2003. Request for Comments. IETF, 3602. http://www.ietf.org/rfc/rfc3602.txt.

[21] Frankel, S. and Herbert, H. *The AES-XCBC-MAC-96 Algorithm and Its Use With IPsec.* RFC 3566 (Proposed Standard). sep. 2003. Request for Comments. IETF, 3566. http://www.ietf.org/rfc/rfc3566.txt.

[22] Granjal, J., Sa Silva, J., Monteiro, E., and Boavida, F. 2008. Why is IPSec a viable option for wireless sensor networks. In *Mobile Ad Hoc and Sensor Systems, 2008. MASS 2008. 5th IEEE International Conference on*, p. 802–807.

[23] Gueron, S. 2012. *Intel Advanced Encryption Standard (AES) New Instructions Set.* https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set. Accessed 6 June 2014.

[24] Housley, R. *Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP).* RFC 3686 (Proposed Standard). jan. 2004. Request for Comments. IETF, 3686. http://www.ietf.org/rfc/rfc3686.txt.

[25] Housley, R. *Using Advanced Encryption Standard (AES) CCM Mode with IPsec Encapsulating Security Payload (ESP).* RFC 4309 (Proposed Standard). dec. 2005. Request for Comments. IETF, 4309. http://www.ietf.org/rfc/rfc4309.txt.

[26] Hui, J. and Thubert, P. *Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks.* RFC 6282 (Proposed Standard). sep. 2011. Request for Comments. IETF, 6282. http://www.ietf.org/rfc/rfc6282.txt.

[27] IANA. 2014. *RObust Header Compression (ROHC) Profile Identifiers.* https://www.iana.org/assignments/rohc-pro-ids/rohc-pro-ids.txt. Accessed 19 May 2014.

[28] IANA. 2014. *Internet Key Exchange Version 2 (IKEv2) Parameters.* http://www.iana.org/assignments/ikev2-parameters/ikev2-parameters.xhtml#ikev2-parameters-6. Accessed 6 July 2014.

[29] Iso. 2009. *ISO/IEC 27000 Information technology — Security techniques — Information security management systems — Overview and vocabulary.*

[30] Jain, A., Kant, K., and Tripathy, M. R. 2012. Security Solutions for Wireless Sensor Networks. In *Advanced Computing & Communication Technologies (ACCT), 2012 Second International Conference on*, p. 430–433.

[31] Jonsson, L.-E. and Pelletier, G. *RObust Header Compression (ROHC): A Compression Profile for IP.* RFC 3843 (Proposed Standard) Updated by RFC 4815. jun. 2004. Request for Comments. IETF, 3843. http://www.ietf.org/rfc/rfc3843.txt.

[32] Jutvik, V. *Contiki-IPsec*. https://github.com/vjutvik/Contiki-IPsec. Accessed 22 June 2014.

[33] Kaliski, B. *PKCS #5: Password-Based Cryptography Specification Version 2.0.* RFC 2898 (Informational). sep. 2000. Request for Comments. IETF, 2898. http://www.ietf.org/rfc/rfc2898.txt.

[34] Kaufman, C., Hoffman, P., Nir, Y., and Eronen, P. *Internet Key Exchange Protocol Version 2 (IKEv2).* RFC 5996 (Proposed Standard), Updated by RFCs 5998, 6989. sep. 2010. Request for Comments. IETF, 5996. http://www.ietf.org/rfc/rfc5996.txt.

[35] Kent, S. *IP Authentication Header.* RFC 4302 (Proposed Standard). dec. 2005. Request for Comments. IETF, 4302. http://www.ietf.org/rfc/rfc4302.txt.

[36] Kent, S. *IP Encapsulating Security Payload (ESP).* RFC 4303 (Proposed Standard). dec. 2005. Request for Comments. IETF, 4303. http://www.ietf.org/rfc/rfc4303.txt.

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

[38] Kim, S., Pakzad, S., Culler, D., Demmel, J., Fenves, G., Glaser, S., and Turon, M. 2005. *Structural Health Monitoring of the Golden Gate Bridge*. http://www.cs.berkeley.edu/~binetude/ggb/. Accessed 18 July 2014.

[39] Kivinen, T. 2013. *Minimal IKEv2 draft-ietf-lwig-ikev2-minimal-01.txt. Internet-Draft*. http://tools.ietf.org/html/draft-ietf-lwig-ikev2-minimal-01. Accessed 9 December 2013.

[40] Kothmayr, T., Schmitt, C., Hu, W., Brünig, M., and Carle, G. 2013. DTLS based security and two-way authentication for the Internet of Things. *Ad Hoc Networks*.

[41] Larzon, L.-A., Degermark, M., Pink, S., Jonsson, L.-E., and Fairhurst, G. *The Lightweight User Datagram Protocol (UDP-Lite).* RFC 3828 (Proposed Standard), Updated by RFC 6335. jul. 2004. Request for Comments. IETF, 3828. http://www.ietf.org/rfc/rfc3828.txt.

[42] Litzenberger, D. C. 2014. *PyCrypto - The Python Cryptography Toolkit*. https://www.dlitz.net/software/pycrypto/. Accessed 19 June 2014.

[43] LWN.net. 2005. *XFRM: BEET IPsec mode for Linux*. http://lwn.net/Articles/144899/. Accessed 30 June 2014.

[44] Madson, C. and Glenn, R. *The Use of HMAC-MD5-96 within ESP and AH.* RFC 2403 (Proposed Standard). nov. 1998. Request for Comments. IETF, 2403. http://www.ietf.org/rfc/rfc2403.txt.

[45] Madson, C. and Glenn, R. *The Use of HMAC-SHA-1-96 within ESP and AH.* RFC 2404 (Proposed Standard). nov. 1998. Request for Comments. IETF, 2404. http://www.ietf.org/rfc/rfc2404.txt.

[46] Manral, V. *Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH).* RFC 4835 (Proposed Standard). apr. 2007. Request for Comments. IETF, 4835. http://www.ietf.org/rfc/rfc4835.txt.

[47] Melen, J. and Nikander, P. 2008. *A Bound End-to-End Tunnel (BEET) mode for ESP. Internet-Draft*. http://tools.ietf.org/id/draft-nikander-esp-beet-mode-09.txt. Accessed 19 May 2014.

[48] Microsoft. *TCP/IP Raw Sockets (Windows).* http://msdn.microsoft.com/de-de/library/windows/desktop/ms740548(v=vs.85).aspx. Accessed 19 June 2014.

[49] Migault, D., Guggemos, T., Palomares, D., Richard, B., and Laurent, M. 2014. Efficient Security for Homenet, IoT and M2M Communications.

[50] Moskowitz, R., Nikander, P., Jokela, P., and Henderson, T. *Host Identity Protocol.* RFC 5201 (Experimental),Updated by RFC 6253. apr. 2008. Request for Comments. IETF, 5201. http://www.ietf.org/rfc/rfc5201.txt.

[51] Olaf Bergmann. 2013. *tinydtls.* http://tinydtls.sourceforge.net/. Accessed 27 November 2013.

[52] OpenCities. 2014. *PROJECT.* http://opencities.net/content/project. Accessed 6 July 2014.

[53] OWASP. 2014. *OWASP Guide Project - OWASP.* https://www.owasp.org/index.php/OWASP_Guide_Project. Accessed 6 July 2014.

[54] Pavkovic, B., Barthel, D., Theoleyre, F., and Duda, A. 2009. Experimental Analysis and Characterization of a Wireless Sensor Network Environment.

[55] Pelletier, G. *RObust Header Compression (ROHC): Profiles for User Datagram Protocol (UDP) Lite.* RFC 4019 (Proposed Standard) Updated by RFC 4815. apr. 2005. Request for Comments. IETF, 4019. http://www.ietf.org/rfc/rfc4019.txt.

[56] Pelletier, G. and Sandlund, K. *RObust Header Compression Version 2 (ROHCv2): Profiles for RTP, UDP, IP, ESP and UDP-Lite.* RFC 5225 (Proposed Standard). apr. 2008. Request for Comments. IETF, 5225. http://www.ietf.org/rfc/rfc5225.txt.

[57] Pelletier, G., Sandlund, K., Jonsson, L.-E., and West, M. *RObust Header Compression (ROHC): A Profile for TCP/IP (ROHC-TCP).* RFC 6846 (Proposed Standard). jan. 2013. Request for Comments. IETF, 6846. http://www.ietf.org/rfc/rfc6846.txt.

[58] Petr Svenda. Basic comparison of Modes for Authenticated-Encryption. (IAPM, XCBC, OCB, CCM, EAX, CWC, GCM, PCFB, CS).

[59] Piguet, C., Masgonty, J. M., Arm, C., Durand, S., Schneider, T., Rampogna, F., Scarnera, C., Iseli, C., Bardyn, J. P., Pache, R., and Dijkstra, E. 1997. Low-power design of 8-b embedded CoolRisc microcontroller cores. *Solid State Circuits, IEEE Journal of* 32, 7, p. 1067–1078.

[60] Postel, J. *User Datagram Protocol.* RFC 768 (INTERNET STANDARD). aug. 1980. Request for Comments. IETF, 768. http://www.ietf.org/rfc/rfc768.txt.

[61] Postel, J. *Internet Protocol.* RFC 791 (INTERNET STANDARD) Updated by RFCs 1349, 2474, 6864. sep. 1981. Request for Comments. IETF, 791. http://www.ietf.org/rfc/rfc791.txt.

[62] Postel, J. *Transmission Control Protocol.* RFC 793 (INTERNET STANDARD), Updated by RFCs 1122, 3168, 6093, 6528. sep. 1981. Request for Comments. IETF, 793. http://www.ietf.org/rfc/rfc793.txt.

[63] Python Docs. 2014. *7.3. struct — Interpret strings as packed binary data — Python v2.7.7 documentation.* https://docs.python.org/2/library/struct.html. Accessed 20 June 2014.

[64] Rantos, K., Papanikolaou, A., and Manifavas, C. 2013. IPsec over IEEE 802.15.4 for Low Power and Lossy Networks. In *Proceedings of the 11th ACM International Symposium on Mobility Management and Wireless Access*. MobiWac '13. ACM, New York, NY, USA, p. 59–64.

[65] Raza, S., Chung, T., Duquennoy, S., Yazar, D., Voigt, T., and Roedig, U. 2010. Securing internet of things with lightweight ipsec. Technical Report. In *SICS*.

[66] Raza, S., Duquennoy, S., Chung, T., Yazar, D., Voigt, T., and Roedig, U. 2011. Securing communication in 6LoWPAN with compressed IPsec. In *Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on*, p. 1–8.

[67] Raza, S., Duquennoy, S., and Selander, G. 2014. *Compression of IPsec AH and ESP Headers for Constrained Environments. Internet-Draft.* http://tools.ietf.org/id/draft-raza-6lowpan-ipsec-01.txt. Accessed 15 May 2014.

[68] Raza, S., Shafagh, H., and Dupont, O. 2014. *Compression of Record and Handshake Headers for Constrained Environments. Internet-Draft.* http://tools.ietf.org/id/draft-raza-dice-compressed-dtls-00.txt. Accessed 16 May 2014.

[69] Raza, S., Shafagh, H., Hewage, K., Hummen, R., and Voigt, T. 2013. Lithe: Lightweight Secure CoAP for the Internet of Things. *Sensors Journal, IEEE* 13, 10, p. 3711–3720.

[70] Raza, S., Trabalza, D., and Voigt, T. 2012. 6LoWPAN Compressed DTLS for CoAP. In *Distributed Computing in Sensor Systems (DCOSS), 2012 IEEE 8th International Conference on*, p. 287–289.

[71] Rescorla, E. and Modadugu, N. *Datagram Transport Layer Security Version 1.2.* RFC 6347 (Proposed Standard). jan. 2012. Request for Comments. IETF, 6347. http://www.ietf.org/rfc/rfc6347.txt.

[72] Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V. *RTP: A Transport Protocol for Real-Time Applications.* RFC 3550 (INTERNET STANDARD), Updated by RFCs 5506, 5761, 6051, 6222, 7022, 7160, 7164. jul. 2003. Request for Comments. IETF, 3550. http://www.ietf.org/rfc/rfc3550.txt.

[73] Shelby, Z., Hartke, K., and Bormann, C. *Constrained Application Protocol (CoAP).* june. 2013. Internet-Draft, draft-ietf-core-coap-18. http://www.ietf.org/id/draft-ietf-core-coap-18.txt. Accessed 27 November 2013.

[74] Song, J. H., Poovendran, R., and Lee, J. *The AES-CMAC-96 Algorithm and Its Use with IPsec.* RFC 4494 (Proposed Standard). jun. 2006. Request for Comments. IETF, 4494. http://www.ietf.org/rfc/rfc4494.txt.

[75] strongSwan. *strongSwan - IPsec for Linux.* http://www.strongswan.org/. Accessed 17 July 2014.

[76] Texas Instruments. *CC2420 | Proprietary 2.4 GHz | Wireless Connectivity | Description & parametrics.* http://www.ti.com/lit/ds/symlink/cc2420.pdf. Accessed 22 June 2014.

[77] Texas Instruments, Incorporated [SLAS256,D ]. *MSP430F11x Mixed Signal Microcontrollers (Rev. D).* http://www.ti.com/lit/ds/slas256d/slas256d.pdf. Accessed 3 June 2014.

[78] Texas Instruments, Incorporated [SWRS046,F ]. *Single-Chip Low Power RF Transceiver for Narrowband Systems (Rev. F).* http://www.ti.com/lit/ds/symlink/cc1020.pdf. Accessed 3 June 2014.

[79] Verschuren, J., Govaerts, R., and Vandewalle, J. 1993. ISO-OSI security architecture. In *Computer Security and Industrial Cryptography*, G. Goos, J. Hartmanis, B. Preneel, R. Govaerts and J. Vandewalle, Eds. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, p. 179–192.

[80] Viega, J. and McGrew, D. *The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP).* RFC 4106 (Proposed Standard). jun. 2005. Request for Comments. IETF, 4106. http://www.ietf.org/rfc/rfc4106.txt.

[81] Watro, R., Kong, D., Cuti, S.-f., Gardiner, C., Lynn, C., and Kruus, P. TinyPK. securing sensor networks with public key technology. In *the 2nd ACM workshop*, p. 59.

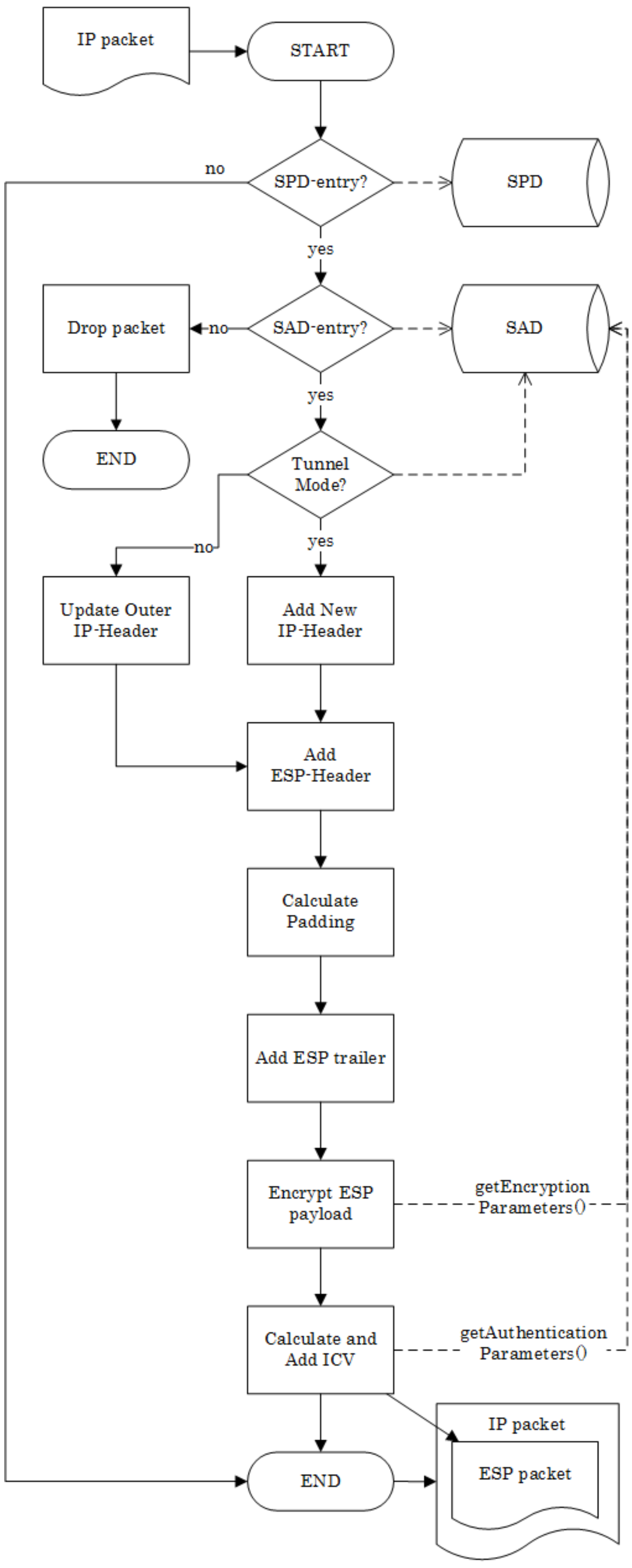
[82] Whiting, D., Housley, R., and Ferguson, N. *Counter with CBC-MAC (CCM).* RFC 3610 (Informational). sep. 2003. Request for Comments. IETF, 3610. http://www.ietf.org/rfc/rfc3610.txt.

## Content of attached CD

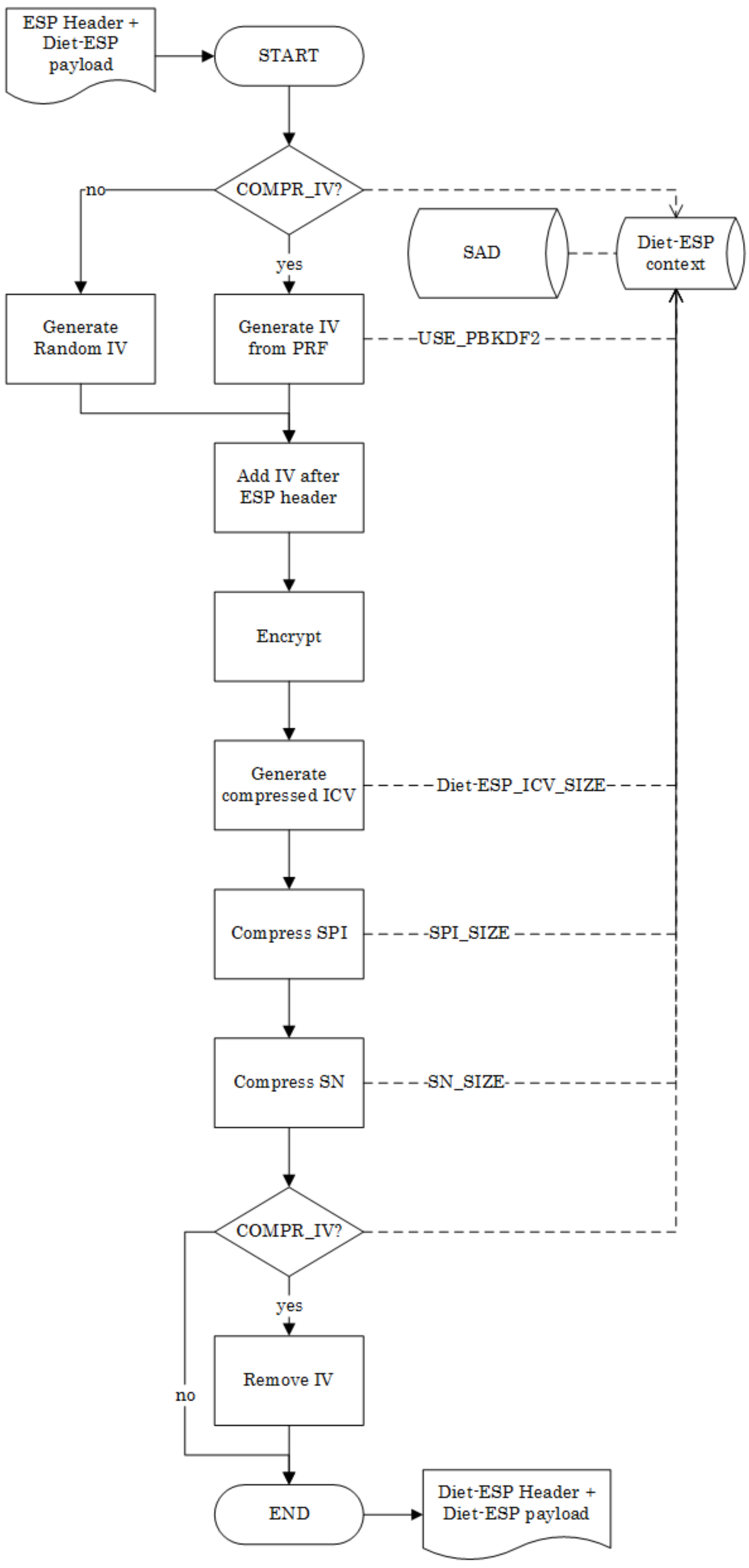
- Dokumentation/PDF/gugg14.pdf
  Digital version of the thesis as PDF file.
- Dokumentation/word/
  Word sources of this thesis
- Dokumentation/Abstract.txt
  Abstract of this thesis as txt file.
- Literatur/
  Used references in PDF files if available.
- Sources/Drawings/
  Source files for drawings used in this thesis
- Sources/memory_print
  Memory print of compilation for Excelyo platform
- Sources/python
  Sources for the Diet–ESP python implementation
- Sources/Contiki-IPsec–contiki–ipsec
  Sources for the Diet–ESP Contiki add–on for the Cooja simulator
- Sources/wavenis–IPstack
  Sources for the Diet–ESP Contiki add–on for the compilation in Excelyo Platform

# Appendix A Flowchart Diagrams

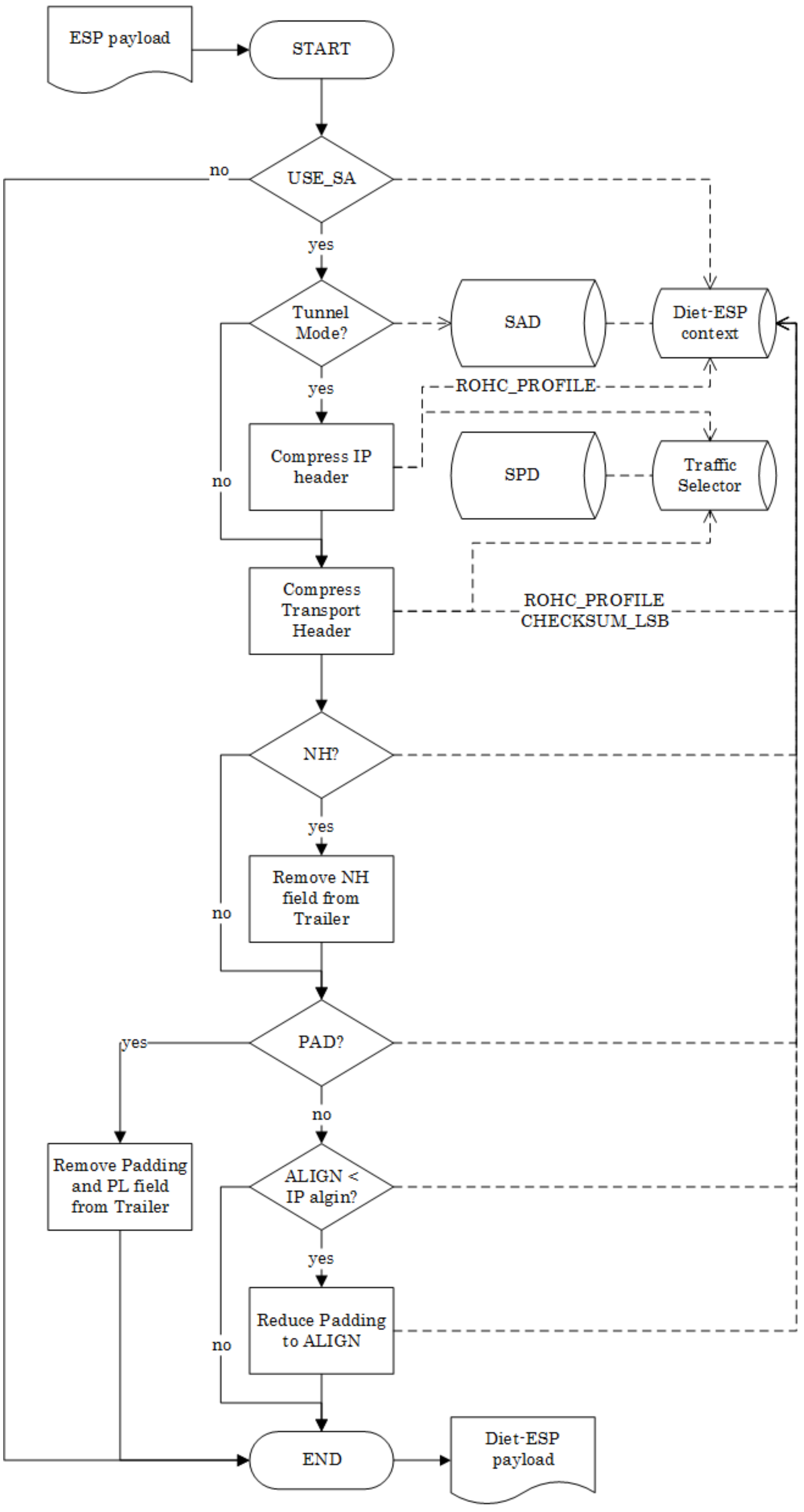
## A.1 Flowchart Diagram: Outgoing Diet–ESP packet procession



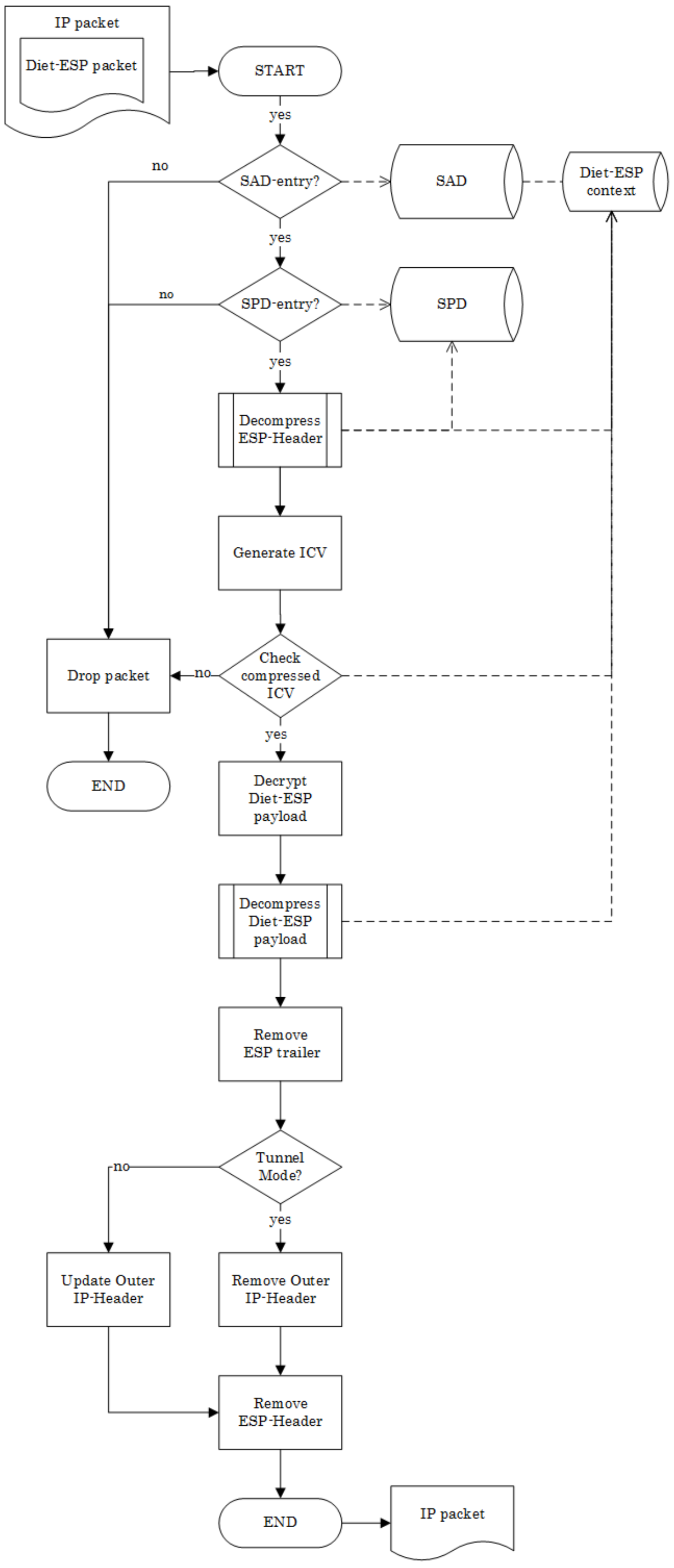
## A.2 Flowchart Diagram: Compress ESP header



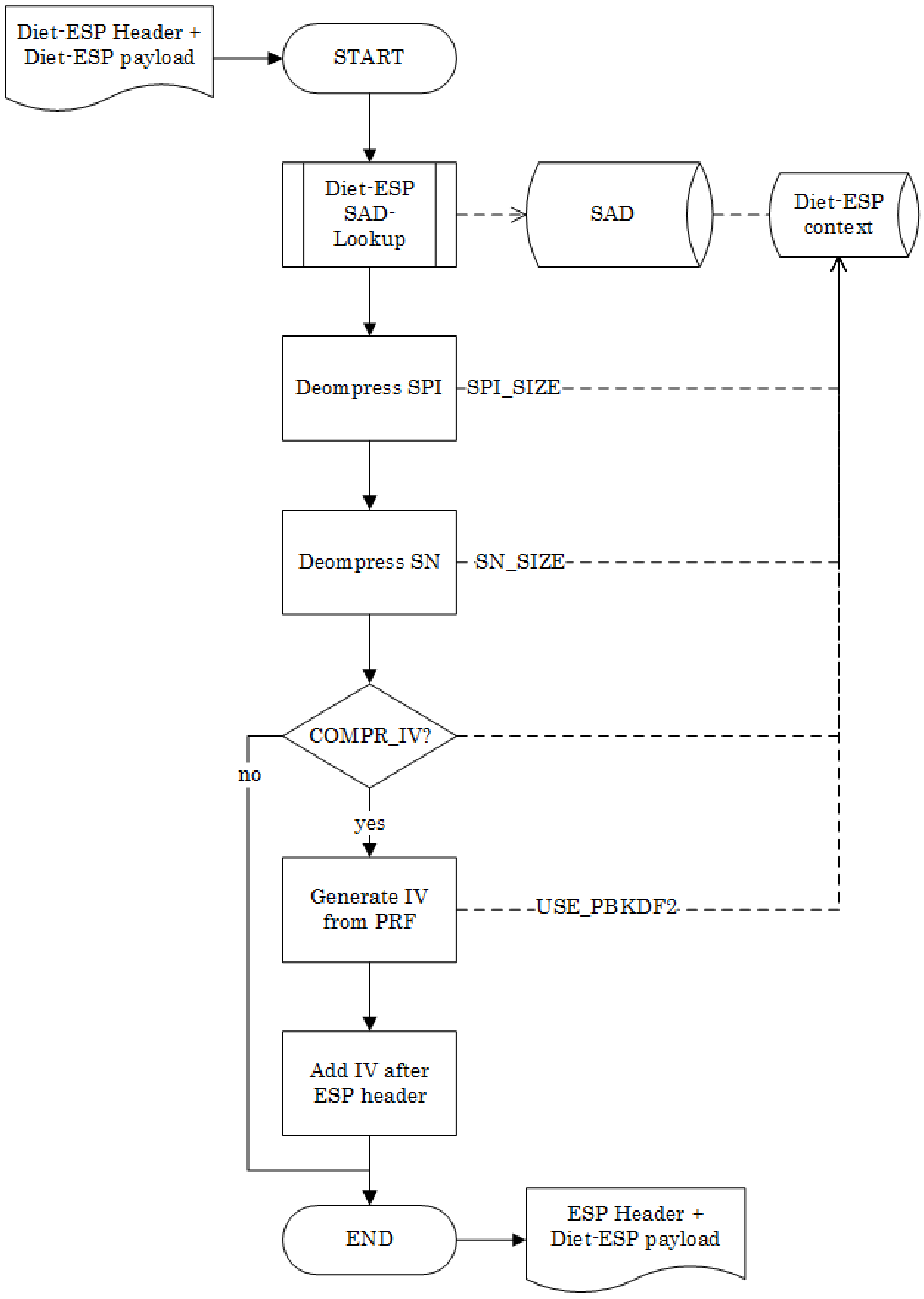
## A.3 Flowchart Diagram: Compress ESP payload



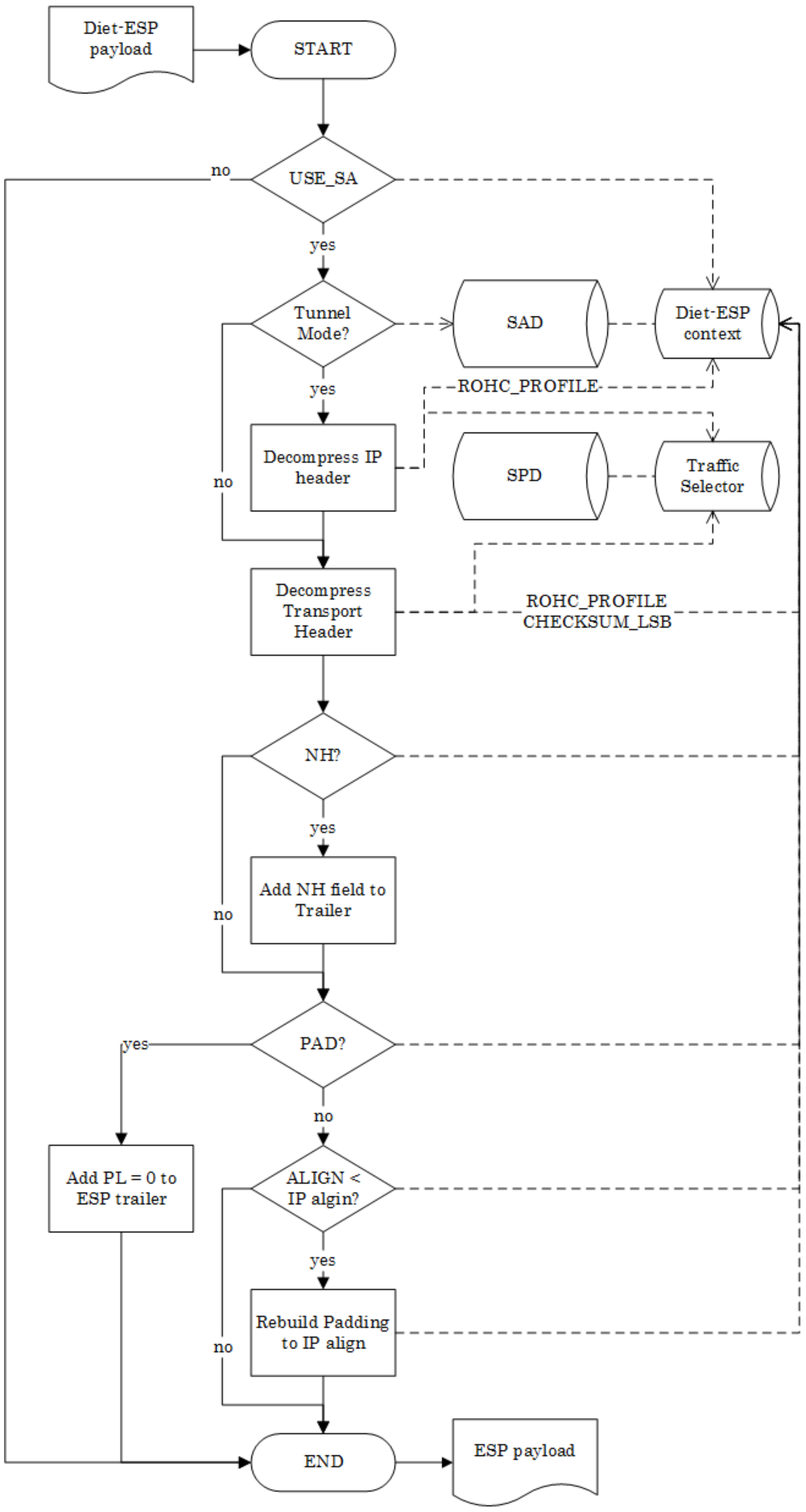
## A.4 Flowchart Diagram: Incoming Diet−ESP packet procession



## A.5 Flowchart Diagram: Decompress Diet–ESP header



## A.6 Flowchart Diagram: Decompress Diet−ESP payload


Diet-ESP payload
START
USE_SA
no
yes
Tunnel Mode?
SAD
Diet-ESP context
yes
ROHC_PROFILE
Decompress IP header
no
SPD
Traffic Selector
Decompress Transport Header
ROHC_PROFILE
CHECKSUM_LSB
NH?
yes
Add NH field to Trailer
no
PAD?
yes
no
Add PL = 0 to ESP trailer
ALIGN < IP algin?
yes
Rebuild Padding to IP align
no
END
ESP payload