IPsec and IKEv2 for the Contiki OS

Vilhelm Jutvik [1]

February 26, 2014

[1]To everyone at SICS and the NES group

Abstract

In this the work I explore...

Contents

Chapter 1

Populärvetenskaplig sammanfattning

nödvändig bakgrundsinformation för en allmänbildad läsare som inte nödvändigtvis är specialist inom området;

hot and cold mediums

Chapter 2

Introduction

The Internet has revolutionized communication. I would argue that this came about because it drastically lowered the cost of communication by removing interface barriers, unifying standards and organizations. Internet is cheap while legacy, purpose-made, communication networks are expensive (e.g. consider the PSTN[1]). These lowered costs allowed people to send and receive information at an enormous rate, creating new types of medias and services such as the blog, the social network, the search engine etc. Because of these benefits, we try to expand the Internet, and now strides are being made to connect the objects that we surround ourselves with, creating what's commonly called he 'Internet of Things' (abbreviated as IoT). It's a vision of an extended Internet where household objects (radiators, lighting etc), sensors and actuators in industrial machinery, cars etc are put 'online'. Data is primarily sent and received by radio as cables are expensive to install and maintain.

The underlying motivation as to why we want to communicate with things are that they and their surroundings matters to humans. A thing such as freezer can notify building management when its about to break down and notify building management before the food is spoiled. Data from the vibration sensor in the bridge helps the engineer to model its health and plan maintenance. A thing can also help another thing, e.g. the temperature sensor telling the radiator to turn as the temperature is falling.

Unlike a regular host on the Internet (e.g. a PC) an IoT device is often purpose-made for a few single tasks (e.g. measuring the temperature in the example above). Data is usually sent and received over radio. This allows the computer's electronics to be small, cheap and consume little power, so little that it can be run of a non-rechargeable battery for the system's entire lifetime. This set of technological characteristics are what commonly characterize the IoT.

Contiki is small and resource efficient operating system that tries to ad-

[1] Public Switched Telephone Network; the ordinary telephone network

dress the above requirements. With only 35 kB of ROM and 8 kB RAM it provides an IP stack as well as multitasking. On the multitude of platforms officially supported, the hardware is often composed of a small 16-bit CPU, a small non-rechargeable battery and a radio operating in the 2.4 GHz ISM band using the IEEE 802.15.4 standard which specifies the physical as well as the media access control layer. Developed mainly at SICS, the OS have gathered a large following around the world, only surpassed by its relative TinyOS. Naturally, Contiki is the primary research platform for IoT at SICS, and is the reason as to why the work of this thesis is based upon it.

As the IoT will control and monitor sensors as well as machines in our surroundings, security is a natural concern. Internet (more specifically, the IP protocol) by itself does not offer any security guarantees by default. This is especially true in the IoT world where the physical layer often is provided by radio. Therefore, this thesis will explore the possibilities of implementing the IPsec security extension of the IP-protocol in the Contiki OS. This will bring methods to secure communications to the IoT, that are also compatible with vast parts of the Internet.

2.1 Problem statement

As the IoT will control and monitor sensors as well as machines in our surroundings, security is a natural concern. Internet (more specifically, the IP protocol[2] by itself does not offer any security guarantees by default. This is especially true in the IoT world where the physical layer often is provided by radio. Any device with suitable radio equipment can read IP-packets destined for others, create and send packets while imposing as another host and manipulate data in transmissions destined for somebody else. This is problematic if the IoT device measures or controls something deemed important to humans, such as a door lock or a motion sensor in a burglar alarm. When receiving a message in the case of the former, the host controlling the door lock wants to assert that: the identity of sender is correct; that the message have not been tampered with; and preferably, it should be encrypted so that no other host except the addressee can read it. At the same time, the technology enabling this must be supported by the lock (the IoT device) and the other host, which might be any type of machine anywhere on the Internet.

In this thesis, I have elected to investigate IPsec, which is one of many similar standards designed to solve these problems. I will implement and evaluate a subset of IPsec and the associated IKEv2 standard for cryptographic key exchange, with particular emphasis on the latter.

The difficulty of the work lies in making an implementation that is compatible with other hosts on the Internet (enabling communication), fulfilling the security requirements (correctness) while simultaneous assuring that the

[2] I would like to remind the reader that IP is an abbreviation of **I**nternet **P**rotocol.

code is small enough to fit in the limited memory. This also relates to energy requirements, especially in the context of cryptography which is heavily used by IPsec, as computations consumes a sizable part of an IoT host's energy budget.

2.2 Research question

Can IPsec and IKEv2 be implemented within the current hardware boundaries while still being interoperable with other Internet hosts?

2.3 Method

Much of the research in IoT is carried out by experiments because of the highly applied nature of the field. Indeed, most of the Internet as-of-today, was constructed on the basis of principles surmised from experiments. Therefore, I decided to implement the various components that makes up a functional IPsec system and then test them. The tests were carried out by subjecting the system to common communication scenarios with other IPsec-enabled Internet hosts: handshaking a new secure connection; receiving and transmitting packets with various security policies applied; housekeeping of connections. The success or failure of the test is determined if it worked, or if it succeeded, how well it did so. Finally, memory consumption and cpu time is measured for the different tasks. It's the results of these tests that will form the basis of the evaluation of the 1) quality and workmanship of the implementation 2) suitability of IPsec for IoT in general.

2.3.1 Development process

The source code was managed using the Git source code revision system. Development was organized in such a way that the project was run in one branch[3] (hereafter called *develop*) parallel to Contiki's master branch. Important patches / changes that occurred in Contiki while the development was underway could thus simply be fetched / merged into *develop*. Patches in develop that proved was then merged in from the

2.4 Limitations

Firstly, the purpose of this work is to answer the questions raised in the problem statement, not to create an implementation that's ready for production use. Thus, parts of the standard that are not necessary to meet this end can be omitted. This also includes parts that are labeled as REQUIRED

[3]Git branches are similar in concept to those in other SCMS. ...

in the standard documents, as they're only necessary if you strive towards full compliance, which in terms of features arguably is superset of that of interoperability.

2.5 Alternative Approaches

There are many ways of securing communication in a computer network, far too many to review in this section. What follows is a swift review of the closest alternatives for security in IoT networks. A thorough review and comparison will be made of these in the discussion towards the end of this thesis.

The traditional approach in the IoT world has been to secure communication by using the IEEE's 802.11.4 link layer and its security features. There are also plenty of research articles outlining completely new security schemes, but none of these have made it into and IETF standard as far as I am aware. Another alternative is the IETF's TLS protocol[4] that operates next to the application layer, but no implementation of that has been completed as of today to the best of my knowledge.

A final possibility would be to come to a conclusion by using analysis. Supposing that the main obstacle to a functional solution is that of ROM, RAM, CPU speed and energy requirements. As one can assume the cryptographic libraries to consume the majority of the resources, one can arrive at an approximation for the hardware requirements by benchmarking them stand-alone. This would certainly help in arriving at an upper bound for the hardware requirements, but the analysis would not tell us anything about the operational problems and benefits that a complete implementation would.

2.6 Scientific Contributions

This report makes three scientific contributions. The first is the demonstration of that it's feasible to implement IPsec with dynamic key management and IKEv2 in the Contiki OS. The second is that it's possible to simplify the IPsec protocol in IPv6-IoT environments without sacrificing much in interoperability. The final is that IPsec's network-centric policy language is ill suited for ad-hoc, self-organized network environments, such as that created by the RPL[5] routing protocol commonly used in conjunction with IoT.

[4]The TLS protocol is a successor to SSL

[5]The Routing Protocol for Low-Power and Lossy Networks (RPL) is an IETF standard (RFC 6553) that creates dynamic routing topologies

2.7 Report Structure

The purpose of the chapter 'Introduction' is to give the reader an overview of the report. The next chapter, Background, will begin by a discussion of the problems that are particular to the field of IoT and how Contiki is constructed. This will be followed by a brief recount of the background and design of IPsec, followed by an introduction to automatic keying and the IKEv2 protocol. In chapter 'Design and Implementation' I outline how the IPsec standard is adapted to IoT and Contiki. Simplifications and omissions of features are explained and argued for. The design is then evaluated in Evaluation by measuring parameters such as time, energy and memory usage when sending and receiving data. Space will also be given to observations of a qualitative nature, such as how well the standard integrates with the Contiki OS. My opinions of the evaluation and suggestions for future work is given in Future Work and Recommendations.

Chapter 3

Background

The purpose of this chapter is to introduce the reader to the Contiki OS and the IPsec standard. A good understanding of these systems are necessary to understand the later chapter 'Design and Implementation'.

3.1 Contiki

Contiki is an operating system designed for computers with severely constrained resources[DGV04]. At the time of its original release in 2004 it was targeted at 8-bit micro controllers, ROM on the order of 100 kB and RAM less than 20 kB[DGV04, 1]. Although the production costs of electronics certainly will continue to fall in future, this doesn't necessarily translate into a lesser interest in resource constrained operating systems such as Contiki - the lower prices can be used to produce more units instead of making existing devices more powerful. The choice of small devices such as the traditional 8- and 16-bit micro-controllers that has been the mainstay of Contiki since its inception[1], is not just about cost, but also about power efficiency. Small chips have been, for various reasons, more energy efficient on a per cycle basis than heavier systems (e.g. the 32-bit Cortex-M3 platform from the ARM Corporation). Having said that, this is set to change as the more powerful micro-controllers are becoming increasingly energy-efficient, a fact which we will address in the section 6.4 'Future Work and Recommendations'.

Having stated this, Contiki's greatest benefit is not that it's very compact, it's that it provides 'a rich enough execution environment while staying within the limitations of the constrained devices'[DGV04, Introduction]. The 'rich execution environment' is a reference to Contiki's capability to dynamically load and unload software, its multithreading, mesh networking[TED10] and more, that are all designed to fit within the constraints, memory- as well as energy-wise (i.e. limit the number of execution cycles and keep periph-

[1]Examples of micro-controllers commonly used with Contiki are Texas Instrument's MSP430 architecture and Atmel's AVR 8-bit dito.

erals such as the radio asleep). We will now review the most important of these features that makes Contiki a popular OS for IoT-devices.

3.1.1 Multitasking

Multitasking (concurrency) is provided in Contiki by the *Protothreads* mechanism[DS05]. Before explaining its inner workings, I will provide the reader with a short review of the problems associated with multitasking in resource constrained systems.

The motivation underlying protothreads is to provide the programmer with multitasking that is easy to use, easy to understand and low in resource consumption. Multitasking has traditionally been implemented in virtually all operating systems by the means of *processes*. In such a model, each *process* is associated with an executable, has its own private stack and memory, set of registers (etc) and can be interrupted (more specifically, *preempted*), at any time by the operating system's *scheduler* in order to run a kernel specific task or to *schedule* another task. In all cases where a process is preempted, a *context switch* has to take place where much of the CPU's registers are changed, caches flushed etc. This is unsuitable for constrained environments due to the massive memory overhead of a process structure, private stacks and the energy required for doing a context switch.

One approach to overcome these drawbacks while still allowing some form of multitasking was employed by another sensor network OS: TinyOS[LMP+05]. TinyOS allowed the programmer to achieve multitasking by implementing their programs as event driven state machines. The machines has no stack and can only keep state (i.e. data) that are implicitly declared as such. As the C programming language doesn't have any event primitives, an extension to C was developed and named *nesC*[GLVB+03].

Contiki took a different route and provided the programmer with a model that resembled the sequential programming style (which is believed to be more comfortable for the majority of developers)[2], but still benefitted from the efficiency of the event driven state machine.

In figure 3.1 on the facing page a code example using Protothreads is shown, in which a radio is awaken at a set schedule to wait for check for transmissions. Protothreads uses a concept called *conditional blocking* which can be explained in the following way: the execution can't leave the thread until a conditional blocking statement is reached (macros prefixed with 'PT'). This can also be referred to as *cooperative multitasking* as control can only leave a program when it voluntarily relinquish it.

As all Protothreads share the same stack and Contiki only needs to remember at which conditional to resume the thread, memory requirements

[2] I found this reasoning to have merit after having participated in a course where application programming for TinyOS as well as Contiki was taught, having observed that the vast majority of his classmates preferred Contiki's Protothreads to TinyOS' nesC.

```
PT_THREAD(radio_wake_thread(struct pt *pt)) {
  PT_BEGIN(pt);

  while(1) {
    radio_on();
    timer_set(&timer, T_AWAKE);
    PT_WAIT_UNTIL(pt, timer_expired(&timer));

    timer_set(&timer, T_SLEEP);
    if(!communication_complete()) {
      PT_WAIT_UNTIL(pt, communication_complete()
                        || timer_expired(&timer));
    }

    if(!timer_expired(&timer)) {
      radio_off();
      PT_WAIT_UNTIL(pt, timer_expired(&timer));
    }
  }

  PT_END(pt);
}
```

Figure 3.1: Example code for a radio transmission program utilizing Protothreads. From [DS05, p.7]. FIX: Check copyright!

can be as small as 2 bytes of RAM per thread. Furthermore, curiously enough, the 'context-switching' mechanism can be wholly implemented using only the C switch control structure. This 'quirk' of the C language was probably first discovered by Tom Duff[DS05, p.10][Wik13a].

3.1.2 IP networking

UDP, TCP as well as ICMP over IPv4 and IPv6 are provided by the μIP stack, described in[Dun03]. The system is surprisingly robust given that its footprint amounts to only 10 kB of ROM and a couple of hundred bytes of RAM: it fulfills all necessary host-to-host requirements stipulated in the relevant RFC documents[Dun03, p.4] (e.g. flow and congestion control). The number of TCP connections and the size of the transmission / reception buffers are only limited by available RAM.

One important reason as to why μIP can achieve such a low memory footprint is that is liberates the OS from the complicated transmission of data between application and IP stack. Most of the requirements associated with this interface, as described in the RFC, are waived in [Dun03, p.4]. This not only removes the need for a lot of software logic, but also allows a drastic simplification of transmission buffer management, something which its designer have used to the fullest extent: there is only enough buffer space for one packet at any time, including the need of transmissions as well

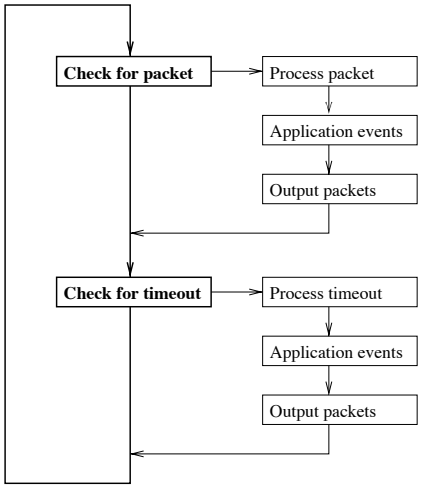

Figure 3.2: μIP's main control loop. From [Dun03, p.6]. FIX: Check copyright!

Table 1: TCP/IP features implemented by uIP and lwIP

Feature	uIP	lwIP
IP and TCP checksums	x	x
IP fragment reassembly	x	x
IP options		
Multiple interfaces		x
UDP		x
Multiple TCP connections	x	x
TCP options	x	x
Variable TCP MSS	x	x
RTT estimation	x	x
TCP flow control	x	x
Sliding TCP window		x
TCP congestion control	Not needed	x
Out-of-sequence TCP data		x
TCP urgent data	x	x
Data buffered for rexmit		x

Figure 3.3: Features of μIP and the now no longer maintained lwIP. Since the table was published in [Dun03, p.4] UDP has been added as well to μIP.

as receptions. Figure 3.2 describes the main loop of μIP which explains its buffer management. The application that wish to transmit data copies it into μIP's buffer and then calls the transmission function with the appropriate addressing information. Upon reception, the procedure is the opposite. The stack figures out which application is interested in the arrived piece of data, then calls its handler, which in turn is programmed to react on the data and to produce a reply, which in turn is written to the very same buffer and so on.

Even though an IP packet can be well over a kilobyte in size (the MTU[3] of IPv6 is 1280 bytes), the buffer can still be much smaller than this if used in conjunction with a MAC[4] layer that has a small MTU (6lowPAN (using IEEE 802.15.4 in turn) is very popular for Contiki and has a MAC MTU of at most 102 bytes[MKHC07, section 4]). This will allow the incoming IP-packet fragments to be spoon-fed to the receiving application, negating the need for a large buffer for transmission purposes. Buffer size then becomes a problem of how large IP-packets one wants the applications to transmit.

3.2 Security of the Internet Protocol

Security on the Internet was, and is, as of today an unresolved issue in my opinion. As its protocol, IP, was originally designed and developed in the

[3]Maximum Transmission Unit: The largest size of a packet that a network or a host can handle.

[4]Media Access Control

open world of academic computer science, no security considerations were made. Instead, the three properties of information security (confidentiality, authenticity, integrity) [VD10, section 8.1] were provided by physically protecting the network's hardware and the imposition of social norms among its users and operators. As the Internet grew, this and other weaknesses started to become problems. In 1995, the IETF[5] released a number of documents starting with RFC[6] 1883[DH95] that defined IPv6, the planned successor of IPv4. Since security had become a concern at the time of its design, provisions for secure communication was included, and this part of the IPv6 standard was named IPsec[7]. This provided the network layer (the layer which IP operates on) and everything above it with secure communication.

However, adaption of IPv6 has been slow despite growing technical problems with the still incumbent IPv4. The resistance to upgrade is widely attributed to the fact that the cost and risk (software adaption, operator training, equipment replacement etc) of making the change in most networks outweighs the potential benefits. This equation will certainly change, if slowly, in the coming years. In the meantime, IPv6 has been found to be a great boon to new networks like Internet-connected smart phones, Internet-enabled set top TV-boxes and, naturally, the Internet of Things. In these cases it can be argued that the cost of adopting the legacy IPv4 is roughly the same as that of adopting IPv6, while enjoying benefits of the latter's simplicity, auto configuration features and more[8].

3.3 IPsec

IPsec is a suite of protocols that secures IP packets in an end-to-end fashion. Flows of packets, distinguished by properties such as source and target address, transport layer protocol, etc can be defined and different security policies applied. The system is not only limited to *host-to-host* packet flows, but also allows a host to act as a gateway (termed *security gateway* in the IPsec standard), enabling a secure link between two networks (*network-to-network*) or a network to a host (*network-to-host*). The suite can be implemented either directly in the IP stack of a host, but may also be located between the link layer and the network layer[KS05, section 3.3], facilitating retrofits of existing stacks.

The motivation of the IPsec system is to deliver the security services

[5]The Internet Engineering Task Force is the governing body of the development of Internet standards.

[6]RFC, or Request For Comment, is the collective name of the IETF's standards

[7]I should be noted that with RFC 6434[JLN11] in late 2011, the requirement that all IPv6-compliant hosts must feature IPsec functionality was dropped.

[8]See chapter 15.1 in [VD10] for a brief discussion about the benefits of IPv6 over IPv4 in the context of IoT.

confidentiality, authenticity and integrity in the OSI[9] stack's network layer, thus securing all data at this level (namely IP packets) and everything that it encapsulates (which is all protocols on higher levels in the stack e.g. TCP, UDP, application protocols). The system is designed in such a way that each compliant host features a set of databases which stores security information such as encryption keys and mechanisms that performs cryptographic operations on IP packets using that information. The hosts uses two extension headers (AH and EPS, as discussed in Understanding the IP headers) to communicate security metadata between the hosts. More complex tasks like key negotiation is carried out by a so called key management protocol, one of which is IKEv2 (described in section 3.6 'The ESP protocol in IPv6'). Now, in this section I will introduce IPsec by describing the components that process the IP packets using information stored in the databases. Then, I will continue in section 3.4 'Understanding the IP headers' by describing the aforementioned extension headers which are used to pass the security meta data between the IPsec hosts.

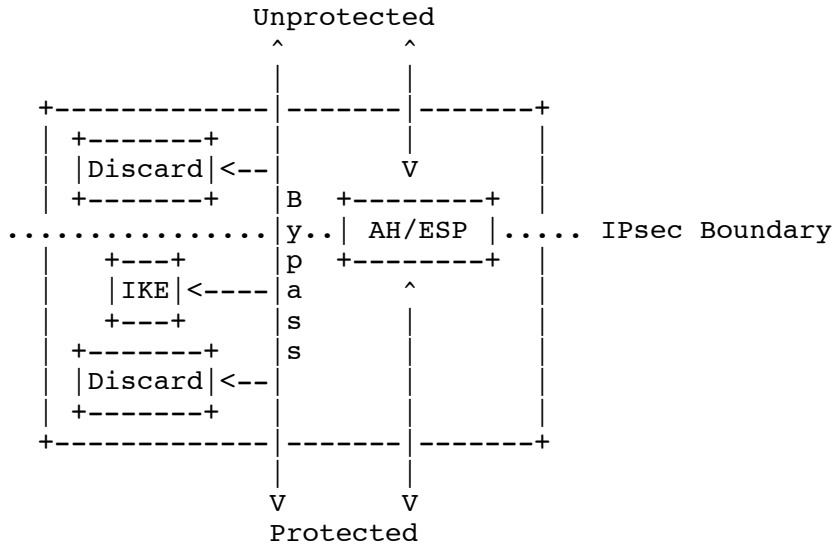

Figure 3.4: IPsec processing of IP-packets. From [KS05, p.8].

In the IPsec processing model IP-packets are said to be passed from *protected* to *unprotected* network interfaces and vice versa. In the usual case, such as above, there are only one of each. The unprotected interface is always the interface connected to the insecure network, while the protected may be either connected to a secure network (the *security gate-*

[9]Open Systems Interconnection is the so called ideal network model consisting of a seven layer protocol 'stack'.

way configuration) or be attached to an operating system's IP stack. The flow of packets between the protected and unprotected interfaces are determined by policies set by the system administrator which are expressed in terms of network- and transport-layer primitives, i.e. IP addresses, port numbers and transport-layer protocols. The policy can cause a packet to be: thrown away (*DISCARD*); forwarded over the IPsec boundary without any action taken (*BYPASS*; removing or applying cryptographic protection of the packet (*PROTECT*). The latter is performed in the module named *AH/ESP* in figure 'IPsec processing of IP-packets. From [KS05, p.8]' on page 16 which handles the decoding and encoding of the AH and ESP IPv6 extension headers (as discussed in length in section 3.4 'Understanding the IP headers').

IPsec affords protection for the traffic passed between networks as well as that between individual hosts. In the former case, two IP networks are linked to each other by an IP-tunnel[10] with an IPsec host at either end, acting as *Security Gateways*. Packets that are routed to the other host's network are encapsulated in a security container (so called AH and ESP headers, which we will explain in section 3.5 'The AH protocol in IPv6' and section 3.6 'The ESP protocol in IPv6') and dispatched to the Security Gateway at the other end of the tunnel. This allows secure communication between two networks and is useful when their hosts are not configured for IPsec. This mode of operation is referred to as *tunnel mode* in the standards documents.

Although tunnel mode is by far the most popular usage of IPsec, this thesis will not implement it as it becomes less relevant in an IoT environment, for a variety of reasons described in chapter 'Design and Implementation'. Instead, the research will focus on the so called transport mode[11] where no encapsulation takes place.

A central concept of the IPsec model is the data structure named the *Security Association* (SA). A Security Association defines a single simplex IP link and the security services it provides to the traffic that it carries. The services can be implemented by one of the two IPsec protocols: *Authentication Header* (AH) and *Encapsulating Security Payload* ESP. Bi-directional (duplex) traffic is enabled by creating a pair of SAs, one for each direction. As I will show, the SA is used by almost all IPsec components and we will return to it from time to time.

[10] An IP-tunnel allows communication between two IP-networks that don't have a routing path between each other. As the packets are transported by means of encapsulation (one IP-packets is transported in the payload field of another), other protocols can be 'tunneled' as well.

[11] Transport mode as in transport layer mode

The policies and cryptographic secrets used in IPsec processing are stored in three databases: the Security Association Database (*SAD*); the Security Policy Database (*SPD*) and its caches; the Peer Authorization Database (*PAD*). The standard stresses the fact that the implementation of these databases should not be viewed as pre-requisites for a compliant IPsec implementation, but that such a system should exhibit the very same behavior as that of a system whose architecture was implemented in line with the standard's model. In the following paragraphs these databases will be used as constructs to explain the inner workings of IPsec, although, as the reader will learn in the chapter 'Design and Implementation', I have actually used these constructs in the final implementation as well. Finally, the system will be explained only in brief as the IPsec suite is simply too large for a thorough review in this thesis. The interested reader is advised to read the complete standard document[KS05] starting at section 4.4 for an in-depth definition of the SAD, SPD and PAD.

The SPD stores the policies that controls IPsec processing. It consists of an ordered list, where each entry is composed of a set of *selectors*, *PFP flags* and a *processing action*.

The *selector* is a data structure that represents a traffic pattern between hosts or networks and serves to distinguish what traffic the policy in question is referring to. The fields are as follows: Local address; Remote address; Next Layer Protocol (e.g. TCP, UDP, ICMP); Local Port (This is dependent upon protocol type, e.g. in the case of an ICMP packet message type and code is listed instead.); Remote Port. Full specifications are found in [KS05, Section 4.4.1.1].

PFP, or *Populate From Packet* flags, are rules used in conjunction with the instantiation of an SA on the basis of the SPD entry. Instantiation in this context means that the newly created SA's selector (which has a similar purpose to a SPD entry's selector) is completed with information from that of the SPD entry's selector. However, for each of the selector's fields there is a corresponding PFP flag. If that flag is set, that field will be populated with information from the packet that triggered the instantiation. This process will be explained in detail further on as it's a central mechanism.

The *processing actions* are the very same as described in 3.3: *DISCARD*; *BYPASS*; *PROTECT*. The action *PROTECT* differs from the the other actions in the sense that is also includes settings of how to protect the traffic, e.g. using a security gateway (tunneling), DSCP[12] services, various SA settings used in instantiation.

The SPD's caches can be divided into three different sections: SPD-O outbound traffic that is not to be protected; SPD-S for outbound traffic that

[12]Differentiated Services Code Point; QoS for IPv6

is to be protected; SPD-I for all incoming traffic. The entries contains the same fields as the SPD, with the exception of the PFP ('Populate From Packet', as described above) field which is not present. From a functional point of view, each entry is an *instantiation* of an SPD entry. That implies that each entry's selector is a subset of one of the SPD's entry's selectors. The IPsec processing mechanism creates such entries in response to encountering traffic that uses one of the SPD's entries. This is made because of two reasons:

> 1) Lookups in the SPD can be slow on large systems with many SPD entries. The SPD cache is faster[13].
>
> 2) a rule in the SPD can order traffic for a large address space to be protected. That address-space can contain many hosts, and each host can in turn demand different SAs for different traffic (e.g. different SAs for traffic over TCP and UDP). Thus, one realizes that each SA must explicitly define what traffic it carries. This is one of the purposes of the SPD-S cache. Each entry corresponds to an SA and its selector defines its traffic.

It's unfortunately hard to get a good overview of the SPD's caching system as the information is sprinkled over large parts of RFC 4301. I recommend the curious reader to begin reading at section 5, 'IP traffic processing'[KS05, Section 5].

The SAD stores the IPsec system's *Security Associations* (SAs). The purpose of the database is to store the parameters of the system's SAs, but can be used to store other information as well. RFC 4301 outlines in section 4.4.2.1 fifteen required fields and because of space constraints, this section will only list the ones that I deem most important.

Security Parameter Index (SPI): Each SA is identified by a 32-bit value. This value is selected by the receiving end of an SA to be unique. This makes it possible for a system to uniquely associate inbound traffic with an SA using the SPI number as the key. Outbound traffic processing uses the SPI number to construct the IPsec headers.

Sequence Number Counter: Every IPsec system maintains a counter for each SA that is incremented by one for each packet that it carries. This information is used in IPsec packet processing and anti-replay protection (as will be explained below.)

[13] As to the question of why the SPD's cache allows a faster lookup, I have to direct the reader to RFC 4301. This speedup is hardly relevant to the thesis in question and thus the curious reader is encouraged to start reading at section 5, taking extra care to read the parts about de-correlation.

Anti-Replay Window: A counter and a bit-map used for detecting packets that have been replayed by an attacker[14]. The anti-reply protection is available for the AH as well as the ESP security protocols, but only when using automatic key management (more on this later).

Symmetric encryption information: Encryption algorithm and IPsec protocol type (AH or ESP) used by the SA along with necessary secrets (keys etc) and parameters.

SA Lifetime: A limit at which the SA should be terminated, expressed in either time or the number of bytes transported by the SA.

The PAD (section 4.4.3) is controlled by the administrator and used by the security association management protocol along with the SPD to negotiate new SAs. The core purpose of the database is to store 1) the identification tokens of the peers that the system is allowed to communicate with and 2) the authentication mechanism, parameters and data of each aforementioned peer.

3.4 Understanding the IP headers

Understanding IP headers are key to understanding IPsec. I will therefore quickly review the structure of the IPv6 headers before moving on to explain their relationship to IPsec. Below the readers finds a schematic of the IPv6 header, also called the IPv6 'fixed header' for reasons that soon are to be discussed.

Fixed header format

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	*Version*				*Traffic Class*								*Flow Label*																			
4	32	*Payload Length*																*Next Header*								*Hop Limit*							
8	64	*Source Address*																															
12	96																																
16	128																																
20	160																																
24	192	*Destination Address*																															
28	224																																
32	256																																
36	288																																

Figure 3.5: The IPv6 header. Figure from [Wik13b].

I assume that the reader is acquainted with the general principles of IP networking, the discussion of the above header fields will be brief. *Version*

[14]In a replay attack, a packet is intercepted by a man-in-the-middle with the purpose of delaying or repeating it. This can trick the targeted system into revealing secret information e.g. reusing the same keys in a stream cipher.

is always six; *Traffic Class* (reminiscent of the IPv4 *TOS* field) and *Flow Label* are for flow and congestion control, which are of no concern to us; *Payload Length* is the length of the payload field, including any so called 'extension header' (more on this soon); *Next Header* specifies the packet's transport layer protocol, or the first extension header's type if one is present; *Hop Limit* is equivalent to IPv4's Time-To-Live field which is decremented by each router along its path, ultimately to be discarded when it reaches zero.

IPv4 placed all of its settings into one big header, many settings which as of today are rarely used, and the possibility of extending it with new features are severely limited. The designers of IPv6 solved this problem by keeping the fixed header (figure 3.5) small and allowing it to be extended by a mechanism called *extension headers*. The extension headers contain any optional information, information which can be intended for the packet's destination host and/or intermediate routers along its path. Although the information is usually supplied by the source host, some headers (such as *Fragment*) can be altered and/or added by routers.

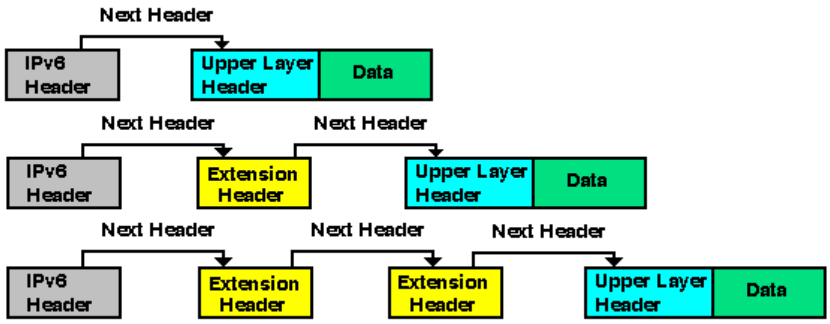

Figure 3.6: Illustration of IPv6's next header scheme. FIX! Not received permission! Source:http://www.zytrax.com/tech/protocols/ipv6.html

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	*Next Header*								*Hdr Ext Len*																							
4	32																																

Figure 3.7: The generic mandatory part of an 'extension header'. Figure from [Wik13b].

The extension header scheme is basically a linked list (figure 3.7). The field *Next Header* denotes the type of extension header to come or, if this is the last extension header, the packet's type of transport layer; *Hdr Ext Len* is the length of this extension header in multiples of eight octets, not including the first eight octets. The rest of the next header contains data

that is specific to each next header type.

Extension Header	Type	Description
Hop-by-Hop Options	0	Options that need to be examined by all devices on the path.
Destination Options (before routing header)	60	Options that need to be examined only by the destination of the packet.
Routing	43	Methods to specify the route for a datagram (used with Mobile IPv6).
Fragment	44	Contains parameters for fragmentation of datagrams.
Authentication Header (AH)	51	Contains information used to verify the authenticity of most parts of the packet.
Encapsulating Security Payload (ESP)	50	Carries encrypted data for secure communication.
Destination Options (before upper-layer header)	60	Options that need to be examined only by the destination of the packet.
Mobility (currently without upper-layer header)	135	Parameters used with Mobile IPv6.

Figure 3.8: This list covers a number of common extensions headers 'next header'. List from [Wik13b].

3.5 The AH protocol in IPv6

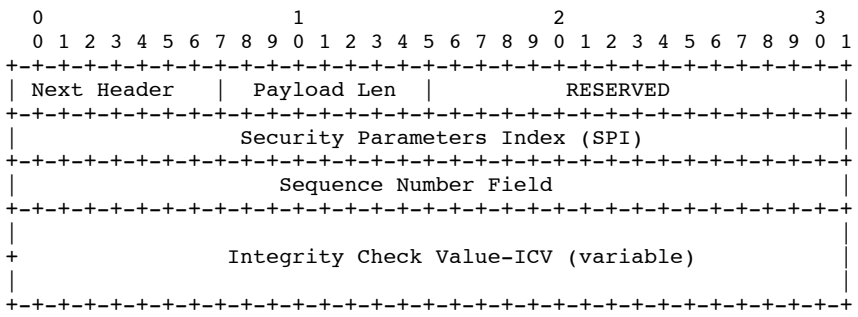

Figure 3.9: The AH extension header. From [Ken05a].

The AH protocol's header (*Authentication Header*) described in RFC 4302[Ken05a] guarantees the *data integrity* and the *data authentication* (origin) of the IP packet which it is applied to. This is achieved by applying a *Message Authentication Code* (MAC) to the packet's content and to the header's immutable fields using a key privy to the two parties.

Figure 3.10 describes the principles of a MAC algorithm: The message, composed of the packet's payload and its immutable fields, are combined and digested[15] along with a key. This produces a *MAC* string (also referred to as a *digest* or *hash* string) which is attached along with the message. IPsec refers to this value as the *ICV*.

Upon arrival the receiver performs the same calculations and compares the resulting *MAC* with that included in the packet. This scheme will, if correctly implemented and algorithms chosen with care, make it computa-

[15] A synonym of *digested* is *hashed*.

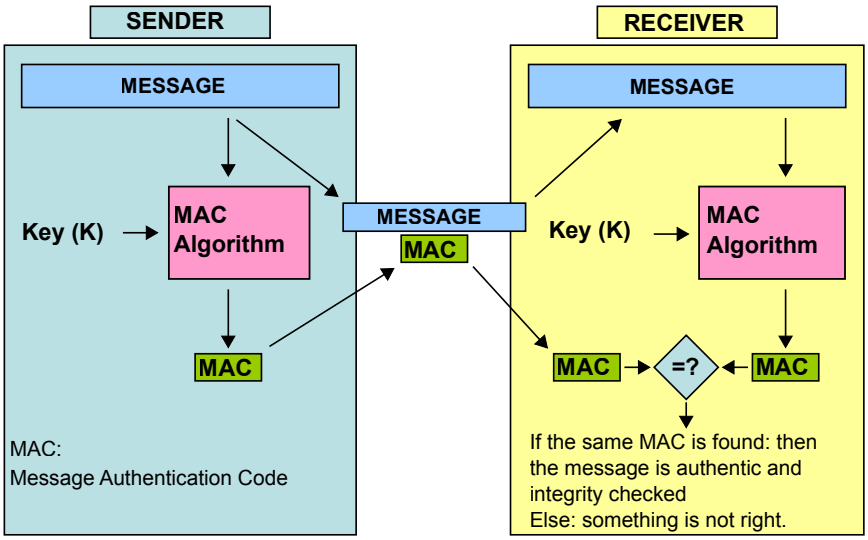

Figure 3.10: Message Authentication Code. From [Wik13c].

tional infeasible[16] for an attacker to alter the message without knowledge of the key. As a side benefit, the packet is also protected against accidental corruption such as transmission errors.

The parameters of the process (such as key, key length and type of MAC algorithm) are SA-specific and are stored in its data structure.

3.6 The ESP protocol in IPv6

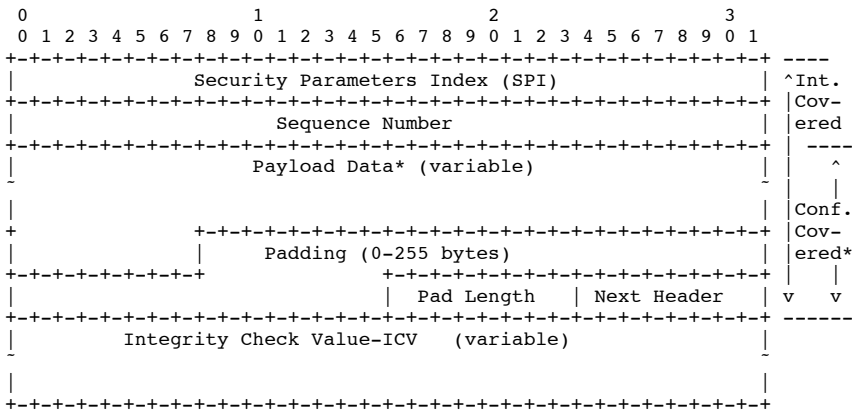

Figure 3.11: The ESP IPv6 extension header. From [Ken05b].

[16] A security attack is said to be 'Computationally Infeasible' if the required cost of performing the necessary cryptographic computations are outside the reach of even very large organizations.

The ESP (Encapsulating Security Payload) protocol described in RFC 4303[Ken05b], provides *confidentiality* and limited *traffic flow confidentiality* in addition to AH's *data integrity* and *data authentication*. The term *Encapsulating* is derived from the fact that the protocol is designed to encase everything that follows it in the packet (extension headers as well as payload). This practice is of benefit in itself, but rather it facilitates the implementation of confidentiality protection. For example, there are many popular algorithms which belongs to the block cipher family e.g. AES-CBC. Such algorithms only operates upon fixed sizes of data and therefore needs to pad[17] the IP packet's payload to a multiple of the block length.

The fact that confidentiality protection is optional makes ESP services identical to that of AH's, but only under certain circumstances. In chapter 'Design and Implementation' it will be shown how this fact can be exploited.

3.7 Automatic key management (IKEv2)

We have now explained the essentials of IPsec: the databases and their relations to each other and how they govern the flow of packets in the IPsec stack. This functionality is all that is required for an interoperable IPsec implementation. However, this requires all SAs to be set by the administrators of the systems in question. I.e. if two such hosts wants to communicate by IPsec the administrators must agree upon, and manually create, the SAs along with their encryption keys and associated settings. Naturally, we would like to automate this in order to lessen the human's workload. This is the purpose of an IPsec Key Management Protocol. At of the time of this document's writing, there are several protocols: IKE (defined in RFC 2407 - 2409)[Pip98][MSST98][HC98]; IKEv2 (RFC 5996)[KHNE10]; KINK (RFC 4330)[Mil06] and IPSECKEY (RFC 4025)[Ric05]. What they all have in common is that they solve the problem of automated key management, albeit in different ways, and that they all interface with IPsec through its databases.

However, this thesis will henceforth only discuss IKEv2 as it's the defacto standard and, more importantly, the original target of study for this work. IKEv2 is the successor to IKE that was widely regarded to be a standard that suffered from unnecessarily complicated and vague standards documents. This issues were addressed in IKEv2 and resulted in, most importantly: better interoperability among implementations; faster keying handshake; implementations with less complex codebase.

[17] To 'pad' data is to prepend or append white space characters (in the case of text documents) or null bytes to a string in order to increase it to a predetermined length.

3.7.1 Overview

As previously stated, IKEv2 is a component that is wholly external to the parts that we have covered so far (IPsec packet processing, SPD, SAD etc). The former are usually, in a general purpose operating system, implemented in the kernel[18] while the latter is written as service running in user space. This is the preferred method on an IPsec host as IPsec's packet processing mechanism needs to be implemented in conjunction with the OS' network stack (referred to as 'bump-in-the-stack' or 'native' in RFC 4301[KS05, p.10-11]) while IKEv2 is shaped into a user-space service. Naturally, we want as much code as possible to run in user space as it facilitates development and allows better integration with the rest of the operating system.

3.7.2 Negotiating SAs

SAs are created when they are required; i.e. when an IP packet is about to leave the host or gateway towards a destination which, according to the policy in the SPD, requires IPsec protection of some sort. Such a packet can be either dropped[19] or stored for later transmission, depending on what suits the implementation best. Now, the IKEv2 service is ordered to establish suitable SAs (according to the security requirements entered in the SPD) with the remote peer in question. All traffic of the same pattern as that of the 'triggering' packet will not be allowed to leave the peer until these SAs are established and entered into the SAD.

IKEv2 communicates with other IKEv2 capable hosts by means of UDP datagrams on port 500. As the purpose of the communication is to negotiate new SAs, the communication can not be protected by any IPsec SAs. Therefore, a correctly configured IPsec host must allow non-protected traffic on this UDP port. In this section I will explain how IKEv2 itself creates a secure communication channel for its datagrams.

I begin by quoting the standard; 'All IKE [IKEv2] communications consist of pairs of messages: a request and a response. The pair is called an 'exchange', and is sometimes called a 'request/response pair'.'[KHNE10, p.5]. Every such request requires a response, and if none is received the sender of the request will re-transmit it until a response is received or a timeout occurs.

The first exchange between two peers (hosts or gateways) that wish to negotiate cryptographic information for a secure communication channel (an SA, that is) is called the IKE_SA_INIT exchange. This will execute the so called Diffe-Hellman key exchange where a shared secret is formed between the peers without divulging it to eavesdroppers. The next exchange

[18] Linux's current IPsec implementation follows this model, and I believe that Microsoft's equivalent for Windows does so as well.

[19] Remember that all parties in an IP-network must tolerate that IP packets are lost on occasion.

is the IKE_AUTH exchange in which the peers authenticates themselves, thus preventing a man-in-the-middle-attack[20]. A secure connection is now established and the associated parameters are stored in a SA called the *IKE SA*.

The IKE SA is only used by the IKEv2 service. Its sole purpose is to establish a secure channel by which the IKEv2 peers can create and destroy *Child SAs*. Such a child SA pair (one for each direction) is now created with cryptographic material derived from the IKE SA (hence the term 'child') in such a way that perfect forward secrecy[21] is upheld. That pair is inserted into the SAD, thus completing the task.

The IKE SA is retained for future tasks such as re-keying, destruction or creation of more child SAs.

[20] Please see 3.3

[21] Keys derived from other, older keys, are said to be derived with PFS if it's performed in such a way that the disclosure of the older keys doesn't threaten the security of the new keys

Chapter 4

Design and Implementation

Implementing Internet standards in Contiki is often a challenging task. The standards are written with the presumption of hosts with vastly more capable hardware, which requires the engineer to simplify parts of the standard, make generalizations, and omit other parts altogether. This might be difficult as it requires good knowledge of the standard in question as well as the behavior of other implementations. The synthesis of this knowledge allows her to save hardware resources by making shortcuts in her implementation, shortcuts which might very well violate the standard document, but still exhibit a behavior which allows communication with the rest of the Internet. This a good example of the very practical, experimental, approach that is typical of Contiki's development culture.

At this point I would like to remind the reader that most implementations of Internet Services are not standards compliant in one aspect or another, but may still be *interoperable* with other implementations. This is especially true of more complex standards which describe many features. IPsec, for example, also describes APIs solely targeted towards the operating system's applications. These are surely neither applicable, nor necessary, in the context of IoT hosts. In this regard, violation of the standard documents is warranted as it will not hamper interoperability.

4.1 Development process

The original goal of this thesis was to develop a working IKEv2 implementation that would operate with an existing IPsec implementation already developed for Contiki. Therefore the literature review started with the RFCs[1] covering that standard, continuing with other documents as my understanding of the field grew.

[1]Described in so the so called RFC (Request for Comments) publication managed by the IETF and the Internet Society

I found the body of standard documents large, complex and the systems described therein rich in features. It was unclear what effects a removal or alteration of a behavior could have on the implementation. Therefore the following principle was decided upon:

> *The guiding design principle of the implementation* is that it must be a subset of the standard and follow the described architecture as close as possible. Any deviation from this requires a thorough analysis of the security implications or the implementation will lose its greatest benefit - the security properties brought by the thoroughly vetted IPsec standard.

Obviously, a thorough investigation and rework of the standards' internals would have the possibility of resulting in a more efficient implementation, but such an endeavor would require an effort that certainly would be outside the bounds of this work.

Simultaneously, I worked with Contiki's source code and documentation in order to figure out how to integrate IKEv2 and the IPsec system into Contiki, a feat easier said than done as Contiki is quite different from the typical multi-user multi-tasking operating system. Implementation then proceeded in a top-down fashion where the data structures were written first, APIs secondly and then finally the actual algorithms.

Contiki targets[2] many different platforms, and two of them is Windows and Linux, i.e. the platform that the compiler is running on. This is called the *native* target and is particular in the sense that Contiki (which is running as an ordinary process in the host's operating system) has access to the wast resources of the personal computer. Needless to say, this greatly facilitates debugging and was therefore used as the predominant build target during the development process. Peripherals such as sensors are emulated and network communication with the host is achieved using a faked serial interface.

During final development the target was changed from *native* to *msp430* (a CPU commonly used in IoT platforms due to its energy efficiency) as the focus turned from functionality to memory and speed improvements. Benchmarking and tuning has to be performed on the correct platform as libraries, the platform's instruction set, and compiler optimization possibilities are target specific. Instead of using actual hardware[3] Contiki was executed on emulated hardware using Contiki's popular simulation environment - Cooja[ÖDE+06]. Apart from providing emulation, Cooja features a complete experimental sandbox which will be further described in the chapter Evaluation.

[2] A compiler is said to be *targeting* a certain CPU's instruction set in the sense that it is configured to produce compatible binaries.

[3] Using real hardware for testing purposes turned out to be a large problem as no readily available platform had the required memory capacity to store the binaries.

4.2 IPsec

At the project's inception, there was already a rudimentary implementation of IPsec in place (described in [RDC+11]). It had the ability to receive as well as transmit AH and ESP headers, but could only handle one set of security parameters, effectively limiting the host to one SA that was used for duplex traffic. This implementation served as a proof-of-concept of the basics of IPsec on Contiki, as well as an IPsec header compression scheme in the underlying MAC-layer. Having stated this, the lack of support for multiple SAs and the inability to negotiate new ones made it difficult to fit it into a real world network environment where communication patterns are subject to constant change and keys need to be renewed as they age and lose their security properties.

In this section, I will begin by describing the IPsec implementation's design specification, its rationale, and most importantly its RFC-compliance. It will then continue with a review of how the technical problems have been solved.

4.3 IPsec Requirements

I decided from early on to implement a subset of IPsec RFC 4301 and associated documents. This was done in order to comply with the *design principle* as outlined earlier, but mostly so because the databases are good at what they do: the implementation can use the SAD to store the SAs; the processing mechanism can use the SPD for policy management etc. There are good reasons as to why theses entities were invented, and leaving them for something else would make the other parts of the system (i.e. IKEv2) harder to implement as the developer would be forced to invent new solutions for doing without these systems. That would increase the work burden.

Tunnel-mode will not be implemented as I believe the need for such functionality will be small in IoT networks using IPv6. The reasoning behind this is twofold:

Firstly, the most common use cases of network-to-network connections are: The single host connecting to a distant network and its resources (e.g. the telecommuter's VPN connection); the branch office's network's router providing a secure route to the main office's network. I believe that the former cases are the principal usage scenarios of today and can't see any good use of tunneling in IoT hosts. However, there are many situations where an entire IoT network (e.g. ventilation system control) would benefit from a secure tunnel between itself and a friendly organization's network (e.g. the ventilation system's operators). If other network paths than the secure tunnel is disabled, the resulting network topography forms a simple

and effective security solution. Having stated this, the secure tunneling code doesn't need to reside in the IoT hosts, but is most conveniently implemented in the network's border router. Secure communication between hosts can still be achieved with IPsec transport mode.

Secondly, I believe that a major reason as to the current preference of tunneling over transport mode is that the latter is incompatible with NAT if used in conjunction with TCP, which is almost always the case. Tunnel mode is presented as the solution in RFC 3715[AD04, 2.1.b)]. However, since this implementation constrains itself to IPv6, this problem disappears as the need for NAT is projected to go away with the introduction of IPv6 networks[4].

ESP along with associated security transforms[5] are described in RFC 4303[Ken05b] and RFC 4835[Man07][6], respectively. I implemented basically all of the functionality labeled as *MUST* in RFC 4303. This resulted in an implementation that basically covered all of the externally visible behavior as described in the RFC, with the exception of the omission of ESN (Extended Sequence Numbers) which were deemed unnecessary as a limit of 2^{32} packets for an SA are sufficient. All security features such as Sequence Number Verification[Ken05b, 3.4.3] were implemented though in order to comply with section 5, Conformance Requirements[Ken05b, section 5].

The document RFC 4835 is the IETF's recommendation as to what security transforms (integrity/authentication, encryption) an IPsec host should be able to handle. The absence of any particular transform is not a security problem per see, but it will lead to one if the hosts then reverts to a common, less secure, transform. In any case, the lack of commonality definitely hampers interoperability. The following algorithms were implemented: AES-XCBC-MAC-96[FH03] (authentication / integrity); AES-CTR[Hou04] (encryption); AES-CBC with 128 bit key[FGK03] (encryption) (FIX: broken in unpack!); NULL[GK98] (encryption). TripleDES-CBC[PA98] and HMAC-SHA1-96[MG98] are labeled as MUST in the standards document[Man07], but were left out as they (firstly) would be more expensive to implement memory-vise and (secondly) afforded equivalent or lesser security than that of the alternatives.

The lack of certain transforms are, however, a small problem as the implementation easily can be extended thanks to the modularity of the IPsec

[4]NAT is commonly used to deal with the lack of IPv4 address space by 'hiding' several hosts behind one public IP. IPv6 is conjectured to resolve this bye virtue of its 2^{128} address space.

[5]'The replacement of the variables in an algebraic expression by their values in terms of another set of variables; a mapping of one space onto another or onto itself; a function that changes the position or direction of the axes of a coordinate system.'[Wik13d]

[6]Labeled 'Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH)'

SA system, which is duplicated in this implementation.

The AH protocol and its associated extension header is not implemented as it's made redundant by ESP in Contiki's μIP stack. The difference between the ESP and AH header is commonly described as that ESP provides confidentiality in addition to the integrity and authentication provided by AH, but that AH provides protection for the IP packet's header and extension headers as well. This is true, but a closer examination reveals that AH's 'extra protection' provides little value in the case of μIP as for the following reasons:

Firstly, most extension headers are exempted from AH's authentication/integrity control since they are designed to be altered by the hosts on the packet's path. For example, consider the *Hop-by-Hop Options* or the *Routing* extension headers. As a counter argument, it can be stated that some extension headers are private to the packet's endpoint, but this argument is void because such headers can be placed inside the ESP header's encapsulation. This practice is inferred by IPv6's standard document[DH98, p.13] which recommends the *Destination Options* header to be placed inside the encapsulation afforded by the ESP header, thus protecting it.

Secondly, the authenticity of the source and destination addresses are important as they are used to match an incoming packet to a socket. E.g. manipulation (spoofing) might allow an attacker with low level authorization to gain access to a concurrent connection enjoying higher privileges. The AH header protects against such attacks, but only if no NAT is employed on the packet's path, in which case it cannot be used. Still, without AH, the IPsec processing directives for incoming packets protects against such attacks by virtue of the following step (p.62, RFC 4301):

> 4. Apply AH or ESP processing as specified, using the SAD entry selected in step 3a above. Then match the packet against the inbound selectors identified by the SAD entry to verify that the received packet is appropriate for the SA via which it was received.

This implies that the cryptographic key used in protecting the packet must match the packet's traffic pattern (i.e. source and destination address, upper layer protocol and, if applicable, port number). Having stated this, it should be noted that this reasoning only holds true when IPsec is not used for multi- or any-cast, something which μIP doesn't support as of today.

Finally, I perceive that I am not alone in questioning the need for AH, even when IPsec is employed in more complicated environments. Personal communication on the IETF's IPsec mailing gave rise to many voices that

wanted to have it removed[7], but most importantly IETF changed the requirement level for AH from MUST to MAY with the introduction of RFC 4301. Another benefit of removing AH from μIP is the possibility of removing the IPsec protocol bit in the underlying 6lowPAN layer as described in [RDC+11], thus freeing up allocated space in its header.

4.4 IPsec Implementation

IPsec is implemented as a part of the μIP stack. The ESP header of incoming packets is unpacked in the loop that handles IPv6 extension headers, a design choice which makes the implementation capable of interpreting the extension header regardless of its position relative to other extension headers.

Incoming packets, protected by IPsec or not, are processed according to the directives described in RFC 4301[KS05, section 5.2] which outlines the processing directives. The steps includes fetching the SA associated with the ESP header's SPI; asserting the packet's integrity; decrypting it and asserting that it's not a replay.

```
while(1) {

  switch(*uip_next_hdr) {
     case UIP_PROTO_ESP:

          if ((sad_entry = sad_get_incoming_entry(esp_header
              ->spi)) == NULL) {
            PRINTF(IPSEC "Dropping incoming protected packet
                because of missing SAD entry\n");
            goto drop;
          }

        /*+  -- REDACTED -- */

          // Assert integrity (if protected)
          if (sad_entry->sa.integ) {
            integ_data.type = sad_entry->sa.integ;
            integ_data.data = (uint8_t *) esp_header;
            integ_data.datalen = auth_data_len;
            integ_data.keymat = &sad_entry->sa.sk_a[0];
            integ_data.out = (uint8_t *) &encr_data.icv;
            integ(&integ_data);
          }

        /*+  -- REDACTED -- */

          // Confidentiality
          encr_data.type = sad_entry->sa.encr;
          encr_data.keymat = &sad_entry->sa.sk_e[0];
```

[7] Please see http://www.ietf.org/mail-archive/web/ipsec/current/msg07184.html

```
            encr_data.keylen = sad_entry->sa.encr_keylen;
            encr_data.integ_data = (uint8_t *) esp_header;
            encr_data.encr_data = iv;
            encr_data.encr_datalen = auth_data_len - sizeof(
                struct uip_esp_header);
            encr_data.ip_next_hdr = uip_next_hdr; // Non-zero
                to indicate ESP header
            espsk_unpack(&encr_data);

            // Verify ICV
            if (memcmp((uint8_t *) esp_header + auth_data_len,
                 &encr_data.icv, sizeof(encr_data.icv))) {
              PRINTF("IPsec: ICV mismatch, dropping packet.\n"
                  );
              goto drop;
            }

            // Replay protection (dynamic SAs only!)
            if (SAD_ENTRY_IS_DYNAMIC(sad_entry) &&
                sad_incoming_replay(sad_entry, uip_ntohl(
                esp_header->seqno))) {
              PRINTF(IPSEC "Error: This packet is a replay\n")
                  ;
              goto drop;
            }
```

Thanks to optimization, much of the code used for processing incoming and outgoing packets is shared. Note that the implementation simply drops outgoing packets that requires protection, but which have no appropriate SA. Such an SA will (hopefully) be created by the IKEv2 negotiaton that is triggered upon the packet's rejection. This policy removes the need for a resource-hogging queue of to-be-transmitted packets, and is permissible according to the standards[KS05, p.53].

```
// Protect packets that are sourced from us, not ones routed
    on the behalf of others
if(! uip_ds6_is_my_addr(&UIP_IP_BUF->srcipaddr)) {
  PRINTF(IPSEC "This outgoing packet is being forwarded.
      Bypassing IPsec stack.");
  goto bypass;
}

/*+  -- REDACTED -- */

// We use the SAD as an SPD-S cache (RFC 4301). Is there an
    SA entry that matches this traffic?
sad_entry_t *sad_entry = sad_get_outgoing_entry(&packet_tag)
    ;

// If not, assert that it's in accordance with the policy of
    this traffic. (RFC 4301, p. 53, part 3b.)
if (sad_entry == NULL) {
  spd_entry_t *spd_entry = spd_get_entry_by_addr(&packet_tag
```

```
      );

  switch (spd_entry->proc_action) {
    case SPD_ACTION_PROTECT:
    PRINTF(IPSEC "SPD: Outgoing packet targeted for PROTECT,
        but no SAD entry could be found." \
      " Dropping this packet and invoking the IKEv2 service
          for SA negotiation.\n");

    // This asynchronous call will be processed after
        uip_process() has finished
    ike_arg_packet_tag = packet_tag;
    process_post(&ike2_service, ike_negotiate_event, (void
        *) spd_entry);

    // RFC 4301 grants us the permission to drop the packet
        triggering an IKE handshake
    goto drop;

    case SPD_ACTION_BYPASS:
    PRINTF(IPSEC "SPD: Outgoing packet targeted for BYPASS\n
        ");
    goto bypass;

    case SPD_ACTION_DISCARD:
    PRINTF(IPSEC "SPD: Outgoing packet targeted for DISCARD\
        n");
    goto drop;
  }
}
// We will now proceed to protect this packet with the SA in
     sad_entry

if (sad_entry->sa.proto == SA_PROTO_ESP) {
  /*+  -- REDACTED -- */

  if (!esp_header->seqno) {
    IPSECDBG_PRINTF(IPSEC "Error: Sequence number overflow.
        Removing SAD entry.\n");
    sad_remove_outgoing_entry(sad_entry);
    goto drop;
  }

  /*+  -- REDACTED -- */

  espsk_pack(&encr_data);

  /*+  -- REDACTED -- */

  integ(&integ_data);

  /*+  -- REDACTED -- */
}
```

The various helper functions dealing with services of the SAD; SPD; packing, unpacking and integrity of the ESP header, etc are not shown here, but are available in the appendix.

4.5 IKEv2 Requirements

The prime requirements of the IKEv2 service is to negotiate, maintain and delete SAs shared with other hosts. All other requirements follows from this. In the remainder of this section I will outline the requirements and justify their rationale.

The IKEv2 protocol requires an IKEv2 peer to establish an *IKE session* with the host that it wish to establish one or more IPsec SAs with. This includes the cryptographic exchange that serves to protect the *IKE session* as well as any future IPsec SA, as the later derives its keying material from it. Naturally, all features required for this handshake has to be supported and implemented in such a way that their security properties are not rendered void. Furthermore, the IKE service has to be able to accommodate multiple concurrent sessions as I believe it will offer flexibility for the applications running on the host (e.g. you, as an application developer, doesn't need to care about how many secure connections your application will open).

The exchange of cryptographic keys is performed by means of a *Diffie-Hellman* (DH) key exchange[8]. This exchange is very hardware-intensive in regard to memory as well as calculations (and thus energy), as is typical of asymmetric encryption schemes. This has limited the use of such security mechanisms in IoT systems.

Briefly, the reason that asymmetric encryption is so resource intensive is that it's centered around operations on very large numbers. The operations are very simple (e.g. $A = g^a mod p$), but in the case of the popular RSA scheme, the numbers a and A, which forms the so called public and private key, respectively, can be so large that they require 3072 bits to represent. A DH exchange with such a long key is most often not feasible in an IoT context.

The method used by RSA (and that is referred to in the example above) is a DH key exchange using certain qualities inherent to multiplicative groups of integers modulo n. A development of this idea, called *Elliptic Curve Cryptography (ECC)* allows a DH exchange using much smaller key sizes, without forsaking security. Considering only the DH exchange, ECC provides

[8]The Diffie-Hellman key exchange is a scheme that is commonly used for exchanging keys over insecure channels (such as the Internet) in a secure fashion. Diffie-Hellman, and its successor, the RSA scheme, are widely used by Internet applications.

no technical drawbacks in comparison to RSA, but only benefits in terms hardware resources and energy. Because of this, IETF standardized several ECC parameters (ECP groups) for the use in the IKEv2 protocol (RFC 5114: Additional Diffie-Hellman Groups for Use with IETF Standards[LK08]). I believe that the reason as to why ECC isn't already in widespread use is that 1) some parts of ECC is encumbered with patents 2) implementations of RSA are widespread and accepted as the standard solution to asymmetric encryption.

GROUP	SYMMETRIC	RSA
1024-bit MODP with 160-bit Prime Subgroup	80	1024
2048-bit MODP with 224-bit Prime Subgroup	112	2048
2048-bit MODP with 256-bit Prime Subgroup	112	2048
192-bit Random ECP Group	80	1024
224-bit Random ECP Group	112	2048
256-bit Random ECP Group	128	3072
384-bit Random ECP Group	192	7680
521-bit Random ECP Group	256	15360

Figure 4.1: In section 4, 'Security Considerations', of RFC 5114[LK08] the authors argue that ECC keys can be made much shorter without sacrificing security. Each line is cross-reference of different encryption types and key lengths offering the same cryptographic security. Notably, the benefits of ECC becomes even greater as the level of security is increased.

In order to evaluate the possibilities of ECC in a resource constrained host, I decided that the IKEv2 implementation should implement the DH key exchange by means of ECC, using one of the shorter groups as seen in 4.1.

IKEv2 supports several authentication methods: public key signatures, shared secrets and the Extensible Authentication Protocol (EAP)[ABV+04]. As EAP is more complicated (i.e. higher complexity) than the other methods, I believe it's of no interest to this implementation and thus I will only focus on the former methods. Section 2.15 in RFC 5996 serves as a good introduction to the mechanisms used by the former methods (public key and shared secrets). What both schemes have in common is that they both perform a cryptographic operation on a string which is a concatenation of different IKEv2 messages and values shared by both parties[KHNE10, p.48]. (Please note that the formerly described DH key exchange only provides the hosts (peers) with a shared secret, not authentication. Thus, without any type of authentication, we're liable to a Man-In-The-Middle attack.)

The public key method (PKCS#1) features the most flexibility and ease-of-use as it allows the peers to authenticate themselves using well known X.509 certificates, possibly using a public key infrastructure. Still, this

method has not been implemented as it requires the implementation of digital signature algorithms[9]; handling the X.509 certificates; the certificate's payloads; etc. Not only would the required work effort be prohibitive, but the value of the work would be questionable as the extra asymmetric operation[10] and the associated code itself would probably make it unpractical on current hardware. Still, it will certainly be practical on the future hardware reference platform that I define in section 6.2 'Evaluation results'.

Because of the aforementioned reasons, I have opted to implement the Shared Key authentication method as described in [KHNE10, p.49]. As it only requires the implementation of a simple pseudo-random hashing function (as described in section 2.13), the demands on ROM and CPU (and thus energy) are limited. The downside of this scheme is that it requires a string (the key) to be manually shared and maintained between the hosts that wishes to communicate. Although this gives rise an administrative hassle and arguably weaker security than the certificate alternative, it certainly affords security that's good enough for many situations.

4.6 IKEv2 Implementation

The IKEv2 implementation supports several concurrent sessions, each modeled as a mealy state machine[11] that persists as long as the IKE SA is alive. As I will show, this enables communication with multiple hosts without excessive memory consumption.

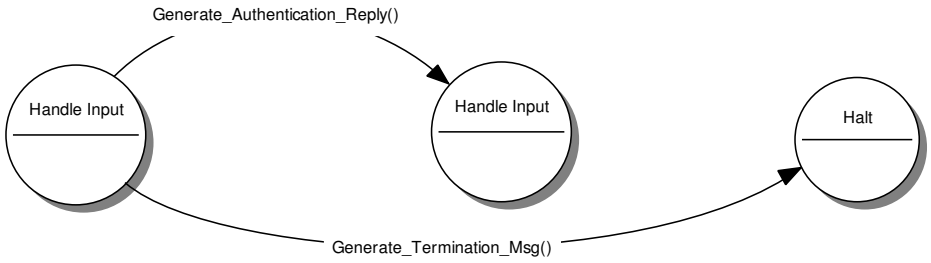

Figure 4.2: State machine example

As to begin the explanation, I would like to introduce the reader to the terminology used by the standard's document (RFC 5996 [KHNE10], hereafter referred to as 'the RFC'). Therein, IKEv2 hosts are named *peers* and

[9] I had access to code that implemented digital signatures using Elliptic Curve Cryptography, but the work required to implement it with IKEv2 would have been far too large.

[10] We already perform one asymmetric operation during the DH key exchange

[11] In a mealy state machine the output is function of the current state as well the current input, i.e. the output is a function of the transition.

their communications are called *exchanges* consisting of a pair of *messages*. There are four different types of exchanges, here displayed in order of usage during the lifetime of an IKE SA[12]: IKE_SA_INIT; IKE_AUTH; CREATE_CHILD_SA; INFORMATIONAL. On the wire, each such message is an UDP datagram that begins the with the *IKE header* and contains one or several *IKE payloads*.

In the IKEv2 implementation, every *message* is received by ike_statem_incoming_data_handler() which multiplexes the message on the basis of the unique identifier in the IKE header (the IKE SA's SPI). Each such SPI is associated with an IKE session and a corresponding state machine which represents that session's state. This determines what *message* should go to which state machine instance.

In the state machine, each *state* is represented by a function that, on the basis of the message received from the other peer, makes a *decision* of how to act, i.e. what message to send in reply. That message is created and transmitted by its associated *transition*, which itself is implemented as a function which only takes the session structure's pointer as its argument. This is a powerful feature of the implementation because it allows a message to be retransmitted without resorting to buffers storing the last transmitted message. Instead, the transition's function (whose pointer resides in the session struct) is simply run again. This saves several hundred bytes of RAM which otherwise would be required for buffer space. As the reply is received, the corresponding state function (determined by the session's data structure), is called and the process repeats.

The data structures and associated mechanism of the previously mentioned machines are interesting, but because of space constraints, they will not be reprinted here. Instead, the curious reader is directed towards the source code and machine.c and machine.h in particular.

Finally, before continuing with the description of the initiator machine, i would like to explain the terminology used for string processing. An example of such an operation is VAR = Ni | Nr. Here the variable VAR is assigned the value of the variable Ni concatenated with variable Nr. 'N' is in this case an abbreviation of 'Nonce' - a variable that exists at both of the peers, albeit with different values. Thus, 'Ni' is the nonce of the initiator and 'Nr' is the nonce the responder. This notation is commonly used throughout the standards document.

4.6.1 Initiator machine

An IKEv2 session is started or *initiated* by one of the peers. The other peer that is contacted by the latter is then considered to be the *responder*. The state machines for the two roles are slightly different, in the sense that

[12] An IKEv2 'session' (i.e. connection between two peers) are denoted as the IKE SA, which is the data structure containing the peer's shared cryptographic secrets.

the steps in the handshake process comes in a different order, but they are essentially the same and most of the source code are in common. This part describes the initiator machine, but it will allow you to understand the responder as well.

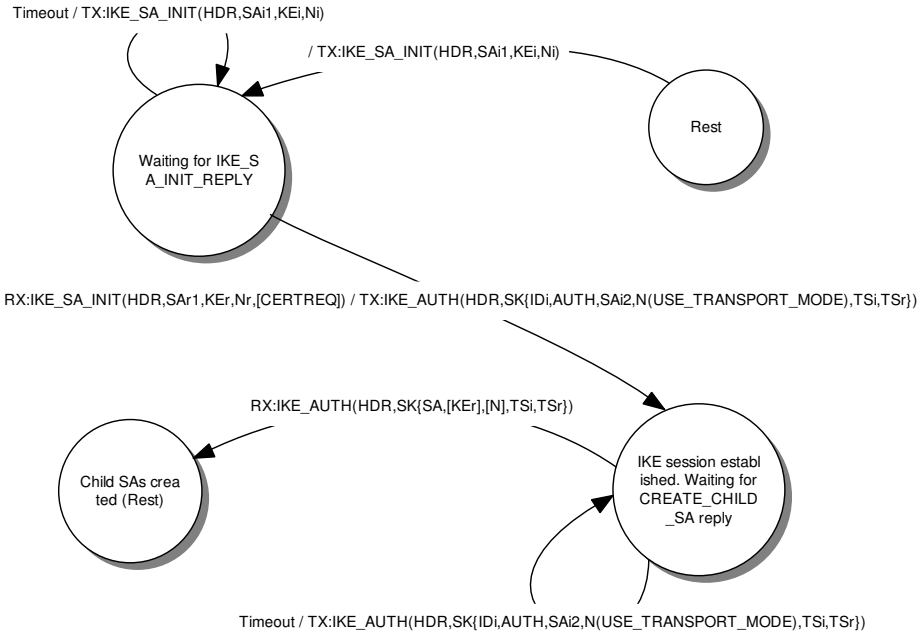

Figure 4.3: The IKEv2 initiator's state machine

An instance of the initiator machine is (as displayed in figure 'The IKEv2 initiator's state machine' on page 39) is **started** when an application on the same Contiki host attempts to transmit a packet that matches the *traffic selector* of an SPD rule labeled *PROTECT* that has no matching SA. Now, the IKE session's data structure is setup by ike_statem_setup_initiator_session(ipsec_addr_t *triggering_pkt_addr, spd_entry_t *commanding_entry) in machine.c. That data structure consists of two parts: one persistent and one ephemeral. The former contains the IKE SA (parameters, keying material used to derive subsequent child SAs and the IKE SA's authentication / integrity- and encryption keys); the remote peer's IP address; the IKE SPIs[13]; data for connection house keeping such as message ID counters and retransmission timer; current state machine transition (i.e. the function that generated the last *message*); target state (i.e. the function that should handle any *responses* received in regard to the sent *request*, or vice versa). The later (ephemeral) data structure stores information that is only of interest during the IKE session's setup: Pointer to the SPD entry

[13]Security Parameter Index, the IKEv2 session identifier

that triggered the negotiation; storage for cryptographic nonces; storage for the peer's SA proposals; the private Diffie-Hellman key. Memory for both data structures are allocated with malloc(). In addition to this, the cryptographic nonce and the private key used in the Diffie-Hellman exchange is generated via a pseudo-random function. The reason that the private key is generated now and stored in the ephemeral data structure (to be thrown away after the handshake is complete) is that we lower the memory requirements, while key generation is comparably cheap energy-wise. Needless to say, this 'trick' will have to abandoned is X.509 certificates or similar is to be used for authentication as the remote peer in that case expects a *particular* private/public key pair, and not just anything[14].

The first exchange, of type IKE_SA_INIT, now begins when the state machine leaves the rest state and transmits the first message. The message consists of three payloads: the cryptographic offer for the IKE SA (i.e. suggested suite of integrity / authentication and encryption transforms); the peer's public Diffie-Hellman key; the cryptographic nonce that later one will be used for authentication the peers. It's also at this point that the public key is computed, something which takes considerable time (see chapter 'Evaluation').

The state machine is now in state #2 and **waits for the peer's reply**, a message containing the very same payloads that we just sent. If the peer accepted our IKE SA offer, the SA payload will now contain the peer's choice of cryptographic transforms, a subset of our previous offer. Likewise, if the peer found our choice of asymmetric algorithm acceptable, the KE payload will contain the peer's public key calculated for the same parameters[15]. Both peers now have a shared Diffie-Hellman secret and can thus form an IKE SA using the secret as the source of keying material. This is done in the following way:

The IKE_SA allows the peers to communicate securely, but only if the risk of a possible man-in-the-middle attack is discounted. This will be remedied in the next exchange, IKE_AUTH, which will authenticate the peers.

Given that we can parse the peer's message without trouble, **the state machine transmits the first message of the IKE_AUTH exchange** by transiting to the next state. This message consists of one encapsulating payload which encrypts the others using the IKE SA. Its structure is almost identical to that of the ESP header, which allowed me to implement the encoding and decoding of these two payloads / headers in the same C-function. The encapsulated payloads are: ID (an e-mail address - one of seven different ID methods); Authentication data (AUTH); an offer of cryptographic transforms for the child SA pair (i.e. the SAs that will be used

[14]N.B. this implementation uses the pre-shared key method ([KHNE10, p.49]) for authentication.

[15]A Diffie-Hellman key exchange's cryptographic transform is determined by its IKE DH group, in IKEv2 parlance.

```
SKEYSEED = prf(Ni | Nr, g^ir)
{SK_d | SK_ai | SK_ar | SK_ei | SK_er | SK_pi | SK_pr } =
   prf+ (SKEYSEED, Ni | Nr | SPIi | SPIr )
```

Figure 4.4: Deriving keys for the IKE SA and keying material for future Child SAs. g^{ir} is the shared Diffie-Hellman secret. N* strings are the responder's and the initiator's nonce strings; SPI* are the IKE SA's SPIs numbers (used for identification); SK_a* and SK_e* is the IKE SAs cryptographic keys; SK_d is a string from which keying material can be derived for future child SAs (hence 'child' in the sense that it inherits keying material from a parent SA); SK_p* is material that will be used for authenticating the peers during the IKE_AUTH exchange. The functions prf and prf+ are described in figure 'The functions prf and prf+' on page 41.

```
prf = HMAC (Key, String)

prf+ (K,S) = T1 | T2 | T3 | T4 | ...
T1 = prf (K, S | 0x01)
T2 = prf (K, T1 | S | 0x02)
T3 = prf (K, T2 | S | 0x03)
T4 = prf (K, T3 | S | 0x04)
```

Figure 4.5: The prf function is an HMAC-function and is negotiated during the IKE_SA_INIT exchange. This implementation uses HMAC-SHA1 exclusively. The utility of the prf+ function is the ability to generate strings of keying material of any length i.e. regardless of the prf function's output. The prf+ function is defined on p.46 in the RFC.)

by IPsec); Notification payload that informs the other peer that only Child SAs for IPsec transport mode are acceptable; Traffic Selectors - one for each direction of the traffic flow that the Child SAs will carry.

As previously stated, IKEv2 can use both digital signatures as well as shared keys, this implementation making use of the latter. In common to both methods is that their authentication data is based upon a signature or hash of the string in figure 'Initiator machine' on page 42.

The fact that the string described in figure 'Initiator machine' on page 42 includes previously transmitted messages (RealMessage1) incurs an extra memory overhead as the messages needs to be buffered. In this implementation I partly alleviate this problem by only buffering the remote peer's message. The peer's own message is reproduced by simply re-running the

```
InitiatorSignedOctets = RealMessage1 | NonceRData | MACedIDForI
GenIKEHDR = [ four octets 0 if using port 4500 ] | RealIKEHDR
RealIKEHDR = SPIi | SPIr | . . . | Length
RealMessage1 = RealIKEHDR | RestOfMessage1
NonceRPayload = PayloadHeader | NonceRData
InitiatorIDPayload = PayloadHeader | RestOfInitIDPayload
RestOfInitIDPayload = IDType | RESERVED | InitIDData
MACedIDForI = prf(SK_pi, RestOfInitIDPayload)
```

Figure 4.6: The string to be signed is a concatenation of the peer's nonces; previously transmitted messages and more.

transition associated with the message in question.

```
AUTH = prf( prf(Shared Secret, "Key Pad for IKEv2"),
<InitiatorSignedOctets>)
```

Figure 4.7: Computing authentication data using the shared secret method. From p.49 RFC 5996.

Finally, using the shared secret authentication method, the authentication data for the AUTH payload is computed as seen in figure 'Initiator machine' on page 42.

The traffic selectors are used for negotiating what kind of IP packets that can be transported across the child SA. The initiator's role is to negotiate a traffic set (traffic selector) that resembles the SPD rule that triggered the handshake as closely as possible. The responder will try to be as accommodating as possible, but might reply with only a subset[16] of the initiator's traffic set in order to not violate its own policy. In the worst case the two peers' SPD configurations are inherently incompatible and the traffic negotiation will fail, causing the creation of the child SA to be abandoned.

The implementation constructs a set of traffic selectors that are in compliance with the standard: the first selector pair have the same address parameters as that of the triggering packet (i.e. the same IP address, the same ports etc); the second is a copy of the triggering SPD rule, but the rule's IP address range is replaced with the IP of the peer[17].

The state machine now expects an IKE_AUTH reply from the peer. Just as with the IKE_SA_INIT exchange, we expect to see the

[16] This is referred to as 'narrowing' in the RFC.

[17] The RFC refers to such address replacement as Populate-From-Packet (PFP). It also dictates that the administrator should be able to set a PFP flag for each one of the fields in the address selector, but I ignored that and hardcoded the PFP flag for the address field.

same payloads in return: The peer's ID; peer's authentication data; child SA transform reply; traffic selector reply.

The authentication data is re-computed, but using the peer's values instead, according to the procedure in figure 'Initiator machine' on page 42 and figure 'Initiator machine' on page 42. If the computed value corresponds to that in the received AUTH payload, we can be certain that the peer that we share the IKE SA with is also in possession of the *shared secret* referred to in figure '**??**' on page **??**.

With authentication complete, the state machine can now create the child SAs (each being half-duplex with respect to traffic flow). The child SA transform (encryption, intergrity and authentication settings) reply is expected to be a subset of that we sent to the peer, if it found it acceptable. If the transform set is indeed a subset of the offer, the transforms are accepted. The corresponding procedure is carried out for the traffic selector set: the reply is accepted if it's a subset of the offer.

```
KEYMAT = prf+(SK_d, Ni | Nr)
```

Figure 4.8: Creating keying material for the Child SAs.

The keys for the cryptographic transforms of both Child SAs are taken from the KEYMAT string in the order described on page p.52 of the RFC. The SAs are now inserted into the SAD and the state machine frees up the ephemeral data structures. The IKE SA is retained for future use, but may be terminated if memory if so wished.

4.6.2 Responder machine

The responder shares almost all of its code with the initiator. What sets them apart is the order by which strings are concatenated (e.g. consider Ni | Nr) and the structure of the state machine.

4.7 Supporting libraries

The suite makes use of several cryptographic algorithms e.g. SHA1, HMAC, AES, etc. This functionality is provided to the software suite in the shape of libraries provided by other authors. These are written with the explicit purpose of being performance-efficient and are therefore well suited to this project. Here follows a list of the libraries used and their contribution to the suite:

ContikiECC is my port of TinyECC, a library for asymmetric encryption using elliptic curves (ECC) originally created for TinyOS. It contains

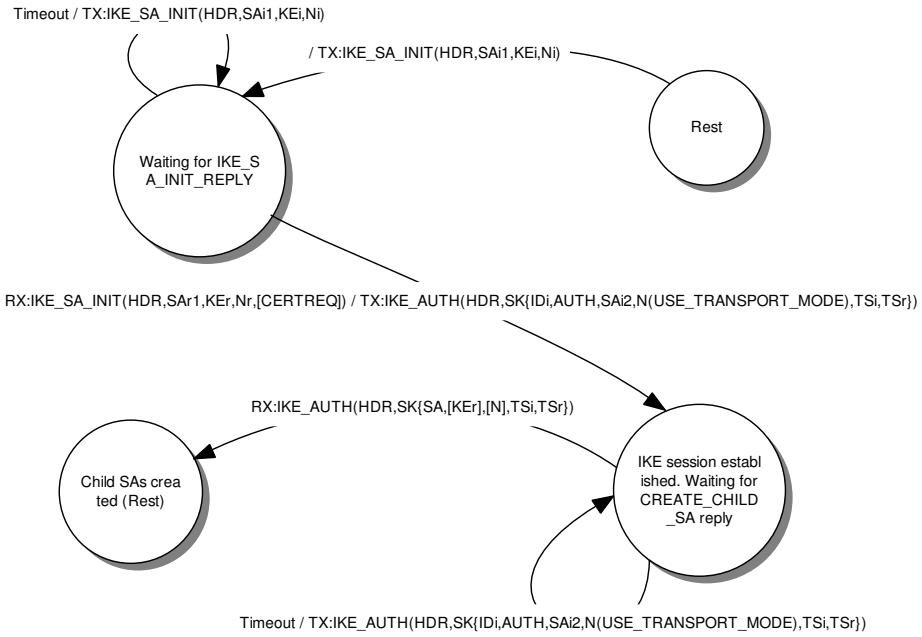

Figure 4.9: The IKEv2 responder's state machine

a multiple precision arithmetic library upon which elliptic curve encryption functions are implemented. These are used by IKEv2's Diffie-Hellman key exchange. Support for ECC-based Digital Signatures, which can be used for X.509 certificates, is also implemented, but is not used by the IKEv2 service.

The AES encryption standard is implemented in software. Previous IPsec implementations[RDC+11] have made use of hardware implementations, but this was not possible as the radio module that provided the functionality in question was not part of the chosen development platform (Wismote, see chapter 'Evaluation'). The library in question is a part of the popular MIRACL SDK.

The HMAC-SHA1-96 standard as defined in RFC 2104[KBC97], is used by IPsec as an integrity/authentication mechanism (see RFC 2404[MG98]), and by IKEv2. The HMAC implementation is also a part of the MIRACL SDK, but I altered it in such a way that it could utilize ContikiECC's SHA1 implementation.

Chapter 5

Evaluation

The software was evaluated using the Cooja emulator as stated in section 4.1 'During final development'. The emulated mote (Wismote) features a 16 bit CPU with an instruction set (MSP430x) that includes instructions for 20 bit memory operations and runs at 16 MHz. There is 16 kB RAM and 128 kB of flash (for storing program data). Usually IoT application like this is evaluated on the Tmote Sky platform, but as I will soon show you, its 16 bit memory space is too limited (keep in mind that the RAM and flash area occupies the same 2^16 address space).

The other peer consisted of Cooja's host using the Linux kernel's native IPsec implementation and the Strongswan software suite for providing the IKEv2 services.

5.1 IPsec implementation coverage

The IPsec part of this implementation should be, to the best of the author's knowledge, compliant with all relevant standards with the exception of certain features (see section 5.1.2 'Major features not implemented').

5.1.1 Major features implemented

- Packet replay protection
- The ESP header
- IPsec Transport mode
- Traffic selectors (traffic multiplexing over SAs)
- SAD, with mechanism for SA expiration
- SPD and related functions (including one SA offer for each SPD rule)
- Confidentiality protection (Encryption):

  - ESP-NULL
  - AES-CTR, varying key length (only 128 bit tested). This was a partial rewrite of Simon Duquennoy's code.

- Integrity protection:
  - AES-XCBC. This was a partial rewrite of Simon Duquennoy's code.

5.1.2 Major features not implemented

- The AH header
- IPsec Tunnel mode
- Various integrity and encryption algorithms that the standards document regards as required.

The lack of support for the AH header and tunnel mode is of no consequence for reasons as described in paragraph 4.3 'The AH protocol and its associated extension header is not implemented' and paragraph 4.3 'Tunnel-mode will not be implemented', respectively. The lack of support for encryption transforms other than AES-CTR and AES-XCBC can be hampering to interoperability, but this IPsec implementation is fashioned in such a way that new transforms can be implemented with ease.

5.2 IKEv2 implementation coverage

The IKEv2 implementation is a subset of the standard described by RFC 5996. It's not standards compliant because of missing features.

5.2.1 Major features implemented

- State machine for responder
- State machine for initiator
- ECC DH (only featuring support for DH group no. 25)
- Authentication by means of a pre-shared key
- Traffic selector negotiation (only a simplification of [KHNE10][section 2.9]'s algorithm)
- SPD interface that allows multiple IKE SA offers
- SK payload: Code for encryption and integrity is shared with IPsec's ESP

- Multiple concurrent sessions supported
- Multiple child SAs per IKE SA

5.2.2 Major features not implemented

- Cookie handling (the code has been written, but it has not been tested)
- Only tunnel mode negotiation supported
- No EAP support
- No NAT support
- No IPcomp support
- Only IDs of type ID_RFC822_ADDR (e-mail address) are supported
- No support for Certificates as IDs, nor for authentication
- No state machine for established sessions

The established state machine was not implemented because of time constraints, but doing so would be a straightforward task and would not require any changes to the architecture of the software. This resulted in that the IKEv2 service doesn't recognize the Delete Payload and thus cannot delete SAs, nor can new Child SAs be created.

5.3 Test setup

The software can be configured to be built with different feature sets so that the developer can reduce the resource requirement. This gives rise to a large number of feature-sets that can be evaluated. In order to reduce the size of the investigation to manageable proportions, I have elected to limit myself to the two most important configurations:

- IPsec with manual keying and a test application that replies to UDP packets. Henceforth named NOIKE.
- IPsec with automatic keying only, IKEv2 and a test application that replies to UDP packets. Henceforth named WITHIKE.

NOIKE's SAs were configured to use AES-XCBC-MAC-96[FH03] for integrity / authentication and AES-CTR[Hou04] for confidentiality. The key length of the latter was 128 bits.

The IKEv2 offer of the WITHIKE system was set up in such a way that it and its peer negotiated SAs of the same configuration as that of the NOIKE

system. The handshake was triggered by the PC by attempting to transmit a UDP packet (using the *nc* utility) from the PC to the mote using a port that was configured for protection. That caused Strongswan's *charon* demon initiated a key exchange with the mote's IKEv2 service (making use of the IKEv2 service's responder state machine). The key exchange used 192 bit ECC keys (the curve is named secp192r1 as described in 'Additional Diffie-Hellman Groups for Use with IETF Standards' in RFC 5114[LK08, p.5]). Its security level is deemed to be comparable to that of a symmetric encryption scheme with a key length of 80 bits[LK08, p.11]. A shared secret was used for authentication as this is the only type of authentication implemented in the service as of today.

5.4 ROM and RAM requirements

The image's ROM requirement was determined using the following method.

> A 'bare' Contiki system is compiled (i.e. one without IPsec) and the image named contiki-bare; Another Contiki image with the feature in question included was then compiled and named in the style of contiki-FeatureName; the size of of the two images' segments are now compared with a suitable tool such as 'size'. The three segments are listed here with their usual members[1]: Data (initialized variables), BSS (uninitialized variables) and Text (Program code and constants).
>
> All code was compiled with MSPGCC 4.7.2 without debug symbols and using the optimization flag Os which emphasizes text segment reduction over execution speed (e.g. limited loop unrolling). Debug symbols and diagnostic messages have been excluded from the image.

This resulted in the following figures for a default Contiki system (i.e. no IPsec / IKEv2 features), NOIKE and WITHIKE, respectively:

5.5 Stack requirements

If the stack grows to large it will overwrite the heap or simply exit the RAM's memory area. Therefore, we want to know its maximum extent in different situations.

The stack requirements for the NOIKE configuration have not been measured as they are considered very limited - probably not more than 100 B.

[1] There is nothing in the specifications for the C language that stipulate where certain language constructs should be stored in the resulting image. For example, constants are usually stored in the text segment, but we cannot be certain without reading the compiler's documentation.

```
 text    data     bss     dec filename
55734     264    9310   65308 manualsa-noike.wismote
47502     252    8960   56714 noipsec.wismote
74241     288    9582   84111 nomanualsa-ike.wismote
```

Figure 5.1: Raw figures for ROM and RAM requirements as given by the 'size' command.

Configuration	ROM (text + data)	RAM (data + BSS)
default (Contiki with μIP6)	46.6 kB	9.0 kB
NOIKE	54.7 (Δ8.1) kB	9.4 (Δ0.4) kB
WITHIKE	72.8 (Δ26.2) kB	9.6 (Δ0.6) kB

Figure 5.2: ROM and RAM requirements, with size difference relative to the default configuration noted.

This can be assumed by the fact that the IPsec source code 1) is not deeply nested (i.e. depth of the function calls) 2) there are no large variables in those function calls.

However, the stack requirements of the WITHIKE configuration are extensive because of the IKEv2 service's need to 1) concatenate large strings in order to compute their hashes 2) perform asymmetric encryption operations. Therefore we will only investigate the stack usage of the WITHIKE option.

The size of the stack was measured by writing a 2 kB long string consisting of the letter 'h' to the top of the stack before commencing the IKEv2 handshake. That string's memory was then investigated after the handshake to see what parts had been overwritten by the service's handshake routines:

```
for ( i = STACK_MAX_MEM - 5;
      i > 4 && strncmp((const char *) (buff + i), "hhhhh", 5);
      i -= 1)
      ;
```

Figure 5.3: STACK_MAX_MEM is set to 2 kB and the pointer *buff* point to the string. Because the stack extends downwards on the MSP430 architecture, the loop starts investigating the end of the string as it's closest to the rest of the stack. It exits when it finds the 'h' pattern.

The maximum extent of the stack during the handshake was found to be 320 Bytes according to this method.

Exchange	Maximum extent of stack
IKE_SA_INIT exchange	321 B
IKE_AUTH exchange	203 B
After handshake	Not applicable

Figure 5.4: Maximum extent of the stack at different times during the key exchange.

5.6 Heap requirements

The use of heap allocation is not customary in Contiki - fixed buffers are used instead. However, I elected to use the heap (by making use of the *malloc* call) in the IPsec as well as the IKEv2 code as the memory requirements varies extensively by activity.

The memory usage was traced by creating a wrapper function around the call to *malloc* and *free* that kept statistics on memory allocation. This resulted in the following memory pattern for the node when acting as the IKEv2 responder:

Exchange	Maximum amount of heap memory allocated
IKE_SA_INIT exchange	930 B
IKE_AUTH exchange	1142 B
After handshake	402 B

Figure 5.5: Maximum size of the heap at different times during the key exchange.

Of the remaining memory, $2 * 106B$ is used for the SAs[2] and 190 B is consumed by the IKE SA. If more SAs are created using this IKE SA, the memory consumption would increase on the order of $2 * 106B$ per duplex connection.

5.7 Latency

The CPU time consumed by the IKEv2 service during a key exchange is 10.9 seconds when run on the Wismote. This excludes the time required to transmit the messages and the other peer's delay. The total time (i.e. clock time) for a key exchange is 11.1 seconds. That includes message transmission time as well as the delay caused by the other peer.

The latency / CPU time associated with the NOIKE option was not investigated as the bulk of its execution time is spent in code that is virtually the same as the one found in [RDC+11] (the code in question concerns

[2]Keep in mind that IPsec requires one SA for each direction

Exchange	CPU Time consumed
IKE_SA_INIT exchange	6.91 s
IKE_AUTH exchange	3.97 s
Total	10.87 s

Figure 5.6: Time requirements for the different phases of the key exchange.

cryptographic operations). The paper in question features a number of investigations covering latency and energy usage of the code concerned.

5.8 IKEv2 total RAM consumption at different times

The RAM consumption of the IKEv2 service is changing as the system boots; SAs are established; and the service returns to an idle state. Investigating these figures allows us to understand how much memory that is left for applications at different times.

State	$BSS + data$	Max. extent of stack	Max. extent of heap	Total Maximum
After boot up	9.6 kB	0 kB	0 kB	9.6 kB
IKE_SA_INIT exchange	9.6 kB	321 B	930 B	10.9 kB
IKE_AUTH exchange	9.6 kB	203 B	1142 B	11.0 kB
Est. SA pair and IKE-SA	9.6 kB	0 B	402 B	10.0 kB
Est. SA pair	9.6 kB	0 B	212 B	9.9 kB

Figure 5.7: The data in the table is based up the figures presented above. The second column is the size requirement of the image, the second and third is memory used by IKEv2 dynamically, and the fourth is the sum of the memory used by IKEv2 at the current phase. The sums are recomputed from the original, non-rounded, size measurements.

This reveals that the maximum memory usage of the IKEv2 service occurs during IKE_AUTH exchange and amounts to 11 kB. It should be mentioned that 203 of those bytes is used by the IKEv2 process's stack and are available for use by other processes' stacks, but subtracting the values in the column 'Total Maximum' from the total available RAM memory[3] gives the maximum amount of memory that other processes safely can use. Finally, another 190 bytes can be freed if the IKE session (IKE SA) is destroyed after creation of the two child SAs. This is currently not done and would require

[3]Which is 16 kB in the case of the Wismote platform.

a new set of IKE_SA_INIT, IKE_AUTH exchanges if the negotiation of new SAs would be required.

5.9 Source code complexity

A common way of measuring source code complexity is by the number of lines of source code (commonly abbreviated as SLOC). Counting the number of of lines of code; including comments, but excluding blank lines; the full software suite (IPsec + IKEv2) encompass 11764 lines of C source code. If comments are excluded, the number of lines becomes 5985.

Of those 11764 lines, 3801 lines (code as well as comments) are consumed by the support library ContikiECC (unused elliptic curves and features are not included in the figure) and the HMAC implementation. The various IPsec transforms (AES-CTR etc) and the supporting AES implementation are not included in this figure.

Out of the total 11764 lines, 7109 lines (approximately 3200 lines if excluding comments) is the exclusive work of the author. In addition to this, my work also consisted of: adaption of TinyECC (written for TinyOS originally) for Contiki, forming what I named 'ContikiECC'; key parts of the HMAC-SHA1-96 implementation was changed in such a way that it could make use of ContikiECC's SHA1-implementation in order to save memory; the implementation of IPsec's transforms were re-written in a number of ways in order to interface with the rest of the IPsec stack and improve clarity.

Chapter 6

Discussion

In this chapter I will discuss the results described above and the methodologies used in gathering them.

6.1 Evaluation method

In addition to demonstrating the usability of the implementation (i.e management of SAs), the primary concern of the evaluation was measuring ROM and RAM consumption as well as CPU requirements (i.e. latency).

6.1.1 IPsec

The evaluation demonstrated the automatic negotiation of SAs, a test which also covered the basic working of the SAD, the SPD and the rest of the IPsec system. The author believes that these result are easy to reproduce in other conditions with different network configurations and Contiki applications of different communication patterns. This is especially true of the IPsec system which encompass header generation and parsing (the μIP6 implementation), the SPD and the SAD, where the implementation should be complete and therefore have a high degree of interoperability.

6.1.2 IKEv2

The evaluation of the IKEv2 system only covered portions of what the standards documents considers mandatory for an implementation. The reason for this is twofold: the minimalistic implementation (i.e. lack of features to be tested) and the time required for configuring, testing and writing more test. The biggest downfall of this was the lack of testing of the traffic selector negotiation, whose algorithm is a simplification of that described in the standards documents. The author is not sure how it will function when nego-

tiating other SPD offers[1] with peers running other IKEv2 implementations than Strongswan.

6.1.3 ROM and RAM measurement

The evaluation of the image's ROM and RAM usage was straightforward and the numbers should be correct, caveat the following: the test application's (the UDP message 'bouncer') was included in all images of table 'ROM and RAM requirements' on page 49; no status, nor any error message strings were included in the binary, making in-field diagnostics difficult.

Stack and heap measurement figures should be correct to the byte by virtue of the evaluation techniques described in chapter 'Evaluation'.

6.1.4 Time

Time / latency testing was only performed on the Wismote platform as it was, arguably, the weakest platform that could comfortably accommodate the software. The rationale of that decision is that by ascertaining the functionality of the software on a simple platform, it can be assumed that it will work better on more powerful ones.

The time required for an IKEv2 handshake is heavily dependent upon the key length of the asymmetric key as it's completely dominated (more than 80%) by ContikiECC's large-number integer arithmetic.

6.2 Evaluation results

Previous publications[RDC+11] have shown that IPsec is feasible on Contiki. In chapter 'Evaluation' of this thesis I show that IPsec with IKEv2 for automated SA management is feasible, at least on the coming generation of 32 bit hardware. In *this section* I will present the reasoning behind the statement by discussing the software's performance on the current Wismote platform and improvements to be expected on future platforms.

With the purpose of making the discussion clearer I have elected to discuss future hardware in terms of a reference platform of my choice, namely STM32W from STMicroelectronics, which Contiki has been ported to. It's an ARM Cortex M3 based platform featuring an embedded IEEE 802.15.4 radio, up to 16 kB and 256 kB of RAM and flash memory (ROM), respectively. There are many reasons to expect hardware like this to become the mainstay of IoT nodes in the near future, but the chief one is that these MCUs are becoming more power-efficient[KKR+12] (in per-instruction as well as in sleep-mode power consumption) than the (in the world of IoT) hitherto incumbent line of 8- and 16-bit devices.

[1] Please remember that the SPD offer for different SPD policies is configured by the peer's administrator.

The investigation of the feasibility of IPsec/IKEv2 (or any other network protocol for that matter) on Contiki begins by determining the ROM and RAM requirements of the protocol's implementation. Subsequently, it must be asserted that there is enough resources left for the typical application using the service. This is not a problem on the Wismote platform, which has 6 kB of RAM and 55 kB of ROM to spare after the IPsec/IKEv2 system has established an SA pair (see section 5.8 'IKEv2 total RAM consumption at different times'). On the future reference platform, these margins will become plentiful.

However, feasibility is not achieved by simply fulfilling the memory requirements. It must be *usable* as well i.e. operate in its environment completing its intended functionality. For IPsec/IKEv2 this translates into the following criteria: are the *communication delays* with other peers tolerable (IKEv2); *energy usage* (both); can the *interoperability*[2] be deemed passable? I will herein only discuss IPsec in the context of interoperability, as the other aspects have been conclusively covered in [RDC+11], which features a very similar implementation (discussed in section 5.7 'Latency').

Interoperability is perhaps the hardest problem to tackle. Beginning with IPsec: interoperability should be good enough for communication with any system, with the exception of those that 1) have specific requirements concerning cryptographic standards 2) requires IPsec tunnel mode. I can see no reason as to why such requirements should be anything other than exceptions.

The IKEv2 service requires, just like the IPsec system, that the peer that it's talking to is configured to only use the features and configuration sets that it understands. I do not anticipate this to be a problem when the peer can be configured for the Contiki system's needs, but if this is not possible, the results are uncertain. Hence, I am of the opinion that the IKEv2 service is usable if the peers' administrators are willing to adapt the configuration to the needs of Contiki.

(At this point I would like to remind the reader that the aspect of administrative intervention which the previous subject touches upon is applicable to IKEv2 as well as IPsec, while the opposite, the case of non-intervention on the behalf of a human, is not applicable to IPsec as that always requires human action for setup.)

The energy usage of the IKEv2 implementation has not been explored, but can be approximated from the time required for running the computations as those vastly exceeds the energy required for the associated transmissions. The corollary of this is that the question of energy consumption

[2] In this context, interoperability is defined as the ability to communicate with other peers.

is determined by the algorithm used for the Diffie-Hellman exchange. As discussed in section 6.2.1 'Encryption algorithm improvements', large improvements can be made by employing optimizations.

Communication delays (the time required for to setup an SA pair with an hitherto not contacted client) amounts to slightly less than 11 seconds. This is definitely acceptable for most other IKEv2 peers in the sense that the delay won't cause[3] a timeout / retransmission, and is probably also enough for most application's demands given that the SA can be established prior to actual usage. In the future reference platform, the time required by the key exchange will be on the order of four times as fast if we assume a clock speed of 32 MHz. The speedup is a result of not only clock speed, but also of an increase in word length from 16 to 32 bit, which will reduce the multiple precision library's operations by half.

6.2.1 Encryption algorithm improvements

The ECC algorithms and IKEv2's management of cryptographic material are areas rife with opportunities for improvement. This section will briefly summarize what can be done to lessen times for handshakes and thus also decrease energy usage.

TinyECC features several optimization techniques which, if employed, would drastically reduces the time required for Diffe-Hellman key exchanges and digital signatures[4]. None of these methods have been ported to ContikiECC due to the time required to do so. They are all described in TinyECC's associated paper ([LN08]) that features a number of benchmarks which reveal that speedups of over eight times are achievable for DH exchanges[5]. Naturally, these improvements comes at a cost of increased ROM and RAM consumption since code complexity increases and pre-computations upon startup are made in some cases.

The IKEv2 service's cryptographic management can be improved in several aspects. One of these is the removal of the unnecessary computation of the public key, something which is currently done every time an IKE SA is negotiated.

The other possible improvement is the introduction of new ECP groups[6] that employs an optimization method called *Point Compression*. This tech-

[3] As an example, the default timeout for Strongswan's IKEv2 daemon *charon* is 120 seconds.

[4] N.B. Digital signatures, which might be used for IKEv2 authentication, are not used by the implementation presented in this work.

[5] Based upon the results displayed in figure 3 and figure 6 in [LN08]

[6] Elliptic Curve groups modulo a Prime

nique exploits the fact that all ECC[7] keys are two-dimensional points described with two integer coordinates. Given the fact that an ECC key is a point that always rests on a pre-defined curve (as specified by the RFC document defining the ECP group) and the coordinate system is cartesian, the missing coordinate can be trivially deduced by the other peer. This halves the size of the key, resulting in a saving of $2 * 192$ bits (48 bytes) (in the case of the 192-bit curve as used in this thesis) in message size (receiving the other peer's public key and transmitting one's own).

This begs the question: Why has this trivial optimization technique hitherto not been employed? This is widely attributed to the fact that the U.S. company Certicom holds several patents relating to ECC in general and ECC optimization techniques in particular. The point compression method, as described above, is covered by a U.S. patent [AMV00]. It can be argued that techniques such as this are trivial and obvious, and would thus render the patent null and void, but there are differing opinions of this in the community (e.g. [Berwn]). I believe that ECC has not been widely adopted as a direct result of this legal limbo.

6.3 IPsec and Contiki

IPsec is widely criticized for being too complex (e.g. [FS00]). Here follows two commonly cited examples of this: the AH protocol, which i elected to remove as I argued that its functionality was already provided by ESP (paragraph 4.3 'The AH protocol and its associated extension header is not implemented'); the fact that duplex connections requires two SAs, increasing memory overhead; the complexities incurred by using traffic flow as primitives in the security policy[8]. Naturally, this complexity puts the suitability of the IPsec/IKEv2 suite into question as it increases hardware requirements. Furthermore, the security provided by IPsec will often have to be complemented in the application layer as applications themselves often use other security primitives than than those of IPsec (which discriminates on hosts, transport protocols and the latter's properties such as ports). A good example of this is TLS commonly used by HTTP web services, which allow identification of the user.

This brings us to the next problem, namely that of network topologies and routing. The SPD, in its standard form as per RFC 4301, expects the network environment to be known at the time of SPD configuration, which was the common case of Internet architecture in the mid-90s when the standard was formulated. This presents a problem in environments that requires dynamic routing, such as the mesh networks[9] commonly used by IoT

[7] Elliptic Curve Cryptography

[8] Good examples of this are found on p. 42 in RFC 5996 which concerns Traffic Selector negotiation and p. 24 in RFC 4301 on the topic of SPD De-correlation.

[9] Mesh networks are networks typically composed of several network terminals that

systems. A protocol for forming such a mesh network is the RPL protocol (described in [WTB+12]) which is commonly used by Contiki. As there is often no knowledge of the network topology before RPL's 'handshake', the configuration of the SPD has to be postponed until then. This problem can be resolved by dynamically configuring the SPD on the basis of information derived from RPL, but no such scheme has been devised to my knowledge.

One possible solution to these concerns is to allow the application layer to configure SAs by means of 'IPsec channels' as described in RFC 5660 [Wil09]. Leaving some of the policy decisions to the application allows security based upon other primitives than described above, such as user-based ones. Still, such a framework would only increase the complexity of the implementation as it's essentially an extension to IPsec, and not a replacement of any internals.

6.4 Future Work and Recommendations

A good source of entropy needs to be found and exploited.

communicates wirelessly and doesn't have any preconception of the network's topology. They are said to form mesh networks when they, aided by a routing protocol such as RPL, create ad-hoc routing topologies with the intent of creating reliable network routes.

Chapter 7

Conclusion

Bibliography

[ABV+04] B. Aboba, L. Blunk, J. Vollbrecht, J. Carlson, and H. Levkowetz. Extensible Authentication Protocol (EAP). RFC 3748 (Proposed Standard), June 2004. Updated by RFC 5247.

[AD04] B. Aboba and W. Dixon. IPsec-Network Address Translation (NAT) Compatibility Requirements. RFC 3715 (Informational), March 2004.

[AMV00] G.B. Agnew, R.C. Mullin, and S.A. Vanstone. Elliptic curve encryption systems, October 31 2000. US Patent 6,141,420.

[Berwn] D.J. Bernstein. Irrelevant patents on elliptic-curve cryptography. `http://cr.yp.to/ecdh/patents.html`, Unknown. Accessed: 2014-02-03.

[DGV04] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, November 2004.

[DH95] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 1883 (Proposed Standard), December 1995. Obsoleted by RFC 2460.

[DH98] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), December 1998. Updated by RFCs 5095, 5722, 5871, 6437, 6564.

[DS05] Adam Dunkels and Oliver Schmidt. Protothreads-lightweight stackless threads in c. *SICS Research Report*, 2005.

[Dun03] Adam Dunkels. Full tcp/ip for 8-bit architectures. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 85–98. ACM, 2003.

[FGK03] S. Frankel, R. Glenn, and S. Kelly. The AES-CBC Cipher Algorithm and Its Use with IPsec. RFC 3602 (Proposed Standard), September 2003.

[FH03] S. Frankel and H. Herbert. The AES-XCBC-MAC-96 Algorithm and Its Use With IPsec. RFC 3566 (Proposed Standard), September 2003.

[FS00] Niels Ferguson and Bruce Schneier. A cryptographic evaluation of ipsec. Technical report, Counterpane Internet Security, Inc, 2000.

[GK98] R. Glenn and S. Kent. The NULL Encryption Algorithm and Its Use With IPsec. RFC 2410 (Proposed Standard), November 1998.

[GLVB+03] David Gay, Philip Levis, Robert Von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *Acm Sigplan Notices*, volume 38, pages 1–11. ACM, 2003.

[HC98] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409 (Proposed Standard), November 1998. Obsoleted by RFC 4306, updated by RFC 4109.

[Hou04] R. Housley. Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP). RFC 3686 (Proposed Standard), January 2004.

[JLN11] E. Jankiewicz, J. Loughney, and T. Narten. IPv6 Node Requirements. RFC 6434 (Informational), December 2011.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997. Updated by RFC 6151.

[Ken05a] S. Kent. IP Authentication Header. RFC 4302 (Proposed Standard), December 2005.

[Ken05b] S. Kent. IP Encapsulating Security Payload (ESP). RFC 4303 (Proposed Standard), December 2005.

[KHNE10] C. Kaufman, P. Hoffman, Y. Nir, and P. Eronen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 5996 (Proposed Standard), September 2010. Updated by RFC 5998.

[KKR+12] JeongGil Ko, Kevin Klues, Christian Richter, Wanja Hofer, Branislav Kusy, Michael Bruenig, Thomas Schmid, Qiang

Wang, Prabal Dutta, and Andreas Terzis. Low power or high performance? a tradeoff whose time has come (and nearly gone). In *Wireless Sensor Networks*, pages 98–114. Springer, 2012.

[KS05] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), December 2005. Updated by RFC 6040.

[LK08] M. Lepinski and S. Kent. Additional Diffie-Hellman Groups for Use with IETF Standards. RFC 5114 (Informational), January 2008.

[LMP+05] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[LN08] An Liu and Peng Ning. Tinyecc: A configurable library for elliptic curve cryptography in wireless sensor networks. In *Information Processing in Sensor Networks, 2008. IPSN'08. International Conference on*, pages 245–256. IEEE, 2008.

[Man07] V. Manral. Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH). RFC 4835 (Proposed Standard), April 2007.

[MG98] C. Madson and R. Glenn. The Use of HMAC-SHA-1-96 within ESP and AH. RFC 2404 (Proposed Standard), November 1998.

[Mil06] D. Mills. Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI. RFC 4330 (Informational), January 2006. Obsoleted by RFC 5905.

[MKHC07] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (Proposed Standard), September 2007. Updated by RFCs 6282, 6775.

[MSST98] D. Maughan, M. Schertler, M. Schneider, and J. Turner. Internet Security Association and Key Management Protocol (ISAKMP). RFC 2408 (Proposed Standard), November 1998. Obsoleted by RFC 4306.

[ÖDE+06] Fredrik Österlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-level sensor network simulation with cooja. In *Proceedings of the First IEEE International*

Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006), Tampa, Florida, USA, November 2006.

[PA98] R. Pereira and R. Adams. The ESP CBC-Mode Cipher Algorithms. RFC 2451 (Proposed Standard), November 1998.

[Pip98] D. Piper. The Internet IP Security Domain of Interpretation for ISAKMP. RFC 2407 (Proposed Standard), November 1998. Obsoleted by RFC 4306.

[RDC+11] Shahid Raza, Simon Duquennoy, Tony Chung, Dogan Yazar, Thiemo Voigt, and Utz Roedig. Securing Communication in 6LoWPAN with Compressed IPsec. In *Proceedings of the International Conference on Distributed Computing in Sensor Systems (IEEE DCOSS 2011)*, Barcelona, Spain, June 2011.

[Ric05] M. Richardson. A Method for Storing IPsec Keying Material in DNS. RFC 4025 (Proposed Standard), March 2005.

[TED10] Nicolas Tsiftes, Joakim Eriksson, and Adam Dunkels. Low-power wireless ipv6 routing with contikirpl. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '10, pages 406–407, New York, NY, USA, 2010. ACM.

[VD10] Jean-Philippe Vasseur and Adam Dunkels. *Interconnecting Smart Objects with IP - The Next Internet*. Morgan Kaufmann, 2010.

[Wik13a] Wikipedia. Duff's device — wikipedia, the free encyclopedia, 2013. [Online; accessed 19-July-2013].

[Wik13b] Wikipedia. Ipv6 packet — wikipedia, the free encyclopedia, 2013. [Online; accessed 22-April-2013].

[Wik13c] Wikipedia. Message authentication code — wikipedia, the free encyclopedia, 2013. [Online; accessed 8-May-2013].

[Wik13d] Wiktionary. transformation — wiktionary, the free dictionary, 2013. [Online; accessed 27-July-2013].

[Wil09] N. Williams. IPsec Channels: Connection Latching. RFC 5660 (Proposed Standard), October 2009.

[WTB+12] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, JP. Vasseur, and R. Alexander. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC 6550 (Proposed Standard), March 2012.