JACOBS
UNIVERSITY

Security in IPv6-enabled Wireless Sensor Networks: An Implementation of TLS/DTLS for the Contiki Operating System

by

Vladislav Perelman

A thesis for conferral of a Master of Science in Computer Science

Prof. J. Schönwälder
Name and title of first reviewer

Dr. H. Stamerjohanns
Name and title of second reviewer

Date of Submission: June 29, 2012

Jacobs University — School of Engineering and Science

Declaration

I, Vladislav Perelman, born 1989-10-02, hereby declare that this thesis is the result of my independent work, has not been previously accepted in substance for any degree and is not concurrently submitted for any degree.

This thesis is being submitted in fulfillment of the requirements for the degree of Master of Science in Computer Science.

.....

Place, Date Vladislav Perelman

Acknowledgments

I would like to sincerely thank my friends and family for supporting me during my studies. Special thanks to Anuj Sehgal for always willing to help me out whenever Contiki would be too stubborn and, of course, big thanks to my supervisor professor Jürgen Schönwälder for providing constant guidance and support throughout my entire two years of graduate school.

Abstract

During the last several years the advancements in technology made it possible for small sensor nodes to communicate wirelessly with the rest of the Internet. With this achievement the question of securing such IP-enabled Wireless Sensor Networks (IP-WSNs) emerged and has been an important research topic since. In this thesis we discuss our implementation of TLS and DTLS protocols using a pre-shared key cipher suite (TLS_PSK_WITH_-AES_128_CCM_8) for the Contiki operating system. Apart from simply adding a new protocol to the set of protocols supported by the Contiki OS, this project allows us to evaluate how suitable the transport-layer security and pre-shared key management schemes are for IP-WSNs.

Contents

1 Introduction **1**

2 Background **3**
2.1 Wireless Sensor Networks . . . 3
2.2 IEEE802.15.4 . . . 5
2.3 IPv6-enabled WSN . . . 6
2.3.1 6LoWPAN . . . 7
2.4 Security . . . 8
2.4.1 SSL/TLS . . . 9
2.4.2 DTLS . . . 10
2.4.3 IPsec . . . 12
2.4.4 Key Management . . . 13

3 Related Work **15**
3.1 Key Management in WSN . . . 15
3.2 Link-layer Security . . . 17
3.3 Network-layer Security . . . 18
3.4 Transport-layer Security . . . 21
3.5 Application Layer Security . . . 22

4 Implementation **23**
4.1 AVR-Raven Platform . . . 23
4.2 Contiki Operating System . . . 24
4.3 Network Setup . . . 24
4.4 Design Principles . . . 25
4.5 API and General Overview . . . 26
4.6 Implementation Details . . . 28
4.6.1 Structure . . . 29
4.6.2 Input Parsing . . . 29
4.6.3 Output generation . . . 31
4.6.4 Data Hashing . . . 31
4.6.5 Encrypting and Decrypting of Messages . . . 32
4.7 Data Structures . . . 33

4.8 Additional Notes 34
4.8.1 Dynamic Memory Allocation 34
4.8.2 PSK 35
4.8.3 Tuning Parameters 36

5 Evaluation and Testing **37**
5.1 Interoperability 37
5.2 Memory Usage 38
5.3 Timing Evaluation 39

6 Conclusions **42**

A Appendix **44**
A.1 Source Code 44
A.2 Using TLS/DTLS 44

Chapter 1

Introduction

Networks are all around us – they connect millions of devices, they let people communicate while being across the world from each other, they became an important and ubiquitous part of our everyday lives. With the latest advancements in technology more and more devices become capable of using the network and they do not have to be powerful computers – a simple light-switch or a thermostat can be connected to the Internet. This tendency leads to the development of the IP-enabled Wireless Sensor Networks (IP-WSNs), or what some call The Internet of Things (IoT) – a network of uniquely identifiable objects (be it a door lock or your car keys) that can be accessed from anywhere via the Internet and that can be used for making the lives of their owners better in one way or the other.

Of course, while some people think of how to make use of the IoT for the better, there are always those, who will try to misuse it either in order to gain access to private information of others or for other adversary purposes. While turning someone's alarm clock off can be seen as a harmless prank, the results of feeding false information to the network of a (nuclear) power plant or a power grid can be catastrophic. This is why the question of securing communication in the IoT is of highest importance and has to be carefully considered.

A lot of research already has been done on this topic and yet some of the questions regarding which cipher suites and which key management mechanisms fit certain criteria best still remain unanswered. Probably the only way to answer those questions is to try to implement various security protocols and evaluate their performance.

Transport Layer Security (TLS) is a very popular protocol that is used heavily in the Internet. Implementations for the Tiny OS operating system already exist, however, the Contiki OS, which is quickly gaining popularity

over the the last couple of years, still lacks support for TLS. Therefore implementing TLS for the motes running on the Contiki OS would be of great benefit to community building IP-enabled WSNs.

It is worth noting that TLS relies on the connection-oriented Transport Control Protocol (TCP) as its transport mechanism while many applications that are used in the WSNs run over the connection-less User Datagram Protocol (UDP), hence implementing the Datagram Transport Layer Security (DTLS) would also be of high value. Taking these ideas into consideration, we have decided to implement both, the TLS and DTLS protocols, for the Contiki OS and to evaluate their performance to find out whether these security mechanisms are appropriate for WSNs giving the memory constraints of involved devices.

To our knowledge this is the first freely available implementation of TLS for Contiki and the most complete implementation of DTLS.

The rest of this thesis is structured as follows. Section 2 provides some background information about the topic. Section 3 gives an overview of the state of the art in the field of securing the IP-WSNs. Implementation details follow in section 4 with evaluation of the implementation in section 5. We will give our conclusive remarks in section 6.

Chapter 2

Background

2.1 Wireless Sensor Networks

Over the last ten years a lot of researchers have been focusing close attention to the development of the so-called Wireless Sensor Networks (WSNs). The growth of WSNs became possible due to the significant advancements in the areas of hardware manufacturing, electronics, wireless communications and efficient software algorithms, which in turn led to the appearance of low-cost, low-power, multifunctional sensor nodes that could communicate with each other wirelessly over relatively short distances. A WSN is a network that consists of such nodes (also called "motes"). Nodes are typically spatially distributed and there can be anywhere between a few of them to several hundreds and even thousands of them in one network. As the name suggests a sensor node can contain various sensors for monitoring physical or environmental conditions, such as temperature, sound, pressure, radiation level, etc. Apart from these sensors, motes also have a radio transceiver, a microcontroller and some sort of an energy source (typically a battery).

There are numerous ways in which WSNs can be of great benefit for their users. Nodes can monitor energy consumption of various devices which then can be sent to some server for later use [1], they can also keep track of seismic activity in regions with high potential danger risks or radiation level in the areas located close to the power plants. Akyildiz, et al. [2] have classified possible range of applications as military, health-care, environmental, home applications and other commercial usages.

Sensor networks are usually deployed together with one or several base stations which act as gateways between the sensor network and the external world. Base stations are normally computers or other powerful devices and have either IEEE 802.11 or IEEE 802.3 connectivity. An example of typical

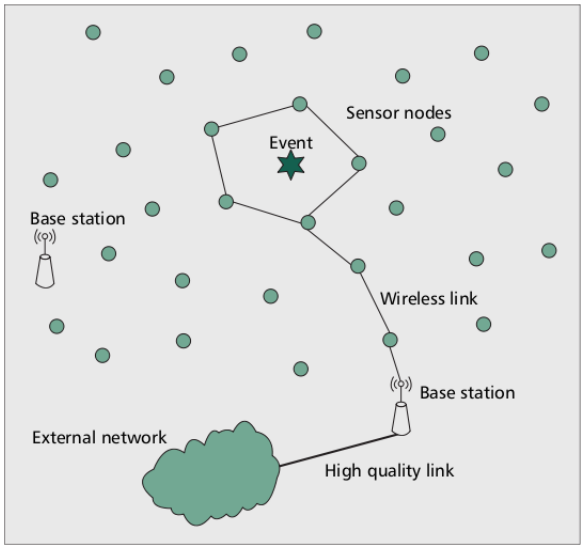

Figure 2.1: An example WSN communication

communication between nodes and external network via the base station is illustrated in Figure 2.1 [3].

Having large number of nodes in one network results in the necessity of making these nodes as cheap (and hence as small) as possible to reduce the overall cost of the network. One obvious advantage of having such nodes is ease of deployment as well as their low replacement costs. However, low cost of a node also implies having very limited processing and data storage resources. For example one of the AVR-Raven motes has 16kB of RAM, 128kB of Flash memory and an 8-bit ATmega1284p CPU. As a rule of thumb, constrained devices are classified into three categories based on the amount of resources available on the devices.

- Class 0: about 1kB of data and 10kB of code (too constrained).
- Class 1: about 10kB of data and 100kB of code (quite constrained).
- Class 2: about 50kB of data and 250kB of code (not so constrained).

As can be seen most of the devices are not powerful enough to run complex algorithms which require high processing power or store a lot of data. Another disadvantage of nodes in a WSN against a normal computer is the lack of a stable power source. In most cases, nodes, which are deployed in hazardous or hostile environment, are powered by one or two batteries and the lifespan of these batteries is what limits the lifespan of the node, since replacing the batteries is often impossible.

2.2 IEEE802.15.4

As mentioned above, nodes in the WSNs have radio transceivers for wireless communication. IEEE802.15.4 [4] (802.15.4 later in this paper) is an IEEE standard that has been designed for low-power and low-speed wireless personal area networks (WPANs), for which it provides specifications of physical and media access control layers. This specification has gained popularity and became the core standard for wireless communication in WSNs. Large boost in the usage of 802.15.4 was due to its integration into the popular industrial standards such as the ZigBee protocol stack [5].

The original specification has been developed in 2003 and introduced two physical layers based on the direct sequence spread spectrum (DSSS) techniques – one operating with the transfer rate of 20 and 40 kbit/s and another one working with a rate of 250 kbit/s. In 2006, 2007 and 2009 the standard was revised improving the data rates as well as other characteristics. The 2003 specification also defined two classes of devices that use 802.15.4 – Full Function Device (FFD) and Reduced Function Device. The former, as the name suggests, implements a complete protocol stack and is capable of being a PAN coordinator and talking to any other node on the network. According to [4] a PAN coordinator is the principal controller of the PAN and it provides synchronization services through the transmission of beacons (discussed later). RFD on the contrary has a very simple implementation and is limited to being a leaf node in complex topologies. The two basic topologies that are available for an 802.15.4 network are star topology where all nodes communicate via the central PAN coordinator and peer-to-peer topology where nodes can also communicate directly with each other. All other topologies are combinations and extensions of the two basic ones.

802.15.4 specifies two modes of operation in the MAC layer: slotted and unslotted. In the latter one CSMA/CA is used by the sender, while the receiver is listening continuously and sending ACKs back if requested. In the slotted mode the PAN coordinator (each PAN has exactly one coordinator) sends special beacon frames that delimit so-called superframes. Each superframe consists of three periods – in the first one senders use normal CSMA/CA, in the second one each sender has a certain, assigned by the PAN coordinator, time slot during which it can send data, and in the third period the channel is not used giving all nodes a chance to sleep. Apart from the beacon frames that were mentioned above, 802.15.4 introduces three more frame formats – Command frames (data and beacon requests, conflict notification, etc.), Data frames (carrying user data) and Acknowledgement frames (acknowledge successful data transmission). The maximum size of each frame is 127 octets, while the maximum size of the frame header is 25 octets. The general frame format can be seen in Figure 2.2.

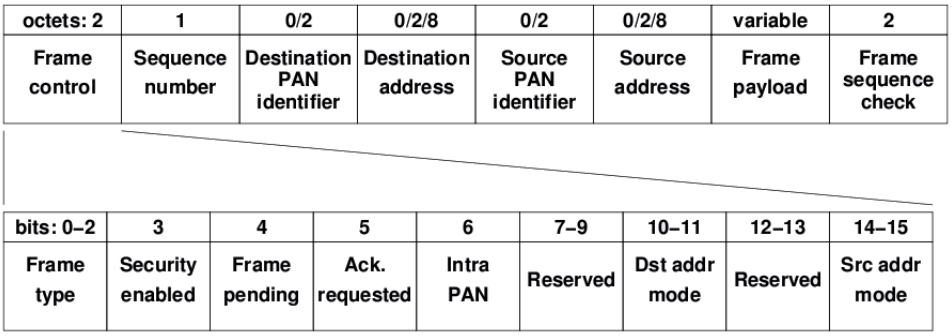

Figure 2.2: General frame format of the 802.15.4 frame

By default 802.15.4 communication does not provide any security, however, the specification describes the usage of the AES protocol either for encryption only (in CTR mode), authentication only (32-, 64- or 128-bit MAC in CBC mode) or both encryption and authentication (in CCM-mode). Clearly, added security comes with the price of using extra bytes from the already small data fragments. For example, AES-CCM-128 mode adds 21 octets to the 802.15.4 MAC header making it take up to 46 octets leaving only 81 octets for higher layers data. Also the specification does not describe key management mechanisms, leaving this task to be provided by higher layers.

2.3 IPv6-enabled WSN

When the WSNs were originally developed they were imagined as a separate infrastructure that would be responsible for gathering information and sending it to the base station, which would act as a gateway between the WSN and the rest of the world. However, over the time the role of the node within a WSN has been rethought to move away from the mere sensing device to that of an active agent in the Internet, which could be accessed from anywhere and anything using IP. Connecting WSNs and the IP world has numerous advantages among which are [6]:

- IP is well-known, wide-spread and is proven to be working. Every device in the modern Internet uses IP which implies all of them will be able to communicate with nodes within WSNs if needed.
- IP is independent of the used physical and MAC layer protocol, hence it can be run on top of 802.15.4 without modification of the latter.
- All of the protocols and applications running on top of IP will be technically available for use in WSNs. Whether their use will be possible given the constrains of the low-power devices as well as the 802.15.4

	802.15.4 Network	*Typical IPv6 Network*
Packet Size	Maximum of 127 octets	Maximum Transmission Unit (MTU) at least 1280 octets
Bandwidth	Typically 250 Kbps	54Mbps (802.11g) / 100 Mbps (Ethernet)
Addressing	16-bit short or IEEE 64-bit extended MAC addresses	128-bit IPv6 addresses
Power	Low, most devices run on battery.	No constraint, most devices are connected to a power network.

Table 2.1: Comparison between an 802.15.4 and a typical IPv6 network

frame format is a question which requires separate discussion for each of the protocols and applications.

- Tools for diagnostics, management, and commissioning of IP networks already exist.

The above arguments justify usage of IP on sensor nodes and Hui et al. answers the question whether IPv6 or IPv4 should be used by claiming that "IPv6 is better suited to the needs of WSNs than IPv4 in every dimension" [7]. They state that the IPv6 network architecture allows utilization of various mechanisms that have become wide-spread in the WSNs: sampled-listening, hop-by-hop feedback, etc. Moreover they believe that IPv6 allows for a more efficient implementation than IPv4 as well as better compression of addresses is possible compared to IPv4.

Clearly, when integrating IPv6 and WSNs one faces numerous challenges that result from the differences in the general structures of typical 802.15.4 and IPv6 networks. A high level comparison between these networks is presented in Table 2.1 [8]. In the next section we discuss the 6LoWPAN standard that has been developed to overcome these challenges.

2.3.1 6LoWPAN

The IPv6 over Low power WPAN (6lowpan) working group at the IETF was established to tackle the problems of integrating IP with the low-power, low-cost WSNs. As a result of this group's work the specification of an adaptation layer allowing to transport IPv6 packets over 802.15.4 links was introduced in RFC4944 [9].

The specification states that IPv6 packets must be carried within the 802.15.4 data frames (discussed earlier) and that the unslotted mode of operation in

the MAC layer should be used. It defines how the fragmentation and reassembly of the IPv6 packets should be done as well as the compression of the IPv6 header. The compression was recently updated by the RFC6282 [10].

According to the specification all LoWPAN encapsulated datagrams transported over 802.15.4 links are to be prefixed by an encapsulation header stack, where each header contains a header type followed by zero or more header fields. The header type is defined by the dispatch code, which is the first byte of each header. For example `01000001` byte specifies that the following header is an uncompressed IPv6 header, or `01111111` byte implies usage of the IPv6 Header Compression defined in [10] (this is true only as of September 2011 when the RFC6282 was published).

The original specification described two stateless header compression mechanisms LOWPAN_HC1 and LOWPAN_HC2, which worked based on the principle of omitting all fields that could be calculated from the context. One big disadvantage of those mechanisms was their limited value when it came to using routable addresses in which case LOWPAN_HC1 required full IPv6 source and destination addresses to be sent in-line. To improve this shortcomings the specification was updated by the RFC6282 which defined a new encoding format – LOWPAN_IPHC for effective compression of unique local, global and multicast IPv6 addresses based on shared sate within contexts. In addition the document also defines LOWPAN_NHC – an encoding format for arbitrary next headers. In the best case scenario of the link-local unicast communication over UDP both IPv6 and UDP headers can be compressed down to a total of 4 octets.

As mentioned above the 6LoWPAN layer also defines fragmentation and reassembly of the IPv6 packets in case they do not fit in the small payload of the data fragments of 802.15.4. In that case the first fragment carries a header that has a dispatch code starting with `11000` which is followed by an 11-bit datagram size and a 16-bit datagram tag. All subsequent fragments carry a header that has a dispatch code starting with `11100`, followed by the size, the tag and an additional 8-bit datagram offset.

Several implementations of 6LoWPAN standard have been developed and are currently being maintained, which among others include implementations for TinyOS and for Contiki OS – both popular operating systems specifically designed for constrained devices.

2.4 Security

If there is one thing that everyone in the Internet community agrees on, then that is that the question of providing security is highly important and needs

to be well thought of in every protocol and application available. When talking about WSN these are the main services that should be achieved to provide reasonable security:

- *Confidentiality* – All important information that is being transmitted between communicating parties remains unknown for everyone else. This is usually achieved by encrypting the data such that even if a third-party eavesdropped on all the packets transmitted it would not be able to access the data. What kind of information needs to be protected is dependent on the application.
- *Authentication* – Allows communicating parties to ensure the identity of each other. This way an attacker cannot claim to be someone else to gain access to important data or to spread invalid information. Authentication is usually achieved by sending a message authentication code (MAC) together with the message.
- *Integrity* – Ensures that the message was received in the exact same way it was sent, not in any way altered in transit by an adversary. A simple cyclic redundancy checksum (CRC) that is used to detect random errors during packet transmission or a MAC can provide integrity.
- *Availability* – Indicates that the nodes in WSN will provide service whenever required despite possible Denial of Service (DoS) attacks. Achieving this, however, is extremely hard since DoS attacks can be launched at any layer of the WSN and can heavily reduce the network performance by jamming the radio channel, exhausting the power supplies of the nodes or using various other methods.
- *Authorization* – Ensures that only authorized nodes can access services and resources on the network.

In the next part of the section a brief introduction to several important and widely used security protocols will be given.

2.4.1 SSL/TLS

One of the most common encryption protocols that is nowadays used for secure web browsing, e-mail, VoIP and other application is Transport Layer Security protocol, or TLS [11]. TLS is based on its predecessor – Secure Sockets Layer (SSL), which was developed by Netscape Communications.

TLS is a layered protocol and comprises several other protocols as shown in Figure 2.4.1. It uses the Record Protocol to fragment the data into manageable blocks, optionally compress the data, apply a MAC, encrypt

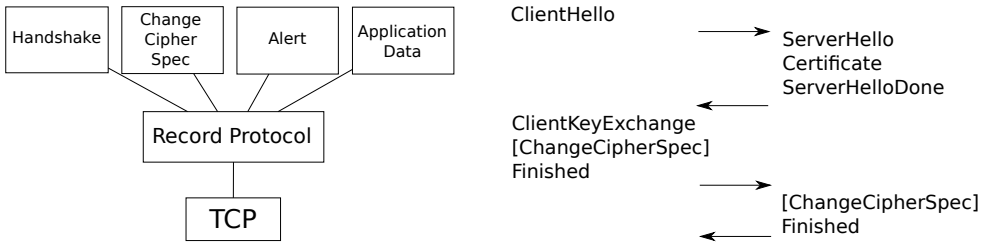

Figure 2.3: left: Structure of the TLS protocol, right: simple TLS handshake

and transmit the result in the blocks which are called records. The Record Protocols uses a reliable transport, such as TCP and can encapsulate data from four other protocols:

- *Handshake protocol* – used for negotiating which encryption and compression algorithms to use during secure communication, for authentication of the involved nodes using their certificates (optional) and for establishing keys needed for encryption. A simple TLS Handshake can be seen in Figure 2.4.1.
- *ChangeCipherSpec protocol* – a very simple protocol that has only one message in it that is sent during the Handshake protocol. This message tells the recipient that all messages that follow will be encrypted and compressed using the algorithms and keys which were just negotiated using the Handshake protocol.
- *Alert protocol* – used for notifying the opposite side about a deviation from the normal execution of the TLS protocol that requires in most cases either closing the established connection or starting the rekeying process. An alert message could be of type error or warning depending on the severity level of the situation.
- *Application Data protocol* – transports the encrypted application data that is delivered after decryption to the upper layers.

2.4.2 DTLS

There is a large selection of applications that use UDP rather than TCP as their underlying transport protocol. These applications are normally delay sensitive and therefore do not need retransmission and reliable delivery that TCP provides. Such applications include real-time audio and video conferences, online games, IP telephony and so on. All of them still need to be protected, however, since they may carry important user data, therefore an equivalent of TLS is required for securing datagram based applications, and Datagram Transport Layer Security (DTLS) is exactly that. DTLS was

meant to be a datagram capable TLS, hence its design was based on that of TLS making two protocols very similar to each other.

DTLS reuses almost all protocol elements of TLS but introduces a number of important modifications in order to overcome the unreliability faced when using UDP as the transport protocol.

Just like in TLS, the data sent in DTLS is split into blocks called records, however, the DTLS Record header format introduces two extra fields – `epoch` and `sequence_number` – which are absent from the TLS Records. The explicit sequence numbering system was included in the DTLS Records in order to accommodate lost or delivered out of order records. Since records in TLS are guaranteed to arrive in the correct order the numbering of records in TLS is implicit. The epoch number is used by the end-points to specify which cipher state is being used to protect the record payload, which is again one of the means of dealing with lost records and records arrived out of order, especially during the handshake procedure. The DTLS Record format is shown in Figure 2.4, where the boxed fields are DTLS specific.

To mitigate possible DoS attacks that TLS is not vulnerable to, such as the resource consumption attack, DTLS adopts the cookie exchange technique. Upon receiving a Client Hello message server does not start a normal handshake procedure but rather responds with a stateless cookie that the client must replay as a sign that it is willing to communicate with the server. Once the server receives and verifies the cookie it continues with the regular handshake.

Applications that use UDP have to be able to deal with the possible lost messages and messages received out of order. Therefore once the DTLS handshake is complete and the Application Data is sent, unreliability of UDP is not a problem for DTLS. However, during the handshake DTLS protocol has to make sure that all the messages do get delivered because otherwise the client and the server may deadlock, waiting for the next messages in the handshake procedure. To avoid that, a retransmission mechanism had to be introduced. Each end-point has a timer and keeps retransmitting previously sent message every time the timer expires until the next expected message is received.

The last major modification that is included in DTLS is the format of the Handshake message. Just like in TLS, the DTLS records do not preserve boundaries of underlying messages, but due to the nature of UDP transport the fragments of the Handshake message which was split over several records could be received in the wrong order. To cope with that, the Handshake format had to be modified to include explicit sequence numbering as well as the offset and the length of the fragment sent. The new Handshake format can be seen in Figure 2.4, where the boxed fields are DTLS specific.

```
struct {
    ContentType         type;
    ProtocolVersion     version;
    uint16              epoch;
    uint48              sequence_number;
    uint16              length;
    opaque              payload[length];
} DTLSRecord
```

```
sruct {
    HandshakeType msg_type;
    uint24  length;
    uint16  message_seq;
    uint24  frag_offset;
    uint24  frag_length;
    HandshakeMessage msg_frag[frag_length];
} Handshake;
```

Figure 2.4: left: DTLS Record format, right: DTLS Handshake format. Boxed fields are DTLS specific and are absent in the TLS protocol.

2.4.3 IPsec

Internet Protocol Security (IPsec) [12] is a protocol suite for securing IP communication on an end-to-end basis by encrypting and authenticating each IP packet of a communication session. Unlike other security protocols such as TLS or SSH, which operate above the transport layer of the TCP/IP model, IPsec operates in the network layer and hence protects any application traffic that goes through the network without the need for modifying applications or specifically integrating IPsec into them.

IPsec makes use of two traffic security protocols to provide most of the security services – the Authentication Header (AH) [13] and the Encapsulating Security Protocol (ESP) [14]. The AH protocol provides integrity and data origin authentication with optional anti-reply features while ESP also offers confidentiality. Since in most cases ESP can provide the required security services it is a mandatory protocol to implement when implementing IPsec, while the AH has been made an optional protocol to implement. Both protocols can be used in two modes – transport and tunnel. In transport mode the IPsec header is inserted after the IP header and the protocols provide protection primarily for next layer protocols since only the payload of the original IP packet is encrypted and/or authenticated. In the tunnel mode a complete IP packet is encrypted and/or authenticated and then encapsulated into a new IP packet with a new IP header.

One of the fundamental concepts in IPsec is the concept of Security Associations (SA). An SA is a simplex association between two IPsec endpoints that provides security services to the traffic carried by it. In other words an SA is a collection of algorithms and other security parameters (such as keys) that is being used for protecting the flow of information in one direction, which implies that for a typical bi-directional communication a pair of SAs is required. Both ESP and AH protocols make use of SAs and are required to support this concept. SAs could be established and maintained manually, however that does not scale and hence is not recommended. Instead another component of IPsec, namely the Internet Key Exchange protocol (IKE or IKEv2) is used [15].

IKE performs mutual authentication between two parties and establishes an IKE security association that includes information, which is used to efficiently establish SAs for ESP or AH. All IKE communications consist of pairs of messages which are called "exchanges" or "request/response pairs". An IKE session starts with the IKE_SA_INIT exchange that negotiates security parameters for the IKE SA, sends nonces and Deffie-Hellman values. The second exchange, IKE_AUTH, transmits identities, authenticates them and sets up the SA for the AH or ESP Child SA (another name for the SA which is being set up for usage by AH or ESP). In the common case a total of 4 messages should be exchanged to establish IKE SA and the first Child SA.

IKE is one of the protocols that is responsible for key management – an essential part of any cryptographic protocol since all encryption and authentication operations must involve keys. A brief introduction to key management basics is given next.

2.4.4 Key Management

Encryption protocols that provide confidentiality have to use some keys in order to transform plaintext messages into ciphertext; packet authenticity can be achieved by adding a MAC to the message that is computed by using a hash function that hashes the concatenation of the message and a key.

There are two types of keys that are being used in the cryptographic systems – symmetric and asymmetric. When a symmetric key is used, both sender and receiver share the same key that is known only to the two of them and no one else. The sender encrypts the message M with the known key K to produce the ciphertext $C = E(M, K)$ which is then sent to the receiver that uses the decryption algorithm to retrieve the original message $M = D(C, K)$. Examples of popular symmetric algorithms are Advanced Encryption Standard (AES) [16] and Rivest Cipher 5 (RC5) [17] that make use of simple hash, rotation and scrambling operations that can be efficiently implemented in software or hardware.

In the asymmetric cryptography system, each node has a pair of keys (K_s, K_p) which are called private and public keys (hence another name – public-key cryptography). The private key is kept secret while the public key is published. If the sender wants to send a message M to the receiver it uses the receiver's public key K_p to get the ciphertext $C = E(M, K_p)$, which is then sent and the only one who can decrypt it is the receiver by using his private key: $M = D(C, K_s)$. Such algorithms can be computationally very expensive since they require dealing with very large prime numbers and exponents of high degree. Common asymmetric algorithms are Deffie-

Hellman (DH) [18] (that is used for establishing a shared secret) and the RSA algorithm [19].

In both symmetric and asymmetric systems the security of the whole system relies mainly on the secrecy of the keys involved, if the key is compromised then an adversary can use it to spread false information in the system hence the whole system becomes broken. This results in a very important issue of key establishment, especially for the symmetric systems where two parties need to agree on one key without letting others know what this key is. The problem becomes much easier in the public-key cryptography since there each node can keep secret its own private key and publish the public key. This ease of key establishment comes with the price of being much more computationally expensive.

Chapter 3

Related Work

An extensive amount of work has already been done in the area of securing the wireless sensor networks, however connecting the WSNs and the IP world brought new challenges and hence more research questions. An internet draft [20] discussing security in 6LoWPANs was presented last year, however its authors only analyzed the threats and challenges that this topic affords and did not provide any solutions to the found problems. In this section the most recent advances in the area of IP-enabled WSNs will be presented.

3.1 Key Management in WSN

For a while public-key cryptography was considered too heavy-weight for sensor networks and hence a lot of research was dedicated towards finding better ways of establishing the secret key which could then be used by symmetric key algorithms. There exist several surveys [21–23] that in details discuss various key distribution techniques as well as other aspects of key management and security in WSNs in general.

Most solutions to key establishment in WSNs involve the so-called pre-distribution approach, in which every node in the network is preloaded with certain key material that is used during the key establishment process. Since the goal is for every node to be able to talk to any other node the easiest solution would be to preload each node with $N - 1$ keys – one for each other node in the network. Clearly this approach though being effective does not scale well with the growth of the network due to a large memory cost. To cope with the memory constraint the partial pre-distribution could be used in which some nodes can establish shared keys directly and then help establish indirect shared keys between other nodes.

A typical scheme that follows such approach is the random key pre-distribution

(RKP) introduced by Eschenauer and Gligor in [24]. In their scheme each node is pre-loaded with a subset of keys, called a key ring, which is randomly selected from a global pool of keys. This results in a large probability that two neighbor nodes will have a shared key. Based on RKP other schemes have been developed that improved certain characteristics of the network [25–27]. The main problem with all of the random key predistribution approaches is that they leave a small chance that two nodes will not be able to establish a key and hence will not be able to communicate, which is not acceptable. To solve that problem deterministic approaches were developed.

Chan and Perrig introduce Peer Intermediaries for Key Establishment (PIKE) [28], a class of protocols where some nodes act as a trusted intermediary to perform key establishment between neighbors. Another examples of deterministic key distribution schemes are mGKE (a Group-based Key Establishment scheme for mobile sensor networks) by Zhou, et al. [29], IOS (ID-based one-way function scheme) by Lee and Stinson [30], schemes by Delgosha and Fekri [31, 32] and many others.

All of the above schemes were developed to solve the problem of distributing the secret keys between pairs of nodes that want to communicate. As mentioned earlier, using public-key cryptography would solve the problem by removing the need for such distribution in the first place. Originally it was thought that using PKC would not be feasible due the high computational cost of the operations involved, however the development of Elliptic Curve Cryptography has proven otherwise.

ECC is a public-key cryptography approach that is based on the algebraic structure of elliptic curves over finite fields and was proposed independently by Victor Miller [33] and Neal Koblitz [34] Like any other type of public-key cryptography ECC is based on a very hard, intractable to solve mathematical problem – the harder it is to solve it the more secure the algorithm. In case of the well-known RSA it is the Integer Factorization problem, in case of ECC it is the elliptic curve discrete logarithmic problem (ECDLP), and unlike the former problem that has a sub-exponential solution, solution to ECDLP is fully exponential. This implies that ECC can offer the same level of security as RSA but while using much smaller key sizes. As shown in [35] a 160-bit ECC key provides the same level of security as a 1024-bit RSA key, and 224-bit ECC is equivalent to the 2048-bit RSA. Smaller keys result in faster computations, less memory consumption as well as power consumption and bandwidth savings which makes ECC a feasible choice for using on constrained devices. According to various evaluations presented in [36, 37], using ECC results in approximately 1.5 times smaller memory consumption on the motes and in 4 to 5 times faster cryptography operations when compared to using RSA.

The Elliptic Curve Diffie Hellman (ECDH) [38] and the Elliptic Curve Digital Signature Algorithm (ECDSA) [39] are the Elliptic Curve counterparts of the DH and DSA respectively. There have been several implementations of these algorithms for use in WSNs [40, 41], including a widely-used TinyECC [42] – a portable and efficient library developed at the North Carolina State University. TinyECC as well as many other implementations was developed specifically for TinyOS and was written in nesC. So far no such library that would target the Contiki OS has been developed, however attempts to port existing cryptographic libraries to Contiki to prove feasibility of using ECC PKC were made [43].

In the rest of this section an overview of the most recently proposed security mechanisms in the IP-enabled WSNs will be presented, grouped by the layers in which the mechanisms are operating.

3.2 Link-layer Security

The first fully implemented link-layer security suite for WSNs was TinySec [44]. Written in nesC it was specifically designed for use with TinyOS and was incorporated into the official TinyOS release. Authors motivated introducing security into the link layer by noting that the general communication pattern in the WSN was many-to-one when many nodes send similar information to the base station. In order to reduce the number of sent messages (and therefore increase the lifetime of each node) in-network processing such as aggregation and duplicate elimination is often used which requires nodes in the network to access and modify the sent messages. Having security in higher layers would not allow such message manipulations. On the other side having link-layer security implies not being able to provide end-to-end security which would be advantageous when the WSN is connected to the Internet.

TinySec supports two modes of operation – authenticated encryption (TinySec-AE) and authentication only (TinySec-Auth). In the former the data payload is encrypted using a Skipjack [45] block cipher and then authenticated using a CBC-MAC which is computed over the encrypted payload and the packet header. In the authentication only mode the entire packet is authenticated but the data is not encrypted. The overall implementation of TinySec requires 256 bytes of RAM and 8152 bytes of ROM. As the authors mention, TinySec generally has a lot in common with the link-layer architecture that 802.15.4 provides. The major differences are in the choice of encryption algorithms used and their modes of operation. Also devices that support 802.15.4 must be able to perform the encryption operations in hardware, while TinySec performs cryptography in software.

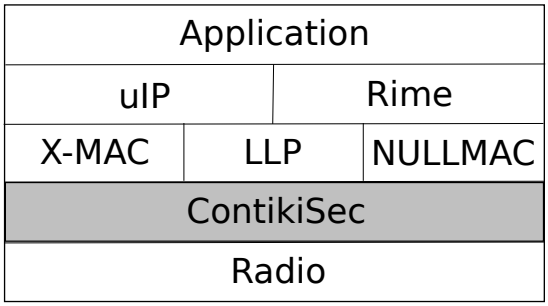

Figure 3.1: The Contiki OS stack with ContikiSec

In 2004 an open source operating system for embedded devices called Contiki OS was introduced. Contiki has become quite popular over the last few years and as a result in 2009 a security layer designed specifically for Contiki and called ContikiSec was presented [46]. Figure 3.1 shows the overall structure of Contiki OS stack with the ContikiSec included.

ContikiSec provides three modes of operation – confidentiality only (ContikiSec-Enc), authentication only (ContikiSec-Auth) and both authentication with encryption (ContikiSec-AE). After evaluation of six different block ciphers AES in the CBC-CS mode has been selected to be used for the encryption only mode. Contiki-Auth uses CMAC algorithm for generating a 4-byte MAC. Contiki-AE that achieves the highest level of security by providing confidentiality, authentication and integrity uses Offset Codebook Mode (OCB) with the AES as the underlying block cipher.

One major disadvantage of ContikiSec is the usage of the most primitive key management mechanism – the system relies on all nodes in the network having the same 128-bit key. This clearly makes the whole network much more vulnerable to various attacks when compromise of one node makes the entire network not secure. Introducing public-key cryptography protocols to the system (such as ECDH, ECDSA, etc) would make it more fit IP-enabled WSNs.

3.3 Network-layer Security

Providing security in IP-enabled WSNs at the network layer is the most recent direction in the security research. Starting from 2010 several proposals were presented on how to integrate security into the 6LoWPAN adaptation layer.

Jorge Granjal et. al. in [47] propose and evaluate a Secure Interconnection Model for WSN (SIMWSN) that provides fundamental security properties

like confidentiality, authentication and integrity as well as end-to-end security between sensor nodes and host in the Internet by introducing new 6LoWPAN security headers. SIMWSN consists of several components that are responsible for enforcing security profiles, managing keys, intrusion detection and managing nodes (status as well as resource availability). These components are defined separately for the actual nodes and for the security gateways that support both IPv6 and 6to4 tunneling on the Internet interface and act as the sink nodes for associated WSNs. Security gateways are assumed to be powerful devices like PCs. Figure 3.2 illustrates the operational scenario of SIMWSN.

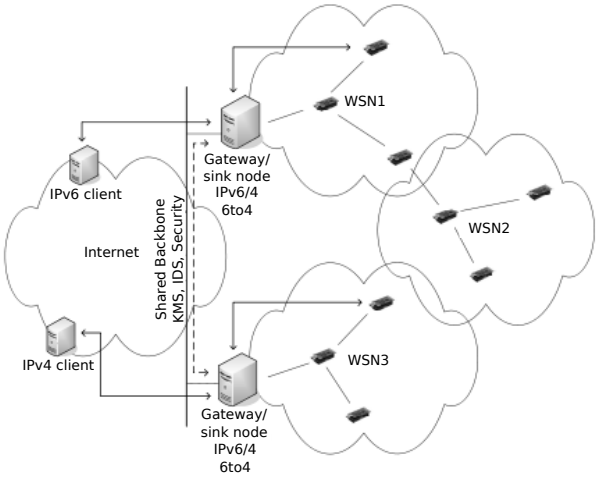

Figure 3.2: SIMWSN operational scenario

According to the authors, SIMWSN allows for end-to-end security in transport mode, like IPsec in traditional IP environment or alternatively it could configure the security gateways to protect the connected WSN from the Internet. In the latter case nodes on the Internet would connect to the gateways using IPsec in tunnel mode or without security, while security on the WSN side will achieved by using SIMWSN in tunnel mode.

The key management system that is present on the security gateway supports IKE to communicate with the Internet hosts. It also transmits the 160-bit ECC public key to each node in the corresponding WSN during the bootstrap process, the list of all keys is kept at the gateway and is maintained by the Security manager. Using the keys that each node in the WSN has authentication using ECDSA and establishing a session key using ECDH is possible.

Authors also propose the security extensions for 6LoWPAN packets in order to cope with the SIMWSN – this at the time was the first proposal towards

the network-layer security in IPv6-enabled WSNs. They describe what dispatch codes could be used in the 6LoWPAN headers and how compressed AH and ESP headers would look like (see Figure 3.3). However as the paper was written before the update to the 6LoWPAN compression standard was released, authors make use of the LOWPAN_HC1 compression which is now discouraged.

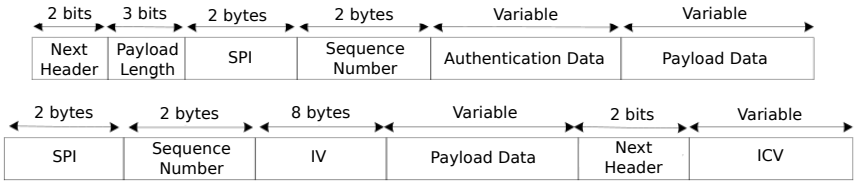

Figure 3.3: AH (above) and ESP (below) compressed security headers

SIMWSN has not been implemented and remained only a proposal, however Shahid Raza et al. continued the idea of using network-layer security in IP-enabled WSNs and in 2011 introduced a compressed version of IPsec for 6LoWPAN networks [48], which was specified, implemented and evaluated.

In their work authors have presented a possible encoding for the AH and the ESP extension headers using the up-to-date LOWPAN_NHC. They propose to use the already defined encoding form for IP extension headers by using the two remaining values for the EID (IPv6 Extension Header ID) as specified in the RFC, namely 101 and 110, to encode AH and ESP respectively. Figures 3.4 and 3.5 illustrate the proposed encodings of AH and ESP headers. A full specification of what each bit in the encodings describes can be found in the technical report [49].

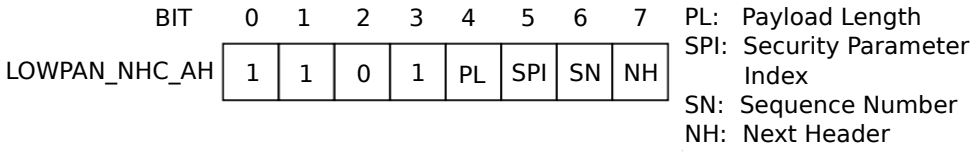

Figure 3.4: NHC encoding for IPv6 Authentication Header

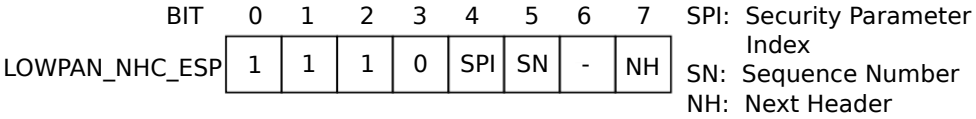

Figure 3.5: NHC encoding for IPv6 ESP

As mentioned earlier the compressed version of IPsec was implemented, however the implementation, which was done for the Contiki OS, uses pre-shared keys in order to establish SAs. In other words another part of the IPsec suite, namely the IKEv2 protocol has not been done. What has been done is the AH and the ESP protocols including the mandatory security suites such as HMAC-SHA1-96 for AH and AES-CBC for ESP. The overall memory footprint of the IPsec implementation ranged from 3.9 kB to 9 kB ROM and 0.3 kB to 1.1 kB RAM depending on the used protocol and mode of operation.

3.4 Transport-layer Security

One commonly used way of providing end-to-end security is using TLS protocol, which guarantees security between applications, includes a key exchange and provides authentication, confidentiality and integrity. Several implementations of TLS (or rather its predecessor SSL) have been presented for use in IP-enabled WSNs, such as Sizzle [36] or SSNAIL [37].

Sizzle was the first implementation of the HTTPS stack (HTTP that is being used over SSL) that used MD5 and SHA1 as hashing algorithms, RC4 for bulk encryption, ECDH and ECDSA for key exchange and showed that using those protocols was feasible for constrained devices. In the used setup, however, the TCP/IP connections from the Internet terminated at the gateway and not at the motes and multiple devices running Tiny OS were connected via that gateway while the secure web servers that were running on the motes were mapped to the distinct ports on the gateway. Several assumptions were made with regarding to the SSL handshake in order to decrease the number of messages being sent, but even then the number of packets sent was quite large, mostly due to the fact that the authors had to also implement a reliable TCP-like data transfer mechanism, since at the time of creating Sizzle Tiny OS did not include such mechanism.

Evaluation of Sizzle showed firstly that using ECC for public-key cryptography is by far more suited for constrained devices than RSA (which was implemented solely for the performance comparison) and secondly that performance of the secure web server on the mote is quite acceptable for infrequent communications.

SSNAIL is a lightweight SSL implementation that was developed as a security mechanism for the project called Sensor Networks for All-IP worLd (SNAIL), which had a goal of implementing a IP-WSN platform with a widespread test-bed. SNAIL sensor nodes run on two different platforms – OSAL (Operating System Abstraction Layer) [50] of TI solution and ANTS EOS [51] of RESL (Real-time an Embedded Systems Laboratory) in ICU.

SSNAIL uses the same security mechanisms as Sizzle does, namely ECC with ECDG-ECDSA-RC5-MD5, however it eliminates the usage of the security gateway. Its implementation uses 30 kB of Flash memory and around 500 bytes of RAM.

3.5 Application Layer Security

In certain (quite rare) cases, applications choose to introduce their own security protocols in the application layer instead of using security in the underlying layers. One example of such protocol is the User-based Security Model (USM) for the Simple Network Management Protocol version 3 (SNMPv3) [52] which is responsible for authenticating, encrypting and decrypting SNMP packets. In 2010 S. Kuryla has implemented an SNMP agent under the Contiki OS [53] and that implementation also included the USM security. It provided support for the HMAC-MD5-96 authentication and CFB128-AES-128 symmetric encryption operations and was by far the largest part of the implementation with respect to the statically allocated RAM – 122 bytes out of 235 bytes for the entire agent. In terms of Flash memory the cryptography primitives, such as AES and MD5, occupied around 60% of the entire implementation – approximately 20 kB.

Chapter 4

Implementation

In this section we will provide information about the developed TLS and DTLS libraries starting from describing the platform, the operating system and the network setup used for development and testing. We will then go into the design principles followed during the project, give a general overview of how the libraries work and finally give various details regarding the implementation.

4.1 AVR-Raven Platform

The AVR-Raven board, that was used throughout this thesis for development and evaluation purposes, is produced by the Atmel Corporation. It includes two microcontrollers (MCUs), a radio receiver chip and an LCD display. The 8 MHz 8-bit AVR ATMega1284P microcontroller is responsible for the communication while the 8 MHz 8-bit AVR ATMega3290P controls the LCD Display.

Both MCUs are the modified Harvard architecture 8-bit RISC single chip MCUs, with the program and the data being stored in separate physical memory systems, that appear in different address spaces. The Flash, EEPROM, and SRAM physical memory systems are all integrated onto a single chip. The ATMega1284P has 16kB of SRAM, 128kB of flash non-volatile program memory and 4kB of EEPROM. The ATMega3290P has 2kB of SRAM, 32kB of flash and 1kB of EEPROM.

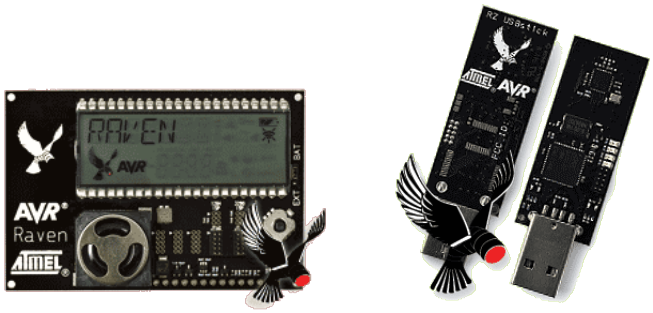

Figure 4.1: AVR Raven mote and the AVR RZUSBstick used for providing an IEEE 802.15.4 interface to a PC.

4.2 Contiki Operating System

The Contiki OS [54] is a small, open source, highly portable multi-tasking operating system, which was developed at the Swedish Institute of Computer Science for use in a number of memory-constrained networked embedded systems and wireless sensor networks. It is written in C, requires between a few bytes to several kilobytes of RAM, depending on the configuration, and is freely available under a BSD license.

The Contiki OS consists of an event-driven kernel on top of which application programs are dynamically loaded and unloaded at runtime. Each program must start at least one process and only one process can be running and using CPU at a time. It is a responsibility of each process to give up the execution and to prevent the entire system from deadlocking.

Inter-process communication is done via posting events. Once one process needs to notify another process about something, it places a respective event to the event queue. The kernel goes over the queue and dispatches the event to the requested process (or to all running processes, if the event was broadcasted). It is also possible to pass data between the processes by posting an event together with the pointer to the data.

4.3 Network Setup

During the implementation and testing phase the following setup was used. A simple client application running on one of the AVR Raven motes would connect to a laptop that was acting as a proxy forwarding all of the messages from the client to a simple server application running on the second mote and the other way around. This way we were able to sniff on the packets sent during the TLS/DTLS handshake and the following communication by

using Wireshark on the laptop. In order to provide the laptop with the IEEE 802.15.4 interface, to be able to communicate with the motes, the AVR RZUSBstick, a USB stick with a 2.4 GHz transceiver, was used. Both the AVR Raven and the AVR RZUSBstick can be seen on Figure 4.1 and the setup is presented in Figure 4.2.

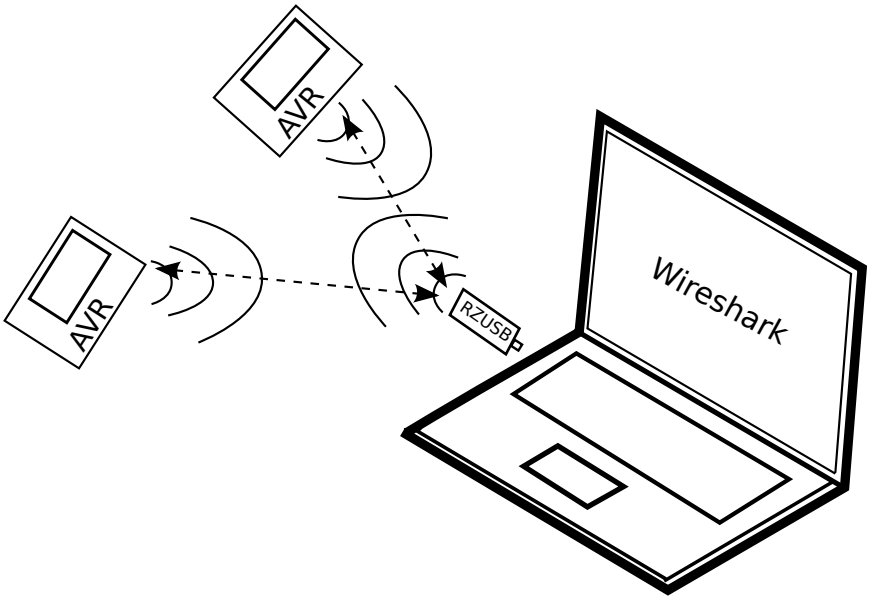

Figure 4.2: Overview of the network setup used during implementation and evaluation.

4.4 Design Principles

When implementing the TLS and DTLS protocols several design principles were kept in mind. Firstly, the resulting libraries should have a simple and straight-forward API, that could be easily adopted by programmers, who have experience with programming for the Contiki OS. Secondly, when facing the choice between having a very optimized (speed- and size-wise) but hardly readable and maintainable code, and having an easier understandable code at the cost of additional variables or function calls, the latter option was preferred. However, keeping in mind the memory size limitations present on the devices in the WSNs, it was essential to keep the implementation as light-weight as possible.

TCP/IP support in the Contiki OS is implemented using the uIP TCP/IP stack. We have decided to take the existing uIP interface as an example and we created a similar interface for secured communication. We will now give

a brief overview of how TCP and UDP communication is carried out in the Contiki OS.

Contiki provides a set of functions for connection management and for sending and receiving of user data. When a connection is established, a certain event is sent to the application to notify it that data can be sent. When application data needs to be transmitted, it is copied to a specifically allocated buffer `uip_sappdata`, from where it is read by a `tcpip_process`. That process takes care of the actual transmission (and retransmission in case of TCP). If new data has been received from the remote end-point, this data is stored in another buffer (`uip_appdata`) and a special event is raised. Once the user application receives this event, it can access the data from that buffer. Since this event-based style of programming has become natural for Contiki developers we decided to design the TLS/DTLS library in the similar fashion.

4.5 API and General Overview

The following are the API functions and events that are available in the TLS/DTLS library, shown together with their counterparts from the uIP TCP/IP stack. Since most of the names in the TLS and DTLS libraries are identical up to the letter 'd' in 'dtls', they are shown together.

- `int (d)tls_listen(uint16_t port, uint8_t max_conn)` – Start listening for incoming TLS/DTLS connections on the specified port. (tcp_listen)
- `void (d)tls_connect(uip_ipaddr_t *ripaddr, uint16_t port)` – Start a TLS/DTLS connection with the specified end-point. (tcp_connect, udp_new)
- `int (d)tls_write(Connection* conn, char* toWrite, int length)` – Send data over an existing TLS/DTLS connection. (uip_send, uip_udp_send)
- `void (d)tls_close(Connection* conn)` – Close an existing TLS/DTLS connection.
- `(d)tls_event` – Event raised to notify the user that a (D)TLS-related event has occurred. (tcpip_event)

Once the application receives the `(d)tls_event` it can check which event has occurred by calling the following functions:

- `(d)tls_connected` – returns true if the D(TLS) connection has just been established. (uip_connected)

- `(d)tls_newdata` – indicates if there is new data available on the connection. (uip_newdata)
- `(d)tls_rehandshake` – indicates that a rehandshake procedure has started, therefore any new data sent by the application will not be delivered until the connection is reestablished. Once that happens a new `(d)tls_event` will be raised and `(d)tls_connected` will return true.
- `(d)tls_closed` – indicates that the D(TLS) connection has been teared down. (uip_closed)

Once the application receives an event that the new data has been received, it can access this data in the special buffer `(d)tls_appdata`. The application can also get the length of the received data by calling the `(d)tls_applen()` function.

Figure 4.3 shows the general overview of how the TLS library works, the overview of how the DTLS library works is very similar and therefore omitted. As can be seen from the figure, the library should be used either by the client or the server side, but not both. The process, which acts as a TLS server, is started by the call to the `tls_listen()` function. This function takes two parameters: a port number on which the server will be listening for incoming connections and an integer, specifying the maximum number of allowed connections at a time (at the moment the implementation allows only one connection at a time, but the parameter was introduced having possible future improvements in mind).

The application, which wants to act as a client, has to connect to the server by calling `tls_connect()` with the IP address and the port of the server as parameters. This call results in starting the `TLS_client_process`. As discussed in Section 2.4.1 the TLS communication has to start with a handshake, during which keying material for further communication is negotiated. Therefore the first message that is sent by the client to the server is the ClientHello message. Upon receiving this message, the server replies with the ServerHello and the rest of the handshake follows. After the handshake is successfully finished, both the client and the server processes send a `tls_event` to the application processes together with the `Connection` data structure which holds all necessary information about the just-established TLS connection and which is then used for sending data over this connection. Details about the contents of the data structure are discussed in section 4.7.

Having received the event the application should check which event occurred by means of the functions discussed above. Once the application understands that the connection has been established, it can start sending data by supplying it to the `tls_write()` function together with the `Connection`,

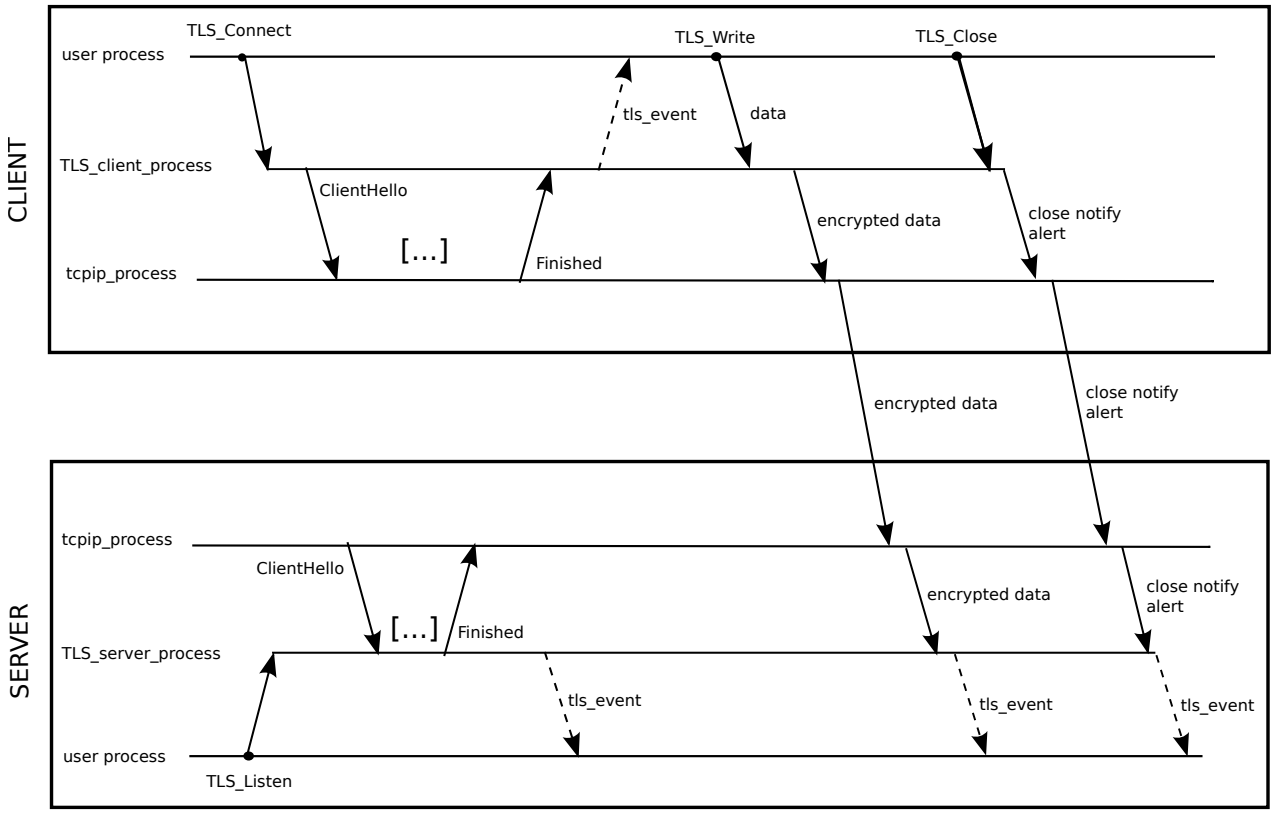

Figure 4.3: TLS overview

as mentioned earlier. The TLS process then takes the data, encrypts it using the keys negotiated during prior handshake and feeds the resulting ciphertext to the `tcpip_process` which takes care of delivering it to the destination. On the receiving side, the TLS process gets the ciphertext from the `tcpip_process`, decrypts it, stores the result into a `tls_appdata` buffer and sends an event to the user application. The application can then access the data and respond accordingly.

4.6 Implementation Details

In this section we will discuss various implementation details of the TLS/DTLS library. We will first look at how the code is structured and will then go over each file and data structure used in depth.

4.6.1 Structure

The first question that had to be answered even before starting to implement the library was how the library should be accessed and where in the Contiki OS it should be placed. As we were taking the uIP stack as an example implementation, it was decided to place TLS and DTLS alongside uIP into the `core/net` folder of Contiki. This way any application can make use of the TLS or DTLS security by simply including `<net/dtls/dtls.h>` or `<net/tls/tls.h>` respectively to their application as well as setting `DTLS=1` or `TLS=1` in the application's Makefile. Minor modifications had to be applied to the main Makefile located in the root of the Contiki OS, in order to compile the source code of the respective library when the `DTLS` or `TLS` flag is set.

Since TLS and DTLS protocols are closely related and hence are quite similar to each other, the TLS and DTLS libraries we have implemented also have very similar structures. One can subdivide the code base into four parts that perform different tasks – input parsing, output generation, data hashing and encryption/decryption of messages. In the following subsections we will look at each of those tasks in more details.

4.6.2 Input Parsing

In general input parsing for both TLS and DTLS is split into three stages – processing the input received from the uIP layer in order to retrieve full (D)TLS records, processing those records to get to the messages from underlying protocols (Handshake, Application Data, etc.) and parsing the message to check its validity.

In case of TLS, which uses TCP as the underlying transport, the stage of processing the input is more complicated than that needed for DTLS, even though TCP provides reliable delivery of data, unlike UDP. Since TCP provides a stream of data, the message sent by an end-point can arrive at the receiving end-point in arbitrary chunks of data, without preserving the boundaries of the TLS records. Therefore, the library has to take care of buffering the incoming message in case what it received was just a part of the TLS record. Since the TLS record header includes the full length of the record, once the length is received, the library knows how much memory needs to be allocated. However, it could happen that the record header is split by the TCP engine, in which case the length could still be missing or be split in half. The `process_input()` function copes with every possibility of input fragmentation and passes the retrieved record to the `process_record()` function.

According to the DTLS specification each DTLS record has to fit into a UDP datagram, which makes input parsing much simpler – once the UDP datagram is received one only has to process the record header before passing the contents of the record to the next stage of input parsing.

Once the first stage of input parsing is complete, the TLS and DTLS libraries have to process the contents of the received TLS/DTLS record – this can be the data from one of the four underlying protocols. In TLS, this task is almost identical to that, faced in the first stage – the TLS Record layer does not preserve boundaries of the underlying protocols' messages, just like the TCP engine does not preserve boundaries of the sent data. As such, the `process_record()` function in the TLS library is structurally very similar to `process_input()` and once the record is processed the Handshake message or the Application Data message is passed to the `act_on_full_message()` function.

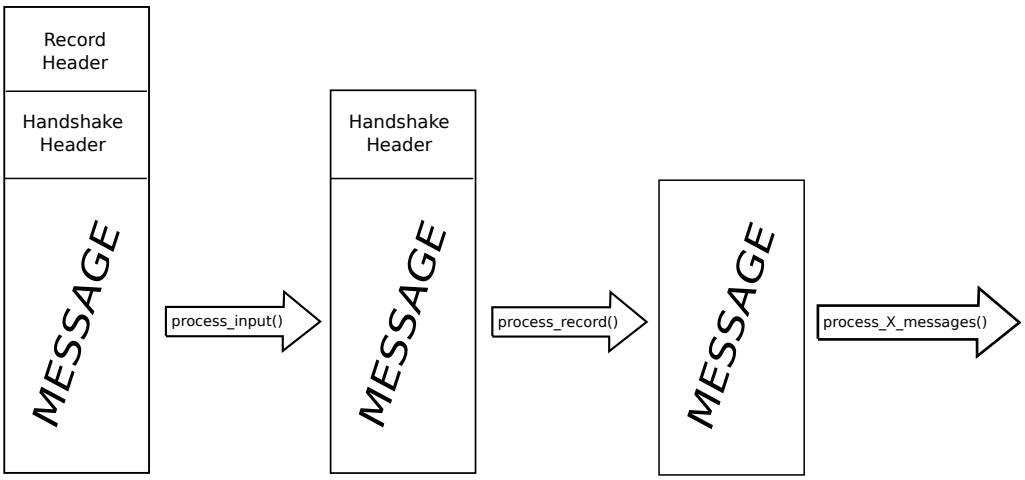

Figure 4.4: TLS/DTLS input processing pipeline

Parsing of the DTLS record is very different to that of TLS record. Since UDP does not promise that all datagrams are delivered in the right order, if delivered at all, DTLS introduces explicit fragmentation fields in the Handshake protocol messages (see Section 2.4.2). Using information from those fields, the end-points are able to reconstruct the entire message even if the fragments arrived in the wrong order. This also makes parsing of the messages simpler than in TLS.

Once the data from the underlying protocol has been retrieved it is passed to the `act_on_full_message()` function, which takes care of the last stage of input parsing, by calling either `process_server_messages()` function in case the application using the library acts as a client or `process_client_messages()` function otherwise.

The overview of the input parsing pipeline is depicted in Figure 4.4.

4.6.3 Output generation

Once the input is parsed and processed the TLS/DTLS library either deciphers the Application Data and passes it to the user application or, in case of the Handshake Protocol messages, it needs to generate a response and send it back to the other end-point. To know what this response has to be, we have implemented a simple state machine, that keeps track of what message in the handshake protocol we are expecting to receive next. Since we only get to the output generation step once the input processing completes successfully, knowing which message we were expecting to get uniquely identifies which message has to be sent back.

The functions `response_to_client_messages()` used by the server as well as `response_to_server_messages()` used by the client take care of doing all necessary computations before the response can be constructed. Once the preparations are complete, we can generate the response message by using one of the `create_X` functions declared in the `util.h` file. Each of those functions constructs a valid Handshake protocol or Alert protocol or Application Data protocol message and places it into the provided buffer. Once the response is created, it can be sent out to the other end-point.

4.6.4 Data Hashing

There are several instances throughout the TLS and DTLS handshake when it is required to perform SHA256 hashing and HMAC message authentication code construction. For example, the Finished message of the handshake for both TLS and DTLS is the hash, acquired over all of the previous handshake messages starting from ClientHello message. In case of the stateless cookie exchange in DTLS, RFC 6347 suggests that the cookie is generated by the means of HMAC-SHA256 over the information received in the original ClientHello message. Also the premaster secret as well as the master secret and the keying material required for further data encryption are generated using the Pseudorandom Function (PRF) which is described in RFC 5246 and uses HMAC-SHA256 as its core component. The implementation of the PRF function can be found in `util.c`.

In order to perform hashing and HMAC computations, a small library written by Olivier Gay (and distributed under the BSD license) is being used [55]. To save code space, parts of the library that were not required (such as code responsible for HMAC-SHA224, HMAC-SHA512, etc.) have been removed, leaving only the essentials needed for our purpose.

4.6.5 Encrypting and Decrypting of Messages

According to the TLS and DTLS specifications the first message that has to be encrypted in the TLS/DTLS session is the last message of the handshake, namely the Finished message. Starting from this message the rest of the communication until a rehandshake or a teardown has to be encrypted using the cipher suite negotiated during the handshake. Our current implementation supports so far only one cipher suite - TLS_PSK_WITH_AES_128_CCM_8. It was described in [56] and was one of the mandatory suites to implement according to the CoRE Working Group of the IETF and their Constrained Application Protocol (CoAP) Internet Draft [57].

The Counter with CBC-MAC mode (CCM) for AES encryption provides both confidentiality and authentication while using only AES encrypt operation, therefore making compact implementations possible. This ability of providing a high level of security while keeping implementation small is what makes this mode a good fit when working with constrained devices. Its implementation can be found in `aes_ccm.c`, `aes_ccm.h` and `aes.h`. The implementation of the AES encryption was originally taken from the OpenSSL library and then further modified and adapted for the targeted platform by Siarhei Kuryla during his work on SNMP for the Contiki OS. One of the biggest modifications that has been introduced was moving the large array of constants, that the OpenSSL library was using, from RAM to the read-only flash program memory, thus avoiding filling up 4096 bytes in RAM.

The `encrypt()` and `decrypt()` functions have been written by carefully following the steps described in the NIST Special Publication 800-38C on the CCM Mode for Authentication and Confidentiality [58]. Since the cipher suite used by our TLS/DTLS implementation requires some parameters of CCM to be of certain value, these values have been hardcoded using define statements rather then being passed as parameters to the functions. For example, the value `t`, specifying the number of bytes used for authentication, has been set to 8, `n`, which is the length of nonce, has been set to 12 and so on. If the AES-CCM implementation is to be used in the future, those values should be changed according to the requirements.

The AES-CCM implementation has been verified using the example vector from [58].

The entire flow of input processing and output generation can be seen in Figure 4.5. As an example, the last message of the handshake has been taken – it is the only message that goes through all steps of the process, since it has to be first decrypted, then processed, then the response has to be generated, encrypted and sent.

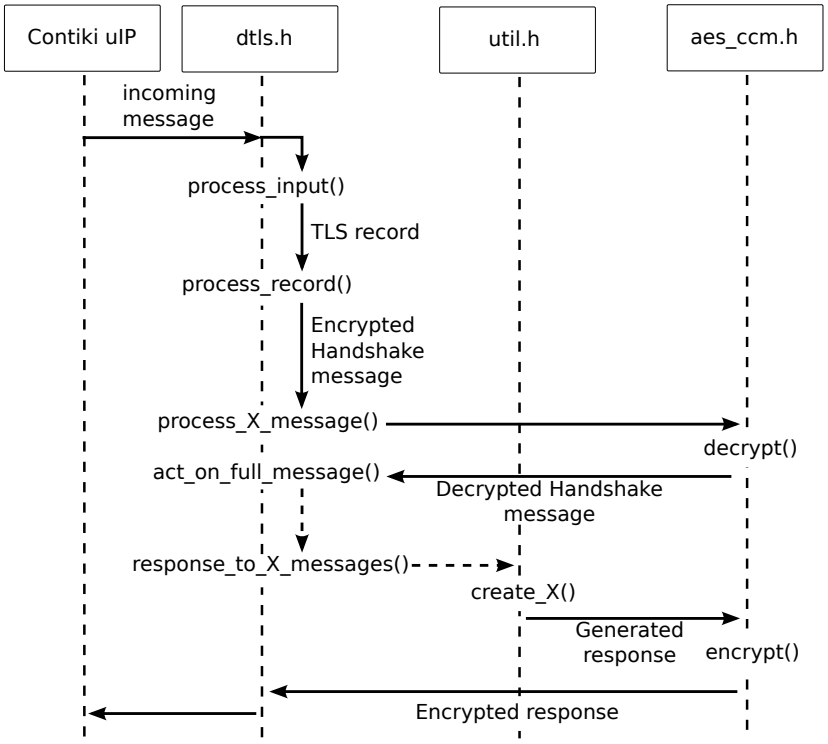

Figure 4.5: Full overview of input processing and output generation.

4.7 Data Structures

In this section we will present some of the data structures used in the code and explain what they were needed for. All of the data structures for TLS and DTLS are defined in `tls.h` and `dtls.h` respectively.

As described earlier, when the TLS or DTLS connection between two endpoints is established, they receive an event together with the `Connection` data structure that holds all necessary information about the established connection in order to be able to send and receive data over it. The `Connection` holds the required security parameters and the information about the actual TCP or UDP connection over which the TLS or DTLS connection has been established. You can see the `Connection` data structure for TLS and DTLS below:

```
typedef struct {
    SecurityParameters* securityParameters;
    struct uip_conn conn;
} Connection;

typedef struct {
```

```
    SecurityParameters* securityParameters;
    struct uip_udp_conn conn;
} Connection;
```

The `SecurityParameters`, needed to be able to successfully encrypt and decrypt data, holds the necessary cryptographic material – server's and client's keys and IVs:

```
typedef struct {
    char* client_write_key;
    char* server_write_key;
    char* client_write_IV;
    char* server_write_IV;
} SecurityParameters;
```

Security parameters used in our implementation differ significantly from those defined in the RFC 5246, appendix A.6. [11], since there much more information is being stored. However, since our implementation only supports one cipher suite and also does not support session resumption, information, such as which MAC algorithm or which Cipher type is being used as well as random values of client and server, is simply not needed.

When the client calls the `tls_connect()` function it provides the IP address and the port of the server to which it wants to connect to. This information then has to be passed to the `TLS_client_process` as the `void* data` parameter of the `process_start` function. For that purpose a simple data structure is needed as shown below:

```
typedef struct {
    uip_ipaddr_t *addr;
    uint16_t port;
} Data;
```

4.8 Additional Notes

In this section various details about the implementation will be discussed, such as our choices for the dynamic memory allocation, storing and retrieving of preshared keys, rehandshake parameters and so on.

4.8.1 Dynamic Memory Allocation

The Contiki OS provides three ways to dynamically allocate and deallocate memory – `memb` block allocation, `mmem` memory allocation and the normal C `malloc()` function. The latter one, while being the usual method in a typical C environment, seems to be the least preferable when it comes to programming for the Contiki OS. According to various discussion on

Contiki-related forums the `malloc()` implementation takes too much space and therefore should be avoided whenever possible.

The `memb` library provides a set of functions for managing blocks of data of constant size. The initial call to `MEMB(name, structure, num)` function allocates an array of `num` objects each of the size required by `structure` and places it into the static memory. The library provides functions for accessing and modifying those objects. Since this method provides memory management for specific structures, while we we were interested in a more generic way of allocating memory, we decided to make use of the last remaining option – the `mmem` library.

The managed memory allocator (`mmem`) provides a service similar to `malloc`. It keeps a statically allocated memory pool and uses a linked list of `struct mmem` objects to access that memory. Each such object contains a link to the next object in the list, a pointer to the allocated memory from the memory pool, and the size of the allocated chunk (see Figure 4.6). A call to `MMEM_PTR` returns a pointed to the allocated memory, therefore providing a level of indirection when accessing memory from the original pool (unlike `malloc`).

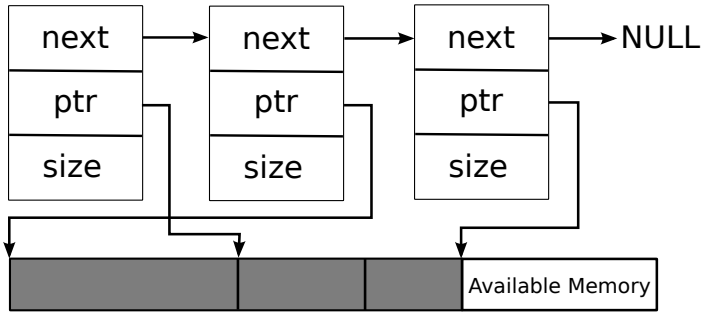

Figure 4.6: Organization of the managed memory in Contiki OS.

Before the `mmem` library can be used, a call to `mmem_init()` has to be made in order to statically allocate the pool of memory. On the AVR-Raven platform this call is not done by default, hence we added it to the `platform/avr-raven/contiki-raven-main.c` file. By default 4 kB of memory are allocated, but since we do not need that much for our purposes the `MMEM_CONF_SIZE` variable has been set to 512 bytes in the TLS and DTLS Makefiles. The user of the TLS/DTLS library can increase that value if need be.

4.8.2 PSK

The cipher suite that we decided to implement makes use of preshared keys (PSK). These keys are supposed to be stored on both the server and the

client side to make the communication possible. When implementing the TLS library, one of the potential use cases which was kept in mind was securing the communication with the NETCONF-light server, that was implemented for the AVR-Raven platform and Contiki OS previously. Since that implementation uses the Coffee File System (CFS) extensively and stores the server configuration file in it, it was decided to also store the psk-identity - PSK pairs in that configuration file. NETCONF over TLS draft [59] specifies how the list of those pairs has to be included into the configuration. In case the configuration is not present on the device, the received psk-identity is checked against a hard-coded value and in case of a match a hard-coded value for the PSK is being used.

We decided not to include the CFS support to the DTLS library to keep it simpler and smaller, hence the hard-coded values are being used at the moment. We understand that this does not achieve the best user experience and it will most-likely need to be modified in the future.

4.8.3 Tuning Parameters

There is one parameter in both TLS and DTLS implementations that can be modified according to the user's needs. The `rehandshake_limit` controls the amount of Application Data in bytes that can be sent before the rehandshake will take place. A two-byte variable is used for this parameter, hence the maximum value, to which it can be set is 65535. This value is also the default one.

Another parameter, which is DTLS specific, is the `RETRANSMIT_INTERVAL`, which defines the amount of time in seconds the end-point waits before retransmitting a handshake message since no response has been received. The DTLS specification suggests making this value 1 second, however due to the limited processing power of the constrained devices we have decided to set this interval to 3 seconds. This parameter can be easily modified to suit the needs of users.

Chapter 5

Evaluation and Testing

Once the implementation was complete, it was important to check whether it worked successfully with other TLS/DTLS libraries and it was also essential to measure the amount of memory our library used on the device and the amount of time required for setting up the connection and encrypting data. It was clear that the library had to be small enough for user applications to also fit on the device and fast enough to be applicable in a large number of environments.

5.1 Interoperability

In order to test the correctness of our library, we first had to find another implementation of TLS or DTLS which would use the same cipher suite as us. However, since the cipher suite that we decided to implement is relatively new, finding other implementations supporting it was challenging. In fact, according to [56], not a single open source TLS implementation supports AES-CCM encryption.

We have nonetheless managed to find one DTLS implementation that supported TLS_PSK_WITH_AES_128_CCM_8, namely `tinydtls` by Olaf Bergmann from Universität Bremen [60]. His implementation is not yet entirely complete (Alert protocol as well as retransmission of lost messages are still work in progress) but it still gave us a chance to test whether our AES-CCM encryption was working correctly and whether a simple client program which used our DTLS library, would be able to communicate with a simple server program, that was using `tinydtls`.

For the purpose of testing, the minimal-net platform was used, which compiles Contiki for your host-system and connects it via a TAP device (e.g. `tap0`). The reason for using minimal-net and not AVR-Raven devices, which

we used during the implementation phase, was that `tinydtls` was ported to run with Contiki but was not tested with 8-bit devices and trying to compile it for the AVR-Raven platform lead to issues with numerous bit-shifts found in the source code. Also using minimal-net allowed for easier debugging, which would have been impossible otherwise.

After several bugs were fixed in both implementations, we have managed to successfully send data from the client to the server both when the client was using our library and the server was using `tinydtls` and the other way around. This gives us reasons to believe that our implementation is correct, though further testing with more implementations, once they appear, would be helpful.

5.2 Memory Usage

The evaluation of the memory usage of our TLS/DTLS libraries has been split into three parts – measuring the statically allocated RAM used, measuring the Flash ROM and the stack size required. In order to yield the results only of the libraries themselves, we first computed all of the above for a simple client program, then added the TLS or DTLS capability and subtracted the results to get the final numbers.

Retrieving the amount of statically allocated RAM and Flash ROM used was straight-forward, the avr-size utility, which is a part of binutils program provides those two values. We have tried to break the memory usage down to as many components as possible to view which part of the code needs improvement. The results can be seen in Table 5.2.

Component	Static RAM (bytes)	Flash ROM (bytes)
Contiki `mmem`$^{\star,\diamond}$	516	238
Contiki CFS*	92	7502
AES-CCM$^{\star,\diamond}$	310	14058
HMAC-SHA256$^{\star,\diamond}$	288	3594
TLS*	683	12874
DTLS$^{\diamond}$	847	19342
TLS Total	1889	38266
DTLS Total	1961	37232

Table 5.1: Memory usage of the different components making up the TLS/DTLS implementations. (The components marked with $\star$ are used by the TLS implementation and the ones marked with $\diamond$ are used by DTLS).

While extracting the amount of statically allocated RAM and Flash ROM memory used was relatively simple, getting the stack usage of the programs

was more demanding. The maximum amount of used stack was discovered by the means of first "painting" the stack with a certain value (setting every byte on the entire stack with that value) and then counting how many bytes from the stack still had that set value before the process in question started and after it ended. The idea was that the stack usage would overwrite the set value, hence making the usage detectable and determinable. Clearly the procedure for "painting" the stack had to be declared in such a way that AVR-libc executed the code before any program had started running. To do so, the function got a special attribute that told the compiler to put the code into the .init section of memory, which contains the code executed at startup.

Program	**Max. Stack Used** (bytes)	**Stack Remaining** (bytes)
TLS client	1844	1544
TLS server	1834	1534
DTLS client	2402	838
DTLS server	2454	786

Table 5.2: Approximate maximal stack usage of the TLS/DTLS libraries.

Results of applying this method to a simple server and a client supporting TLS or DTLS are shown in Table 5.2.

In order to make sure that remaining memory on the device is enough to run a relatively complex application while using the TLS library, we have successfully ported the previously written NETCONF-light implementation to send data over TLS rather than over plain TCP. This makes us believe, that while still having room for improvement, the implementation of TLS is small enough to be successfully used together with other applications.

5.3 Timing Evaluation

We have used the internal AVR-Raven timer to approximate how long each step of the TLS and DTLS handshake takes and also how long it takes for a chunk of data of 100 bytes to be encrypted and sent over the established connection. The AVR timer has a precision of 125 ticks per second, therefore our findings are correct up to 8ms. The results for the DTLS Handshake can be seen in Figure 5.3, where abbreviations stand for messages sent during the handshake as explained in Sections 2.4.1 and 2.4.2.

As can be seen, the majority of the time is spent on generating the keys and creating the Finished messages. In fact, the reason why those computations take that long is because of the many HMAC_SHA256 calculations that have to be done. Hashing is also the reason why generating the Hello Verify

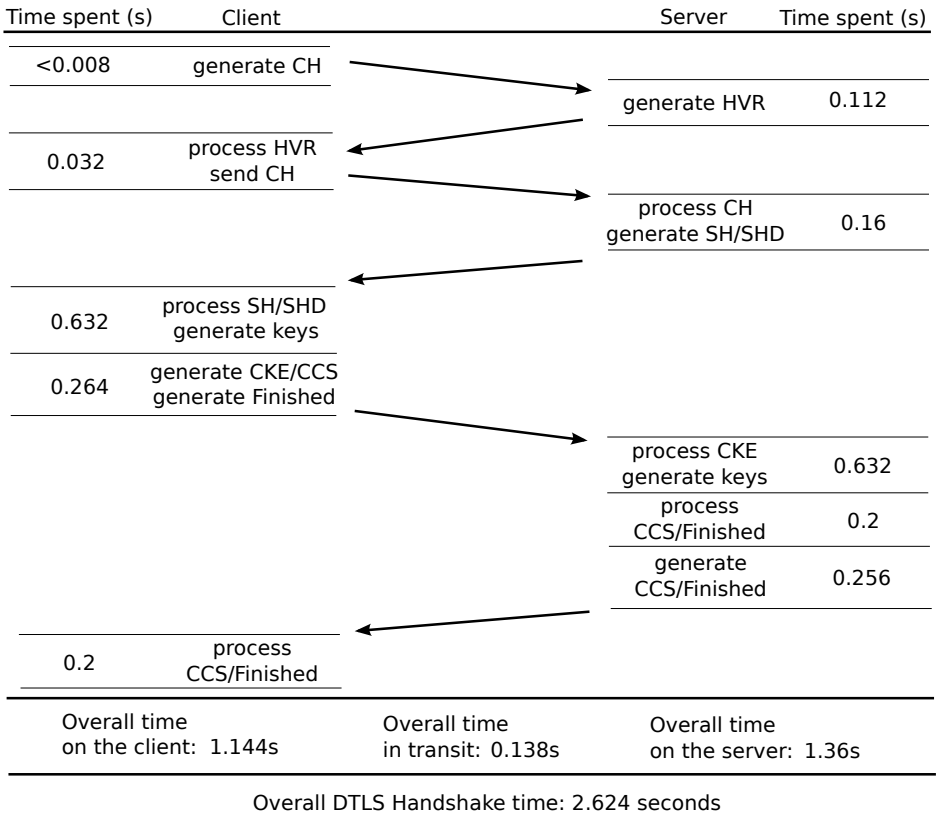

Figure 5.1: Breakdown of the time required for a DTLS Handshake

Request takes that long, since the cookie, which is sent to the client, is a hash of several fields from the Client Hello message.

The TLS Handshake, which is not shown, is very similar to that of DTLS and hence for the most part takes a similar amount of time to complete. The only differences are that before the handshake starts the TCP connection has to be established (this takes 0.048 seconds), the server does not send the Hello Verify Request (thus saving about 0.1 seconds on the server side and 0.032 seconds on the client side), and the handshake has less messages sent hence time in transit is decreased. The overall time needed for completion of the TLS handshake is 2.496 seconds.

It is worth noting that while evaluating the performance of TLS an unexpected behavior was observed. The last flight of messages from the server was retransmitted three times before it was received by the client thus prolonging the handshake by two extra seconds. This behavior appeared to be not random, hence a detailed analysis of where those two seconds were spent was done. We managed to find out that the packets sent from the server were dropped by the 6LoWPAN layer of Contiki due to a bug in its

implementation. Only after a 6LoWPAN fragmentation reassembly timer expired, the received packets would be allowed to pass to the uIP layer. After a minor modification in the code we managed to fix this behavior and avoid the extra wait.

In both TLS and DTLS the encryption of 100 bytes of data takes 0.016 seconds thus making it possible to use the libraries in a large number of interactive applications where fast response time is important. The amount of 100 bytes was chosen because this is how much data the NETCONF-light implementation sends over the established connection at a time.

Chapter 6

Conclusions

Over the period of one semester the implementation of the TLS and DTLS libraries for the Contiki OS has been completed. These libraries provide a simple way of adding security to any Contiki application. Both libraries support the TLS_PSK_WITH_AES_128_CCM_8 cipher suite, which is specified in the draft-mcgrew-tls-aes-ccm-04 [56]. This cipher suite was chosen by the CoRE Working Group of the IETF in their Internet-Draft of the CoAP specification (draft-ietf-core-coap-10 [57]) as the mandatory suite to be implemented when the PSK mode of communication is used.

While not being fully compliant with the TLS and DTLS specifications (since mandatory to implement cipher suites are too heavy for the constrained devices), the implementations follow the specifications very closely. Both TLS and DTLS do not support session resumption, since that would require additional usage of internal storage.

The implementation of DTLS has undergone interoperability testing with the only other available implementation that supports our chosen cipher suite, namely `tinydtls`, developed by our colleagues from Universität Bremen. Our implementation of TLS for the Contiki OS remains to be the only one available at the moment, to the best of our knowledge.

The libraries have been implemented and evaluated using the AVR-Raven platform. To demonstrate that TLS is small enough to be used by other applications even on constrained devices with 16kB of RAM and 128kB of ROM, the NETCONF-light implementation has been successfully ported to run using our TLS library.

Possible future work would include further optimizing the memory usage of the library and providing support for more cipher suites, for instance the ones using elliptic curve cryptography methods, which have been proven to provide a high enough level of security while being small enough for using

in the constraint environment. Also further testing on different platforms would be advantageous.

Additional follow-up tasks, that are of high benefit, would be porting the CCM mode of AES with PSK to various open source libraries like OpenSSL, GnuTLS and so on and porting SNMP for the Contiki OS to run over DTLS instead of USM (however, this would significantly slow down the response time of the SNMP server). In order to speed up the encryption process, adding hardware support for AES cryptography, where it is present, is also an important item on the to-do list.

Appendix A

Appendix

A.1 Source Code

The source code of the library is publicly available under the BSD 2-Clause License and can be found at the CNDS website:
cnds.eecs.jacobs-university.de/software.

A.2 Using TLS/DTLS

To integrate and use the TLS/DTLS libraries the following steps have to be taken.

- Copy the `tls` and `dtls` folders that contain the code to the `core/net` folder within the Contiki OS.
- Modify the `Makefile.include` file, located at the root of the Contiki OS as shown below:

```
...
include $(CONTIKI)/core/net/rime/Makefile.rime
include $(CONTIKI)/core/net/mac/Makefile.mac
#TLS/DTLS change added below
ifdef TLS
    include $(CONTIKI)/core/net/tls/Makefile.tls
endif
ifdef DTLS
    include $(CONTIKI)/core/net/dtls/Makefile.dtls
endif
#TLS/DTLS change done
...
CONTIKI_SOURCEFILES += $(CONTIKIFILES)
CONTIKIDIRS += ${addprefix $(CONTIKI)/core/,dev lib net net/mac net/rime \
```

```
                net/rpl sys cfs ctk lib/ctk loader . }
#TLS/DTLS change added below
ifdef TLS
    CONTIKIDIRS += $(CONTIKI)/core/net/tls
endif
ifdef DTLS
    CONTIKIDIRS += $(CONTIKI)/core/net/dtls
endif
#TLS/DTLS change done
...
```

- In case the `mmem` library is not initialized by default on the platform in use (such as the AVR-Raven platform), `mmem_init()` has to be added to the platform startup code, which is typically found in `platform/<name>/contiki-<name>-main.c`.
- Create an application that requires the use of the libraries within the `examples` folder of Contiki OS. A simple client-server application `test-tls` can also be found at the CNDS website.
- Include `<core/net/tls.h>` or `<core/net/dtls.h>` to the application set `TLS=1` or `DTLS=1` in its Makefile
- The TLS/DTLS API should now be available for use within your application.

Bibliography

[1] CNDS Group at Jacobs University Bremen. WattsApp – The IPv6 Social Telemetry Platform. https://www.wattsapp.net.

[2] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. In *IEEE Commun. Mag.*, volume 40, pages 102–114, 2002.

[3] Yun Zhou, Yuguang Fang, and Yanchao Zhang. Securing wireless sensor networks: a survey. In *IEEE Communicaions Surveys*, volume 10, pages 6–28, 2008.

[4] Wireless medium access control and physical layer specifications for low-rate wireless personal area networks. IEEE Standard, 802.15.4-2003, May 2003. http://standards.ieee.org/getieee802/download/802.15.4-2003.pdf.

[5] Zigbee alliance. http://www.zigbee.org/Home.aspx.

[6] N. Kushalnagar, G. Montenegro, and C. Schumacher. IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals. RFC 4919 (Informational), August 2007. http://www.ietf.org/rfc/rfc4919.txt.

[7] Johnaan W. Hui, Arch Rock Corporation, and David E. Culler. IP is dead, long live IP for wireless sensor networks. In *THE 6TH INTERNATIONAL CONFERENCE ON EMBEDDED NETWORKED SENSOR SYSTEMS (SENSYS'08)*, pages 15–28. ACM, 2008.

[8] Lars Schor. IPv6 for Wireless Sensor Networks, 2009. Semester Thesis, Eidgenössische Technische Hochschule Zürich.

[9] 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. http://www.ietf.org/rfc/rfc4944.txt.

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

[11] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. http://www.ietf.org/rfc/rfc5246.txt.

[12] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), December 2005. http://www.ietf.org/rfc/rfc4301.txt.

[13] S. Kent. IP Authentication Header. RFC 4302 (Proposed Standard), December 2005. http://www.ietf.org/rfc/rfc4302.txt.

[14] S. Kent. IP Encapsulating Security Payload (ESP). RFC 4303 (Proposed Standard), December 2005. http://www.ietf.org/rfc/rfc4303.txt.

[15] C. Kaufman, P. Hoffman, Y. Nir, and P. Eronen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 5996 (Proposed Standard), September 2010. http://www.ietf.org/rfc/rfc5996.txt.

[16] FIPS PUB 197. Advanced Encryption Standard, Nov. 2001.

[17] R. Baldwin and R. Rivest. The RC5, RC5-CBC, RC5-CBC-Pad, and RC5-CTS Algorithms. RFC 2040 (Informational), October 1996. http://www.ietf.org/rfc/rfc2040.txt.

[18] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[19] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[20] S.D. Park, K. Kim, and E. Seo. IPv6 over Low Power WPAN Security Analysis draft-daniel-6lowpan-security-analysis-05. Expires Sepember 16, 2011.

[21] Xiangqian Chen, Kia Makki, Kang Yen, and Niki Pissinou. Sensor network security: a survey. *IEEE Communications Surveys Tutorials*, 11(2):52–73, 2009.

[22] Yun Zhou, Yuguang Fang, and Yanchao Zhang. Securing wireless sensor networks: a survey. *Communications Surveys & Tutorials, IEEE*, 10(3):6–28, 2008.

[23] Seyit A. Camtepe and Bülent Yener. Key distribution mechanisms for wireless sensor networks: a survey. Technical report, Rensselaer Polytechnic Institute, March 2005.

[24] Laurent Eschenauer and Virgil D. Gligor. A key-management scheme for distributed sensor networks. In *In Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 41–47. ACM Press, 2002.

[25] Haowen Chan, Adrian Perrig, and Dawn Song. Random key predistribution schemes for sensor networks. In *In IEEE Symposium on Security and Privacy*, pages 197–213, 2003.

[26] Wenliang Du, Jing Deng, Yunghsiang S. Han, Pramod Varshney, Jonathan Katz, and Aram Khalili. A pairwise key pre-distribution scheme for wireless sensor networks. pages 42–51, 2005.

[27] Sencun Zhu, Shouhuai Xu, Sanjeev Setia, and Sushil Jajodia. Establishing pair-wise keys for secure communication in ad hoc networks: A probabilistic approach. pages 4–7, 2003.

[28] Haowen Chan. PIKE: Peer intermediaries for key establishment in sensor networks. In *In Proceedings of IEEE Infocom*, pages 524–535, 2005.

[29] L. Zhou, J. Ni, and C.V. Ravishankar. Supporting Secure Communication and Data Collection in Mobile Sensor Networks. *IEEE Infocom*, 2006.

[30] Jooyoung Lee and Douglas R Stinson. *Deterministic Key Predistribution Schemes for Distributed Sensor Networks*, volume 3357, pages 294–307. Springer, 2004.

[31] F. Delgosha and F. Fekri. Key pre-distribution in wireless sensor networks using multivariate polynomials. In *Sensor and Ad Hoc Communications and Networks, 2005. IEEE SECON 2005. 2005 Second Annual IEEE Communications Society Conference on*, pages 118 – 129, sept., 2005.

[32] F. Delgosha and F. Fekri. Threshold key-establishment in distributed sensor networks using a multivariate scheme. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1 –12, april 2006.

[33] Victor S Miller. Use of elliptic curves in cryptography. In *Lecture notes in computer sciences; 218 on Advances in cryptology—CRYPTO 85*, pages 417–426. Springer-Verlag New York, Inc., 1986.

[34] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):pp. 203–209, 1987.

[35] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management - part 1: General (revised. In *NIST Special Publication*, 2006.

[36] Vipul Gupta et al. Sizzle A standards-based end-to-end security architecture for the embedded internet. In *Pervasive Computing and Communications*, pages 247–256, 2005.

[37] Wooyoung Jung et al. SSL-Based Lightweight Security of IP-Based Wireless Sensor Networks. In *Advanced Information Networking and Applications Workshops, 2009. WAINA '09. International Conference on*, pages 1112 –1117, may 2009.

[38] ANSI X9.62. "Elliptic Curve Key Agreement and Key Transport Protocols". American Bankers Association, 1999.

[39] ANSI X9.63. "The Elliptic Curve Digital Signature Algorithm". American Bankers Association, 1999.

[40] Haodong Wang and Qun Li. Efficient implementation of public key cryptosystems on mote sensors (short paper). In *In International Conference on Information and Communication Security (ICICS), LNCS 4307*, pages 519–528, 2006.

[41] David J. Malan, Matt Welsh, and Michael D. Smith. A public-key infrastructure for key distribution in tinyos based on elliptic curve cryptography, 2004.

[42] An Liu and Peng Ning. TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. In *Proceedings of the 7th international conference on Information processing in sensor networks*, pages 245–256. IEEE Computer Society, 2008.

[43] Qamar Toheed and Hassan Razi. Assymetric-Key Cryptography for Contiki, 2010. Master of Science Thesis, Chalmers University of Technology.

[44] Chris Karlof, Naveen Sastry, and David Wagner. TinySec: a link layer security architecture for wireless sensor networks. *Networks*, 3:162–175, 2004.

[45] Ernest F. Brickell, Dorothy E. Denning, Stephen T. Kent, David P. Maher, and Walter Tuchman. SKIPJACK Review. Interim Report. The SKIPJACK Algorithm, 1993.

[46] Lander Casado and Philippas Tsigas. ContikiSec: A Secure Network Layer for Wireless Sensor Networks under the Contiki Operating System. In *Proceedings of the 14th Nordic Conference on Secure*

IT Systems: Identity and Privacy in the Internet Age, pages 133–147. Springer-Verlag, 2009.

[47] J. Granjal, E. Monteiro, and J. Sá Silva. A secure interconnection model for ipv6 enabled wireless sensor networks. In *Wireless Days (WD), 2010 IFIP*, pages 1 –6, oct. 2010.

[48] Shahid Raza, Simon Duquennoy, Tony Chung, Dogan Yazar, Thiemo Voigt, and Utz Roedig. Securing Communication in 6LoWPAN with Compressed IPsec. In *Proceedings of the 7th IEEE International Conference on Distributed Computing in Sensor Systems (IEEE DCOSS 2011)*, Barcelona, Spain, June 2011.

[49] Shahid Raza, Simon Duquennoy, Tony Chung, Dogan Yazar, Thiemo Voigt, and Utz Roedig. Securing Internet of Things with Lightweight IPsec. Technical report, 2010.

[50] Texas Instruments, Inc. Z-Stack OS Abstraction Layer Application Programming Interface, 2007. document number:F8W-2003-0002.

[51] Thu-Thuy Do, Daeyoung Kim, and Tomás Sánchez López. An evolvable operating system for wireless sensor networks. *International Journal of Software Engineering and Knowledge Engineering*, 15(2):265–270, 2005.

[52] U. Blumenthal and B. Wijnen. User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3). RFC 3414 (Standard), December 2002. http://www.ietf.org/rfc/rfc3414.txt.

[53] Siarhei Kuryla and Jürgen Schönwälder. Evaluation of the Resource Requirements of SNMP Agents on Constrained Devices. In *Managing the Dynamics of Networks and Services*, volume 6734, pages 100–111. Springer Berlin / Heidelberg, 2011. http://dx.doi.org/10.1007/978-3-642-21484-4_13.

[54] Adam Dunkels, Oliver Schmidt, Niclas Finne, Joakim Eriksson, Fredrik Österlind, Nicolas Tsiftes, Mathilde Durvy. The Contiki OS – The Operating System for the Internet of Things. http://www.contiki-os.org/, Accessed June 2012.

[55] Oliver Gay. Software implementation in C of FIPS 198 Keyed-Hash Message Authentication Code HMAC for SHA2. http://www.ouah.org/ogay/hmac/, Accessed January 2012.

[56] D. McGrew and D. Bailey. AES-CCM Cipher Suites for TLS draft-mcgrew-tls-aes-ccm-04. Expires November 9, 2012.

[57] Z. Shelby, K. Hartke, C. Bortmann, and B. Frank. Constrained Application Protocol (CoAP) draft-ietf-core-coap-10. Expires December 27, 2012.

[58] Morris Dworkin. Recommendation for block cipher modes of operation: The ccm mode for authentication and confidentiality. NIST special publication 800-38c, National Institute of Standards and Technology (NIST), May 2004. See http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C.pdf.

[59] M. Badra. NETCONF Over Transport Layer Security (TLS) draft-badra-netconf-rfc5539bis-02. Expires October 30, 2012.

[60] Olaf Bergmann. tinydtls – a basic dtls server template. http://tinydtls.sourceforge.net/, Accessed June 2012.