Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Media Informatics

Evaluation of the TLS and DTLS protocols in the iSense sensor network operating system

by
Dániel Géhberger

Supervisor
Dr. Attila Vidács

External supervisor
Dr.-Ing. Carsten Buschmann
(coalesenses GmbH)

October 24, 2013

Contents

Kivonat

A szenzorhálózatok napjainkban kezdenek elterjedni a világon, a közeljövőben azonban ez a terjedés jelentősen fel fog gyorsulni. A cél, hogy könnyen lehessen olcsó eszközökből elosztott hálózatokat építeni. Szintén elvárás lesz hamarosan, hogy minden elektronikai eszköz képes legyen kapcsolódni az Internethez. Az Internetes világban a biztonság egyértelműen kulcskérdéssé nőtte ki magát, és biztosítására számos módszer létezik. A szenzorok számára megfelelő megoldás kiválasztásakor elsődleges szempontként figyelembe kell venni azt, hogy képes legyen a mindennapi eszközökkel (például egy webböngésző) együttműködni.

A dolgozat a TLS (Transport Layer Security) és a DTLS (Datagram Transport Layer Security) protokollok megvalósítását mutatja be az iSense szenzorhálózati operációs rendszer felhasználásával.

Ahogy a dolgozatból kiderül, ezeknek a protokolloknak már több implementációja is létezik különböző mikrokontrolleres platformokra. Ezen megvalósítások hátránya azonban, hogy előre osztott kulcsokra építenek, és ennek következtében nem képesek mindennapi eszközökkel való együttműködésre. Ezt a korlátot az itt bemutatott megoldás az elliptikus görbék használatával megvalósított kulcscserével oldja fel. A különböző elkészített kriptográfiai elemek kombinációjával 10 támogatott titkosító készlet (cipher suite) érhető el, ami jelentősen szélesebb körű alkalmazhatóságot biztosít a korábbi megvalósítások által nyújtott 1–2 titkosító készlettel szemben.

A megoldás felépítése moduláris szemléletű, így lehetséges a TLS és a DTLS külön illetve együttes használata, maximálisan kihasználva azok közös vonásait. Ugyan ez a szemlélet igaz a kriptográfiai elemekre is, így a kód mérete és a memóriahasználat az adott alkalmazáshoz szükséges minimális szintre szorítható.

Az elérhető átviteli sebesség erősen függ az aktuálisan alkalmazott titkosítási módszertől. Összességében elmondható azonban, hogy az optimalizálásnak és a hardvertámogatásnak köszönhetően, a sebesség a titkosítással is racionálisan használható marad. Amennyiben a kommunikáció utolsó lépése a rádión keresztül történik, az okozott lassulás a válaszidőben 4–20% közötti értékre szorul. Az elkészített TLS illetve DTLS protokollok képesek együttműködni más rendszerekkel, és idővel piaci termékek részei is lesznek.

Abstract

The Sensor Networks have started to spread in the world, but in the near future this trend will be accelerating dramatically. The main goal is to build up distributed networks with cheap wireless nodes. According to the actual trend, it will be necessary to connect all electronic devices to the Internet. Nowadays the security is one of the key points of the Internet, and there are many different standard and non-standard solutions to achieve the proper level of it. In case of sensors, the possible interoperability with normal applications (such as a web browser) must be considered.

This paper demonstrates the implementation of the TLS (Transport Layer Security) and the DTLS (Datagram Transport Layer Security) protocols in the iSense sensor network operating system.

As it will be detailed in later chapters, there are different available implementations of these protocols for microcontroller-based systems. The drawback of these solutions is that these were built on pre-shared keys, which restricts their interoperability with everyday software solutions. This new implementation overcomes this limitation with the use of elliptic curves for the key exchange. With the variation of the different adopted cryptographic solutions, this solution supports 10 different cipher suites. This increases the usability, compared to the existing solutions, if we consider that those support only 1–2 cipher suites.

This implementation was built in a modular approach, which enables the use of the TLS and the DTLS separately, or together with the optimized re-use of the common parts. The same approach is true for the cryptographic elements, and this way the current code-size and memory footprint can be optimized for the actual application.

The actually used cipher heavily influences the performance of the communication. However, due to the optimizations and the hardware support, the maximum performance is practically in a usable range with the extra security layer. If the last hop of the communication is done via the radio interface, the security causes only 4–20% overhead in the round trip time. The presented solution can work together with other systems and will be part of products in the future.

Chapter 1

Introduction

This chapter gives a brief high-level introduction for the topic of this paper. The first section describes the main ideas and trends of wireless sensor networks. The second section adds security related considerations and the third one introduces the target platform. Finally, the last section summarizes the contents of the different chapters.

1.1 Wireless sensor networks

Nowadays we can find sensors everywhere around us. These sensors can measure the parameters of the environment and in some cases they can also intervene. The tracked parameters are simple ones in most of the cases, such as, movement, light, humidity or pressure. For a temperature monitoring station, only a simple heat sensor could be enough. However, if the task is to monitor the climate of a field, then something more complicated would be needed. The idea is that if we connect the relatively cheap sensors to each other, we could work with a distributed network. On the one hand, this way we can discover many correlations between the measured values, on the other hand, if we have intervening nodes in the system, we can influence the parameters of the environment.

In the past, all deployed sensors were wired constructions, but nowadays every technology in information technology is moving towards the wireless solutions. The field of sensors follows this trend as well which results in wireless sensor networks (WSNs). A typical sensor mote is equipped with a microcontroller, a radio, some LEDs, buttons, batteries and a serial communication interface. The sensors can be built-in ones, or they can be installed on an attachable sensor board. An installation can contain hundreds or thousands of simple nodes and a few special ones called sink nodes (or base stations) which are attached to computers, and this way they could act as communication tunnels between the WSN and the outside world. According to the vision of the future, the sensor motes will be cheap and a lot of these will be used in quite different types of products.

According to the 'all-IP' trend, every object will be connected to the Internet [1]. The nodes in the network need to have IP addresses, as other Internet ready devices. This vision of

the future – when every device will have a unique IP address – is called the Internet of Things (IoT). The wireless sensor networks provide a possible technical solution for this IoT vision. A new and determining type of data traffic will born which will be produced by the Machine-to-Machine communication. Researchers say that there will be approximately 50 billion smart objects by 2020 [2]. Since the security is getting more and more important in information technology, it must be considered in case of the IoT.

1.2 Security in the Internet Protocol stack

The Internet enabled WSN solutions must provide standardized ways to reach sufficient level of security, because it is getting extremely important in information technology nowadays. As it will be detailed in this section, in an Internet Protocol stack security can be provided at different layers.

For WSNs, the IEEE 802.15.4 radio's standard defines security [3]. This is based on the AES (Advanced Encryption Standard) block cipher in CCM (Counter with Counter Block Chaining-MAC) mode and this encryption mode will be detailed in Section 2.3.3.2. The main disadvantage of this solution is that it protects the messages only inside the low-power radio network. Since the aim of this work is to secure the communication from the server to the client (possibly through the Internet), this solution cannot be used.

The IPSec is a protocol suite that runs in the network layer. There are two main protocols to use: Authentication Header (AH) and Encapsulating Security Payload (ESP). The AH provides authentication, and the ESP provides both authentication and confidentiality. Since the IPSec runs in the IP layer, it protects all transport protocols at the same time [4].

This paper focuses on securing HTTP (Hyper Text Transfer Protocol) and CoAP (Constrained Application Protocol) transactions in WSNs. The HTTP is used to access normal web pages with web browsers. The secured version of the HTTP (known as HTTPS) is provided with the TLS (Transport Layer Security) protocol in all general cases. The TLS can operate over stream based protocols (such as the TCP). The CoAP is a protocol developed specially for machine to machine communication, and it operates over datagram based transport protocols (for example UDP). The DTLS (Datagram Transport Layer Security) is the pair of the TLS, but it also handles the missing reliability of the underlying datagram based protocol. The TLS and the DTLS together provide security for the same type of applications as the IPSec. However, it must be noted that for an IPSec implementation the differences between the IPv4 and IPv6 must be considered. Since the target is to interoperate with commercial or standard HTTP and CoAP applications, the transport layer security solutions are selected from the listed options.

1.3 The iSense hardware platform and operating system

The iSense hardware platform is produced by the Coalesenses GmbH[1]. The hardware solutions spread on a large scale, from WSN devices (Ethernet gateway, repeaters, USB stick) to WSN modules (core modules, gateway modules, sensor modules, power modules, ...). The devices are based on the IEEE 802.15.4 radio enabled Jennic microcontrollers (JN5148). For this work I was using Ethernet gateways and Core Modules. The name of the Ethernet gateway is 'NET10' and it has an Ethernet interface besides the 802.15.4 radio. The main advantage of this device is that it can easily connect the sensor network to the normal Internet. The NET10 is also equipped with a microSD card slot. The Core Modules are more simple devices with only radio interface. The JN5148 microcontroller is a 32-bit RISC controller, which can operate up to 32 MHz. It has 128 kB of RAM which is shared for the code and the memory (stack and heap) during runtime. Figure 1.1 shows the NET10 on the left side and the Core Module version 3 on the right side.

Figure 1.1: *iSense NET10 and Core Module 3 devices*

The iSense OS is also provided by the coalesenses GmbH as a software solution for different devices. The system is written in C++. This operating system is compilable for different microcontroller platforms (Jennic, MSP430, Pacemate, XMega), but the mainly used target is the Jennic. The system provides many features, even dynamic memory allocation which is not typical for WSNs. Many protocols are implemented for the OS, among others: routing protocols, over-the-air programming and time synchronization. The OS offers IPv4 and IPv6 protocols stacks with UDP and TCP transport protocols. In addition, it provides a full-featured CoAP, and an HTTP server. The listed protocols and applications enable the following possible scenarios with the iSense devices and OS:

- Ethernet + IPv4 + TCP + HTTP
- Ethernet + IPv6 + TCP + HTTP
- Ethernet + IPv6 + UDP + CoAP
- Radio (& Ethernet) + IPv6 + UDP + CoAP

[1] www.coalesenses.com

The list basically shows that the IPv4 and the TCP protocols were not designed (nor extended) to be used over the radio. In these cases, the application can run on a NET10 box, and the HTTP server is accessible with any normal Internet browser. The IPv6 was extended with the 6LoWPAN layer in order to fulfill the requirements of the radio based communication. This enables that the CoAP server can be accessible also via multi-hop radio communication. Figure 1.2 summarizes the features of the iSense IPv4 and IPv6 Dual Network Stack.

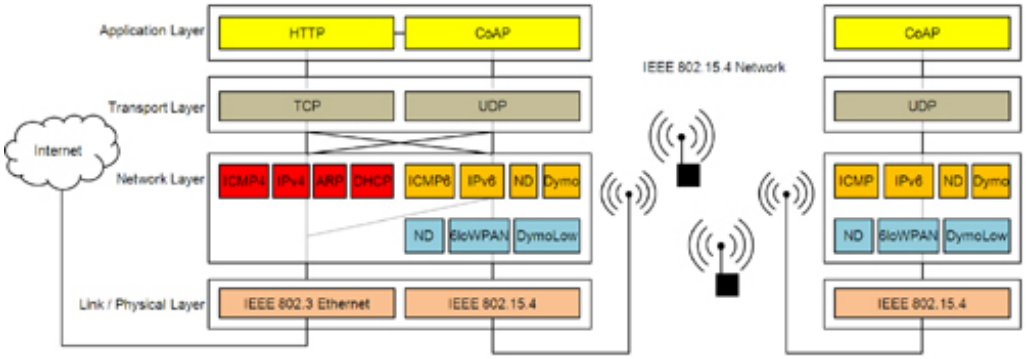

Figure 1.2: *iSense IPv4 and IPv6 Dual Network Stack (Source: www.coalesenses.com)*

1.4 Structure of the paper

In this paper, I show that it is possible to use transport layer security solutions on the Jennic based devices in a way which enables interoperability with wide range of other more traditional implementations. This is achieved with the use of elliptic curve cryptography for the key exchange. The modularity is also a key point in order to provide a scalable product. With this extra security layer, the previously shown list can be rewritten as follows:

- Ethernet + IPv4 + TCP + TLS + HTTP
- Ethernet + IPv6 + TCP + TLS + HTTP
- Ethernet + IPv6 + UDP + DTLS + CoAP
- Radio (& Ethernet) + IPv6 + UDP + DTLS + CoAP

The rest of the paper is structured as follows: Chapter 2 gives an overview about the different related protocols, then it introduces the different symmetric and asymmetric cryptographic building blocks and finally existing solutions will be shown. Chapter 3 starts with the specific list of technical targets and goals of this work and then summarizes the implementation related questions in two parts. First, it gives an overview about the architecture of the system, and then the second part emphasizes and details the main contributions of this paper. Chapter 4 contains the detailed evaluation and interoperability testing of the work. Finally, Chapter 5 concludes the paper and lists some future goals.

Chapter 2

Protocols for IP based WSNs

This chapter discusses the different WSN related protocols in the IP stack from the IP layer to the application layer with the main focus on the security part. The final section of this chapter list the existing solutions for the problem.

Figure 2.1 shows the two basic scenarios which will be used in the whole paper. In both setups the tester computer is connected to the gateway node through a switch. While in the first scenario the computer communicates directly with the gateway, in the second setup the gateway only plays a routing role, and forwards the traffic to another device via the radio interface.

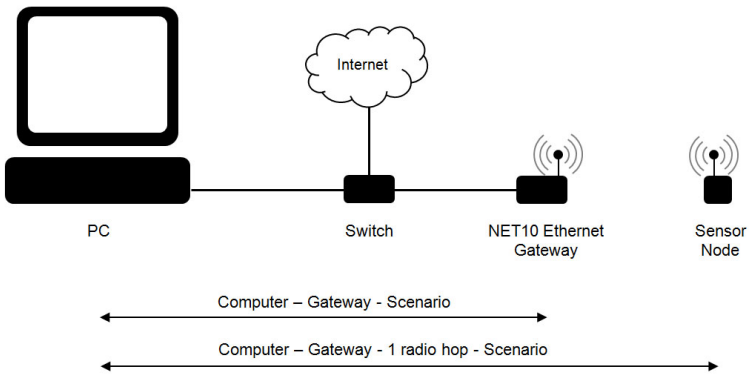

Figure 2.1: *Physical scenarios*

2.1 Radio and IP

WSNs use the IEEE 802.15.4 radio for the communication. The standard provides 250 kbit/s data rate at 2.4 GHz in a 10-meter communication range. The maximum size of the physical layer payload is 127 bytes and there are long (64-bit) and short (16-bit) addressing modes in this protocol [5].

The Internet is based on the Internet Protocol and the original (and still active) version of this protocol was defined in RFC 791 in 1981 [6]. The protocol was basically renamed

to IPv4 when the IPv6 was defined in RFC 2460 in 1998 [7]. The main advantage of the IPv6 is, that it extends the length of the addresses from 32 bits to 128 bits. This address space is believed to be large enough for the following decades. The IP based WSN solutions target only the IPv6, but it is clearly not possible to simply use the pure IPv6 over the IEEE 802.15.4 radio. The most problematic part is that the maximum transfer unit (MTU) of the IPv6 is 1280 bytes, but the physical MTU of the IEEE 802.15.4 is only 127 bytes. In order to bridge the gap between the two protocols, the 6LoWPAN adaptation layer was introduced. Besides the fragmentation, this layer also provides solutions for header compressions [8].

Since this work concentrates on security protocols over the transport layer, the used IP solution should not influence the behavior. In practice, some client applications still lack of real IPv6 support and the IPv4 is not usable when the communication goes through the IEEE 802.15.4 radio. The used IP layer will be always noted for scenarios at the evaluation.

2.2 TCP and UDP

In this work the Transmission Control Protocol (TCP) comes into play only when the client communicates with the gateway device directly. The TCP is not commonly used in WSNs, because of its complexity, and performance reasons [9]. This protocol provides reliable stream based communication, which means that the parties maintain a state and use acknowledgments. The mechanisms of the TCP do not fit efficiently into the IEEE 802.15.4 protocol, as it is detailed for instance in [10], and because of this it is not used over the radio. The TCP related use-case is that the gateway runs an HTTP server over TCP, and the client is a simple web browser.

Researchers also argue about whether the User Datagram Protocol (UDP) is suitable for WSNs or not [10]. It is a connectionless protocol which operates with datagram sockets [11]. The messages are sent without verifying the readiness or the status of the receiver. It is unreliable, because the messages could get lost during the transmission, and the sender will never know about it. There are no acknowledgments and retransmissions in this protocol, but because of the mentioned characteristics it is extremely lightweight which is a key factor in WSNs, and this protocol can be used for the data communication over the radio too. There are several proposed transport layer solutions such as CODA (COngestion Detection and Avoidance) [12] or RMST (Reliable Multi-Segment Transport) [13]. The problem is that the traditional Internet will likely never support these protocols and this contradicts with the main goal of the Internet of Things, which is to move the sensor solutions towards the standard Internet.

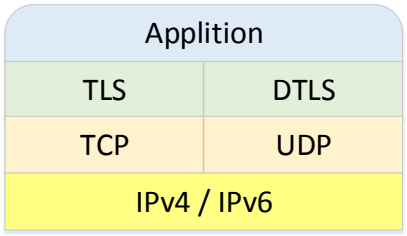

Figure 2.2: *The secured protocol stack*

2.3 Transport Layer Security

The TLS (Transport Layer Security) and the DTLS (Datagram Transport Layer Security) are cryptographic protocols which can provide secured communication over the Internet. As Figure 2.2 shows, the TLS runs over the reliable TCP protocol, and provides security for many well known web services (such as HTTPS). The DTLS is the pair of the TLS protocol and it provides approximately the same security features, however, it runs over the non-reliable UDP protocol. The DTLS is not used at as many places as the TLS, but in case of WSNs where the IP communication is based on UDP, it will be an important protocol in the near future.

TLS	DTLS
1.0[14]	-
1.1[15]	1.0[16]
1.2[17]	1.2[18]

Table 2.1: *TLS and DTLS protocol versions*

2.3.1 Protocol versions

Table 2.1 summarizes the different TLS and DTLS versions. As the table also indicates, the DTLS 1.0 protocol is equivalent with the TLS 1.1 and there is no DTLS 1.1. The original TLS 1.0 protocol is vulnerable against attacks and it is recommended to use the newer versions of the protocol. However, as always in case of the Internet, every protocol change is a long procedure. Table 2.2 shows that only the latest versions of the well-known web browsers supports the newer versions. Figure 2.3 summarizes the server side support for popular sites. All in all, a new implementation nowadays should support all versions of the protocols because of interoperability reasons.

Browser	First version with TLS 1.1 support	First version with TLS TLS 1.2 support
Firefox	19 (disabled by default)	24 (disabled by default)
Chrome	22	29
IE	8 (disabled by default)	8 (disabled by default)
Opera	14	16

Table 2.2: *Support of the TLS protocol in web browsers [19]*

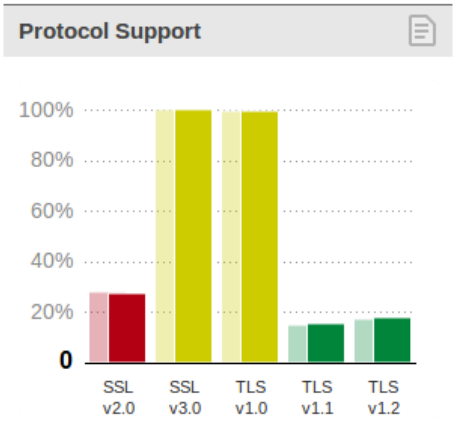

Figure 2.3: *Available TLS versions on popular sites (September 2, 2013 [20])*

2.3.2 General protocol description

The general description in this section is for both protocols, and only the differences are noted. The DTLS protocol was designed as a modification of the TLS, in order to handle the differences of the underling transport layer. Practically the result is that some of the TCP functionalities (which are missing from the UDP) must be implemented in the DTLS. These functions are: resend mechanism, packet duplication handling, sequence numbers, etc. This section omits some parts from the protocols which are not relevant for the current microcontroller based implementation.

The (D)TLS connections have an initial connection phase and a subsequent communication phase. The main goal of the connection phase (which is called handshake in these protocols), is to negotiate a common key which is then used to secure the data exchange in the communication phase. If we consider the traditional OSI network model, then the initial phase fits into the Session Layer and the connected phase fits into the Presentation Layer.

The (D)TLS is actually a collective name for four protocols as it is shown in Figure 2.4. All packets have to go through the Record Protocol which forwards the messages to the upper level. During the handshake, it forbids the communication with the application. When the connection is ready, it deals with the encryption/decryption and authentication of the messages in a transparent way from the application's viewpoint. Since the TCP provides reliable communication, the TLS handles its sequence numbers implicitly. For the DTLS, the sequence number is part of the header of the Record Protocol, and this protocol also validates it.

The Handshake Protocol, as the name implies, processes and sends the messages during the handshake. The asymmetric cryptographic operations also performed here. The goal of this protocol is to prepare the security parameters for the Record Protocol. The Change Cipher Spec Protocol is able to send only one empty message. This message indicates that the sender is going to send encrypted messages in the future. The Alert Protocol is used when something goes wrong. An alert can be `Fatal` or `Warning`. While a `Fatal` error immediately closes down the connection, a `Warning` is only an indicator for the other party.

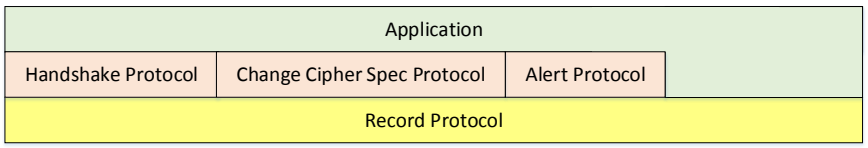

Figure 2.4: *(D)TLS protocols*

The (D)TLS can be used with various cipher suites. A cipher suite is a set of algorithms and it is negotiated in the first message exchange of the handshake. The related concrete algorithms will be detailed in Section 2.3.3. A cipher suite specifies the following parameters:

Asymmetric cryptographic algorithm This algorithm is used during the handshake to perform the key exchange and party authentication.

Symmetric cryptographic algorithm This algorithm is used to encrypt and decrypt the messages.

MAC algorithm The MAC (Massage Authentication Code) algorithm is used to authenticate the messages.

The handshake requires several messages from both sides, as it is shown in Figure 2.5. Some of these messages are optional, depending on the used cipher suite. The client always sends the Client Hello message to initiate the connection. This message contains the protocol version, client random, list of the supported cipher suites, possible compression algorithms and the implemented protocol extensions. The DTLS protocol defines an extra step immediately after the initial Client Hello (indicated with green in the figure). In order to prevent easy Denial of Service attacks, the server should send a Hello Verify Request message with a cookie inside. In order to continue the handshake, the client has to repeat its Client Hello with the same cookie. The server creates the state for the connection only if this cookie exchange was successful, and from this point the two protocols use the same flow.

The server selects a cipher suite from the client's list. In case there is no overlapping option, the server sends an error message immediately. If there is a common suite, the server sends

the Server Hello to the client which contains the protocol version, server random and the selected cipher suite, compression algorithm and extensions. The following messages (indicated with orange in the figure) are depending on the selected cipher suite which will be detailed in Section 2.3.3. The first of these contains the server's certificate in an ITU-T standard format (X.509). The second is the Server Key Exchange which contains additional information for the key negotiation, and it is always signed with the public key from the certificate. The server always sends the empty Server Hello Done message to indicate the end of the hello sequence.

The client verifies the server's identity (based on the certificate). If the server is identified, the client calculates the so-called pre-master secret – based on the collected information –, encrypts it with the server's public key and finally sends it in the Client Key Exchange message. The client then sends the Change Cipher Spec message to indicate that it is going to encrypt the following messages. Finally, it calculates the connection keys and sends the Finished message which contains the hash of all previous messages encrypted and authenticated with the negotiated algorithms.

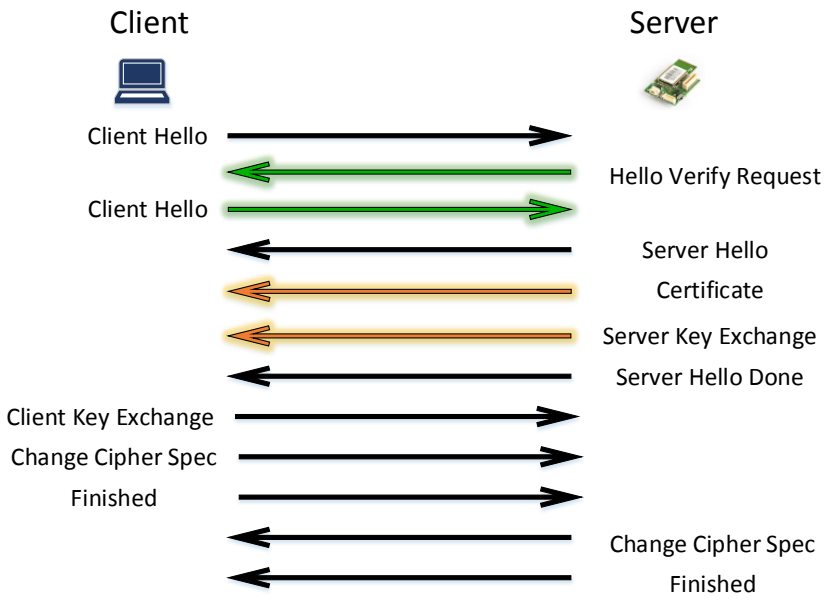

Figure 2.5: *(D)TLS normal handshake*

The server decrypts the payload of the Client Key Exchange and also creates the keys based on the pre-master secret. As the last step of the handshake, it sends the Change Cipher Spec and the Finished messages to the client.

The keys are created in two steps from the pre-master secret. The (D)TLS defines a PRF (Pseudo Random Function) which takes as input a secret, a seed, and a label and produces an output of arbitrary length. The PRF in these protocols is based on the HMAC construction [21]. An HMAC takes a key and other some data as input and creates a keyed hash as output. The TLS 1.0, TLS 1.1 and the DTLS 1.0 protocols use both the MD5 and

SHA-1 hash functions for the HMAC calculation, and the results are XOR-ed. The TLS 1.2 and DTLS 1.2 protocols use only the SHA-256 function. In brief, the PRF takes the input parameters and calls the HMAC function (in a chained fashion) as many times as it is required to get enough pseudo random bytes. The master secret is generated with this PRF from the pre-master secret. The protocols do not use the master secret directly, instead a key material is generated from the master secret again with the PRF. This enables to re-use the negotiated master secret and results in different keys for different purposes. The key material contains different keys for the client and the server and also for the cipher and the MAC algorithms. In same cases an initialization vector (IV) is also generated here.

The RFCs also define sessions and connections. A session is an ID and master secret pair which is stored by the server. The client can include a session ID into the Client Hello message. If the server stores the master secret for the provided ID, it immediately sends the Change Cipher Spec and the Finished messages after the Server Hello, as it is show in Figure 2.6. This way the server resumes the session and there is no key exchange needed. If the client does not provide a session ID or it cannot be identified, the server includes a new random session ID into the Server Hello, and performs the normal handshake. The new key material is generated from the master secret with the new client and server random numbers, which ensures, that it will be different from the previous ones.

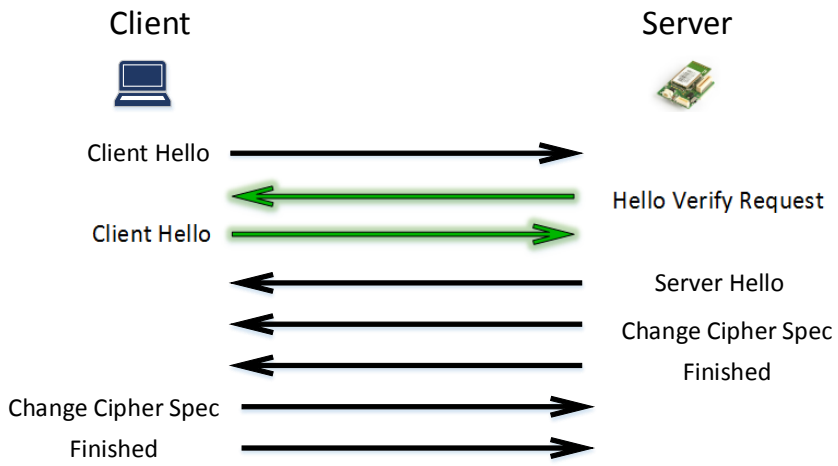

Figure 2.6: *Resume a (D)TLS session*

2.3.3 Cipher suites

The following section introduces the different related cipher suites. The cipher suites work in a 'MAC then encrypt' fashion, which means that the authentication tag is calculated with a hash algorithm, and then it is encrypted together with the plaintext. This section also describes some asymmetric and symmetric cryptographic solutions.

2.3.3.1 Asymmetric algorithms

The traditional cipher suites use the RSA algorithm for the key exchange. The RSA is based on the difficulty of factorization, and with the nowadays' computation power, it requires large primes for sufficient security. The long primes also result in large certificates (and long calculations). In case of a microcontroller implementation, the RSA calculations are extremely slow.

The elliptic curve cryptography (ECC) was originally suggested in 1985, but it has been widely used only since 2004–2005. The ECC is based on the algebraic structure of elliptic curves over finite fields. Instead of targeting the factorization problem, the ECC assumes that finding the discrete logarithm of a random elliptic curve element with respect to a publicly known base point is infeasible. The size of the used elliptic curve determines the difficulty of the problem, the length of the keys. Table 2.3 compares the key sizes of the RSA and ECC algorithms which enable the same security level. The smaller curves result in faster computation, but for instance web browsers do not support weaker curves than 256-bit.

RSA key size	ECC key size
1024 bits	160 bits
2048 bits	224 bits
3072 bits	256 bits
7640 bits	384 bits

Table 2.3: *RSA and ECC key sizes providing the same security level [22]*

In (D)TLS the elliptic curve cryptography is used in the Elliptic curve Diffie–Hellmann (ECDH) protocol to perform the key exchange. This protocol offers solutions to support any elliptic curve, but in practice the 'named curves' are used, which were standardized by the American NIST (National Institute of Standards and Technology) [23]. The public key of the server is always in the certificate. In case of the ECDH, the client encrypts the pre-master secret with this public key, and the server can decrypt it with its corresponding private key. At the server this requires 1 ECC calculation. The weakness of this protocol is that it does not provide perfect forward secrecy, meaning that if the server's private key is ever compromised, then all previous recorded communication can be decrypted. The ECDHE (ECDH Ephemeral) algorithm solves this problem with generating a new public/private key pair for each connection. In this algorithm, the server sends the new public key in the Server Key Exchange message, and its original public key from the certificate is only used to generate a signature on it with the ECDSA (Elliptic Curve Digital Signature Algorithm). All in all the ECDHE requires 3 ECC calculations on the server side (key generation, signature, decryption) [24].

Both the ECDH and the ECDHE based cipher suites have been added as valid choices for the (D)TLS. These are defined (with AES as a symmetric cipher) in RFC 4492, RFC 5269 and in RFC 5489.

It must be noted that the RFCs define the `TLS_RSA_WITH_AES_128_CBC_SHA` cipher suite as mandatory, but the cryptographic libraries and web browsers have been widely adopted the ECC based cipher suites, which enables the omission of the RSA. Since the code size is one of the key factors in this kind of development, it is not really an option to have the RSA just because it is mandatory by the RFC.

A simple solution which was basically defined in RFC 4279 for (D)TLS in constrained environments, is the Pre-Shared Key (PSK) algorithm [25]. In case of a PSK cipher suite, the client and the server must know the same secret and an identifier in order to establish the connection. The server can provide an 'identity hint' in the Server Key Exchange in order to help the client in the correct secret selection. The client sends its 'PSK identity' in the Client Key Exchange. This type of key exchange can be done without any real asymmetric cryptography, however the distribution of the keys can be a serious problem. These cipher suites can be used only in some limited scenarios. For example, if we consider a company which provides the sensors devices and the PC-side client applications as well, the PSK can be a good solution.

2.3.3.2 Symmetric ciphers

The earlier versions (before 1.0 and 1.1) of the (D)TLS protocol support block and stream ciphers. A stream cipher is initialized only once and then generates a (pseudo) random byte sequence which is XOR-ed with the plaintext. During the decryption, the same byte stream is XOR-ed with the ciphertext. It is clear that the cipher must be in the same state on both ends, which is a huge constraint for the DTLS protocol, since it is based on the unreliable UDP protocol. Because of this, the stream ciphers are banned from the DTLS protocol by the RFCs. The target of this work is an implementation which efficiently supports both the TLS and the DTLS, therefore there will not be any further discussion about stream ciphers.

The most traditional block cipher is the DES (Data Encryption Standard) which was first published in 1977. There are several attacks known nowadays against the DES, but its small key size (56 bits) is in itself cannot stand the brute force attack of the modern computers. As a solution, in 1998 the Triple DES was published which basically uses the DES three times. In parallel of the Triple DES, a new encryption standard the AES (Advanced Encryption Standard) was also created. As it was shown in papers like [26], the 3DES is slower than the AES. Furthermore, it uses shorter keys (112 bit vs. 128/192/256 bits) and also shorter blocks (64 bits vs. 128 bits). Regarding the interoperability, all of the traditional TLS implementations support the AES based cipher suites. The last important aspect is that some of the devices developed for WSNs, provides AES encryption engines in hardware. This decides the competition of the two ciphers without question.

The AES in itself is capable to encrypt and decrypt only one block (16 bytes). In order to use this cipher for longer messages, the combination of the blocks is needed which

is called as the 'mode' of the cipher. The most well-known mode is the CBC (Counter Block Chaining), which is shown in Figure 2.7. The idea of this mode is that the next plaintext block is always XOR-ed with the previous ciphertext block. For the first block, an Initialization Vector is used. The TLS 1.0 defines this IV as part of the key material which is updated with the last ciphertext block when after each processed message. This construction led to possible attacks and in other versions of the (D)TLS protocols, the IV is always a new random array and it is sent explicitly in the message.

The MAC is calculated with a hash function (such as SHA-1 or SHA-256) and, as the figure shows, it is copied after the plaintext blocks, and also encrypted. Because the AES takes always 16 bytes as input, a padding is required for the message. The last byte defines the length of the padding (without this byte), and the padding itself contains the same bytes as its length. For instance if the length is 92 bytes, then 4 bytes are required in this form: 03 03 03 03. The padding enables time-related side channel attacks against the (D)TLS protocols which will be detailed in Section 3.8.

The decryption of the ciphertext is similar to the encryption. Each ciphertext block is decrypted with the AES and then XOR-ed with the previous block. It must be noted that the AES requires the same type of steps for encryption and decryption, however, from the implementation point of view, there are important differences which will be examined in Section 3.6.

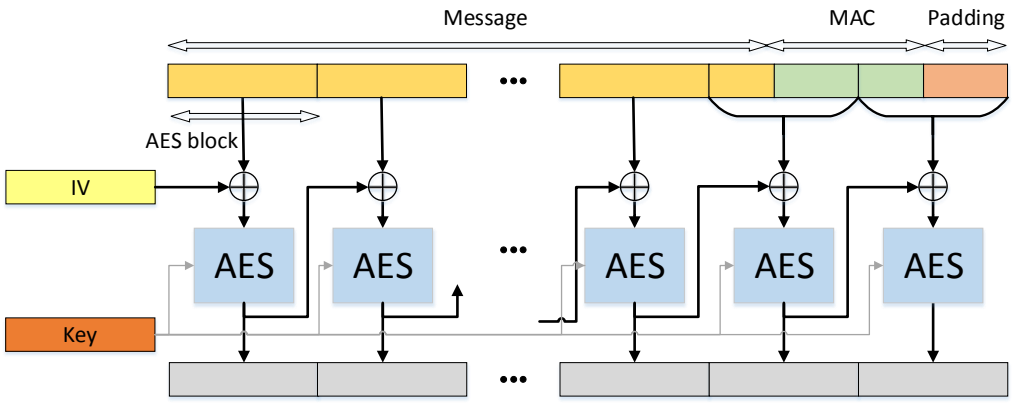

Figure 2.7: *(D)TLS AES encryption in CBC mode*

The 1.2 versions of the protocols define the AEAD (Authenticated Encryption with Associated Data) type of ciphers. The AEAD ciphers are based on block ciphers but provide authentication and encryption as one operation with one key. These are also 'modes' for the block ciphers at the end of the day, but the (D)TLS specifications define these in the separated AEAD category.

The CCM (Counter with CBC-MAC) is an authenticated encryption mode of the AES and it is shown in Figure 2.8. As the name implies, it uses the previously detailed CBC-mode for the MAC calculation. However, instead of generating the whole encrypted output, only

the bytes of the last ciphertext block are used as a MAC. The CCM mode can take two byte streams as input. One of the byte streams (the message itself) is authenticated and also encrypted and the second byte stream is only authenticated. For the MAC calculation both input streams are padded with zeros up to the 16-byte block length. Besides the data streams, the cipher also takes a nonce and a key, where all nonce – key pairs must be unique. This key is used for all AES operations in this cipher mode. The input nonce is used to form a special 16-byte-long 'nonce block' which also contains some flags and a counter part. As the figure shows, this nonce block is used as the input for the AES. The counter part of the nonce block starts from 1 and it is incremented after each step which results in a different AES output. This output is then XOR-ed with the plaintext blocks. For the CCM mode, the last block does not have to be padded, because the unnecessary bytes from the AES output can be simply dropped. The MAC is also encrypted like the normal data, but the value of the counter is 0 for that last block.

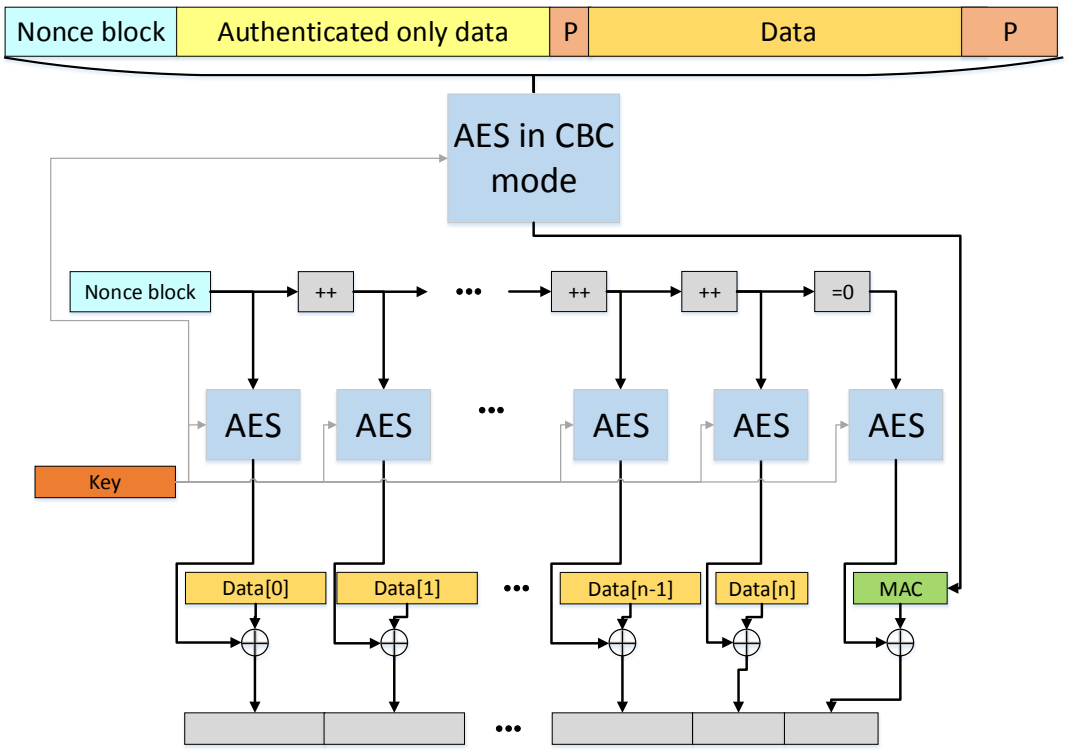

Figure 2.8: *(D)TLS AES in CCM mode – authenticated encryption*

All in all, the construction is a mixture of the block and stream ciphers, but can be used with DTLS as well. One important feature of this cipher mode is that the AES decryption operation is not used. The pseudo random stream can be generated in the same way for the decryption, and XOR-ed with the ciphertext. For the MAC validation, the same authenticated only data must be available.

The IEEE 802.15.4 standard in itself contains a security part for the communication. The standardized cipher is the AES-CCM* [3]. The CCM* is a minor variation of the CCM.

The difference is that, in contrast with the normal CCM, the CCM* mode can be used to encrypt only and authenticate only. However, in case of the (D)TLS both the encryption and authentication is required which results in full interoperability. This is the reason why the AES encryption is available in hardware on WSN devices.

Some cipher suites define the symmetric cipher as 'NULL'. This means that the messages are only authenticated but not encrypted. These suites are not used in general solutions but can be used in some special scenarios. For example, if we consider that a client gathers measurements from IP enabled sensor devices, the data itself can be possibly transmitted without encryption, but it must be authenticated in order to prevent modifications on the fly. Without encryption, the code size is smaller, and the device consumes less energy.

2.3.3.3 Possible combinations

The presented asymmetric and symmetric algorithms can be combined in several possible cipher suites. Regarding the AES, only the 128-bit version is considered. This subsection summarizes the target cipher suites of this work.

The following 4 cipher suites are suitable for interoperability with general applications, like web browsers. All current web browsers offer some of these suites in their Client Hello message. It seems that the browsers will move towards perfect forward secrecy, because for instance the Google Chrome's newest version (29) only offers the ECDHE suites.

- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA

The second group of cipher suites can be applied in special environments. For the first 5, the PSK is used which restricts the usage to cryptographic libraries and self-designed applications. The '8' at the end of the CCM suites defines the length of the MAC tag. The last suite which uses ECC with CCM is still in draft and not RFC state[1]. Because of this, it is only implemented in some cryptographic libraries. The draft does not define the ECDH suite, only the ECDHE. There are valid combinations for ECC and the NULL 'cipher' but these are not implemented in any popular library. The PSK can be also combined with digital signature algorithms in order to provide perfect forward secrecy, but instead of those, it is better to use the normal non-PSK suites.

- TLS_PSK_WITH_AES_128_CBC_SHA
- TLS_PSK_WITH_AES_128_CBC_SHA256

[1] http://tools.ietf.org/html/draft-mcgrew-tls-aes-ccm-ecc-07

- TLS_PSK_WITH_NULL_SHA
- TLS_PSK_WITH_NULL_SHA256
- TLS_PSK_WITH_AES_128_CCM_8
- TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8

2.4 Applications

In this work I consider two standard applications, an HTTP server and a CoAP server. There are many possible applications for a TLS and a DTLS protocol implementation, even in this constrained environment, since it is always possible to construct a new self-designed solution. The iSense implementation of the HTTP protocol provides a normal HTTP server's functionality. This server can use the SD card of the NET10 device as well. The server is normally running over the TCP protocol, but with proper interfaces it is possible to inject the TLS between the TCP and the HTTP server in a seamless way. The testing of this construction is quite trivial with commercial web browsers.

The CoAP (Constrained Application Protocol) is a specialized web transfer protocol for use in constrained environments. The first draft of the protocol was released in the summer of 2010 and the protocol has been changing deeply during the last 3 years. The actual version is the 18th draft, and now it is close the final version. Because of the significant changes, clients and servers which implement different versions of the draft may not be able to interoperate.

The protocol works in a request/response manner, it supports the discovery of resources and services. The well-known URI (Universal Resource Identifier) scheme was adopted for this protocol. Similarly to the HTTP, the CoAP supports GET, POST, PUT and DELETE methods. The CoAP resources can be used for quite different purposes:

- Query different sensor values with GET messages.
- Store data on the server with PUSH messages.
- Control peripherals (for example: switch LEDs on and off)

The CoAP runs over the UDP protocol, and similarly to the HTTP server, it is possible to use the DTLS in order to provide security. The testing of a CoAP over DTLS is, more problematic since most of the client implementations do not support the DTLS by default.

2.5 Existing solutions

There are different existing implementations of the TLS and DTLS protocols for wireless sensor networks. As this section will show, the provided cipher suites are quite limited for these solutions.

The Contiki is one of the most popular operating systems for WSNs. Currently, there are two different public implementations exist. The first is called as TinyDTLS and it was created by Olaf Bergmann [27]. This implementation contains a simple DTLS server, with support for the `TLS_PSK_WITH_AES_128_CCM_8` cipher suite. The second implementation was created by Vladislav Perelman as his master thesis project [28]. This implementation contains both the TLS and the DTLS protocols, but it is not designed to re-use the common parts, which results in duplicated functionalities. This implementation also supports only the `TLS_PSK_WITH_AES_128_CCM_8` cipher suite for both protocols.

The TinyOS is another operating system for WSNs, written in the nesC language. There is also an existing DTLS implementation for this system created by Thomas Kothmayr [29]. In his work he was using a special TPM (Trusted Platform Module) device which can run RSA and SHA-1 operations. With this extension it is possible to run the heavy-weighted RSA in this constrained environment, and use the `TLS_RSA_WITH_AES_128_CBC_SHA` cipher suite which provides interoperability with all available PC side applications.

Chapter 3

Own contribution

After the overview of the different parts of a possible TLS and DTLS implementation for WSNs, the first section of this chapter summarizes the targets and goals of this work. The following two sections introduce the proposed and implemented main components and interfaces. The remaining sections highlight the more important contributions.

3.1 Targets and goals

As it was described in the introduction, the target operating system is the iSense, and the target hardware platforms are the Jennic 5148 based iSense devices. Before my work, there was an existing implementation of the server side of the TLS protocol for iSense, created by Michael A. Strebel as his Master thesis project [30]. I started to work with his code which was not merged into the main iSense repository nor part of the main IP stack and namespace. It was obvious that the coding part was finished in rush, since it contained many dirty and illogical solutions. Also most of the classes were not commented at all. Unlike the other listed solutions in Section 2.5, his target cipher suites were:

- `TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256`
- `TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA`

He implemented his own ECC library, but the performance of this solution was poor, the connection setup time was around 21 seconds. As an AES engine, he adopted the AES implementation of the Wiselib WSN algorithm library [1]. As it will we detailed in Section 3.6, I managed to increase the speed of this algorithm significantly. The following list collects the targets of this work.

Clean and fix the existing code Fix the existing implementation, providing a properly working TLS server.

[1] http://www.wiselib.org/

The DTLS protocol Implement the DTLS protocol with re-using the common parts of the TLS.

New cipher suites Adopt all 10 cipher suites (8 new) which were listed in Section 2.3.3.3. This also includes the AES-CCM cipher mode, the Ephemeral key exchange and the Pre-Shared Keys.

Performance improvements Improve the performance of the ECC and the AES.

Evaluation Test the interoperability of the application with various applications. Create performance related measurements.

One of the key ideas of this work is to provide a scalable solution for the actually needed purposes. I implemented the DTLS protocol with this idea in mind, making it possible to run the TLS and the DTLS together with as less overhead as possible. The different cryptographic parts are also scalable.

3.2 Socket and transport layer interfaces

From the application point of view, the underlying security layer has to be transparent. This can be achieved with common interfaces for the transport and the security layers as it is shown in Figure 3.1. This way if we consider for instance the CoAP server, it stores a `UDPInterface` and a `UDPSocketInterface` object, which makes the switch really simple from the non-secured communication to the secured one and vice versa.

Since the major parts of the TLS and the DTLS are identical, most of the code can be shared between the two protocols. To hide the differences between the TCP and the UDP API from the security protocols viewpoint, a third `SecureSocketInterface` is also necessary.

Figure 3.1 also implies one of the major structural differences in the TLS and the DTLS implementations. If an application creates a TCP socket in listening mode, the TCP opens a new socket for each connection and the HTTP server creates HTTP connections for these. This mechanism enables a simple scenario with one TLS socket for each new connection. On the contrary, a listening UDP socket does not create a new socket, instead it provides the port and host parameters for each incoming datagram. This mechanism is correct, and resource friendly for the pure UDP communication, nevertheless it causes problems for the secure communication. The connections must be distinguished and this is the reason why the DTLS requires the DTLS Connection entity. The only task of the DTLS Socket is to find the right connection for the actual data.

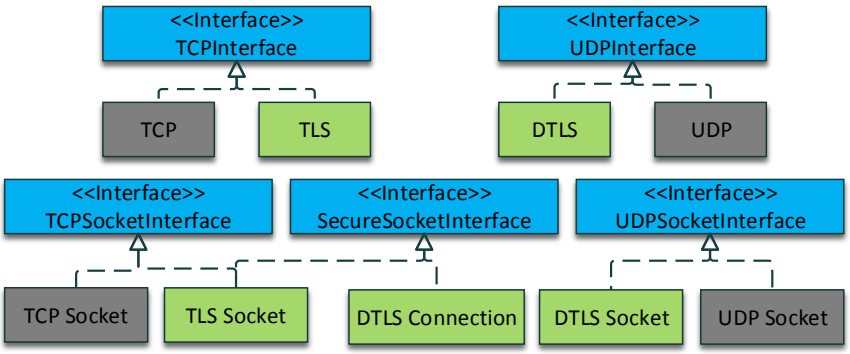

Figure 3.1: *Interfaces for the sockets and transport layers*

3.3 Implementation details of the TLS and DTLS

The implementation consists of several different elements which can be divided into 3 major groups. Figure 3.2 indicates these groups with different colors:

- The cryptographic parts (light).
- The different layers and sockets which are defined by the TLS and DTLS (mid).
- Additional classes for different state parameters (dark).

The interfaces for the layers and sockets have been already described in the previous section. This figure shows the connections with the rest of the elements.

When the CoAP server is started, it waits for a UDP Interface and opens a listening socket, which is in this case a DTLS Socket. When a client initiates the handshake, the DTLS Socket creates the DTLS Connection, the DTLS protocols (Record, Handshake, and Alert) and the connection states. The mechanism is similar for the TLS, but instead of the DTLS Connection, a new TLS Socket is created.

The Record Protocol is responsible for the encryption/decryption and the authentication of the messages. It uses state objects to store the connection parameters. The active state is always the 'current state' which initially does not define any security. During the handshake, the Handshake Protocol prepares the 'pending state' for the secure connection. When the connection parameters are ready, the 'pending state' takes the place of the 'current state'.

The cryptographic part contains an asymmetric and a symmetric cipher and different hash functions. The symmetric cipher is the improved version of the Wiselib's AES, which will be detailed in Section 3.6. For the ECC based key exchange, a special library is ported to the iSense platform (MicroECC). The library will be described in Section 3.4 together with the Pre-Shared-Key manager. The MD5 hash function is only necessary for TLS 1.0.

The SHA and SHA256 are used for the handshake and inside the HMAC for the message authentication in connected status.

The last element in the figure is the TLS Session Manager, which is also a common object for the TLS and the DTLS. The session management will be detailed in Section 3.5.

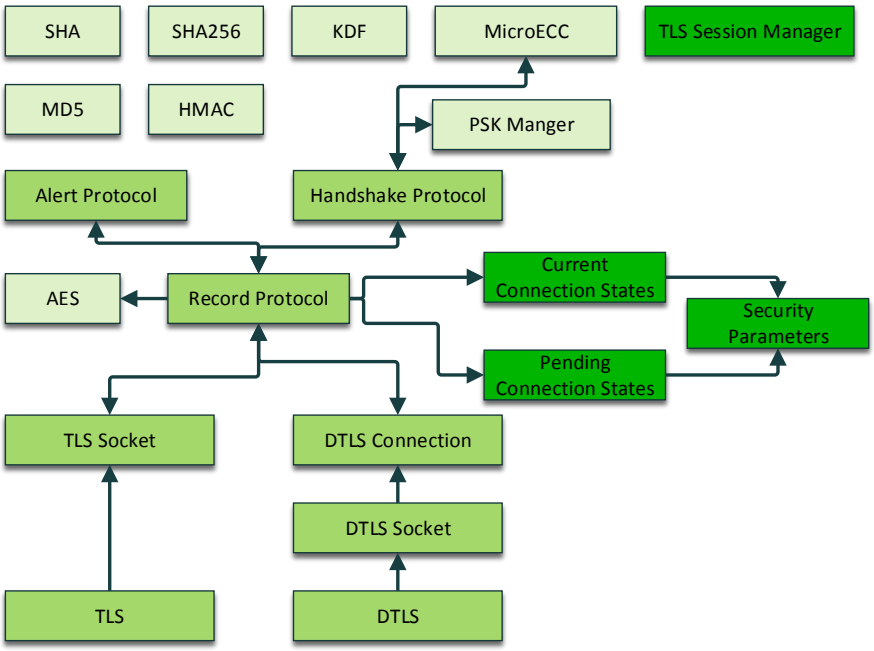

Figure 3.2: *Related classes*

3.4 Solution for the key exchange

The implementation of the ECC support (by Michael A. Strebel) was not fast enough for the normal usage of the cryptographic protocols, especially in case of the HTTP server it is really annoying to wait 21 seconds for a web page. The applied solution is the modified version of the MicroECC library which has an open source version (available on GitHub [2]) directly for 32-bit microprocessor architectures [31]. The library was originally developed for an ARM based controller, and contains also assembly codes for that platform. This ECC implementation supports both the key-generation, ECDH and ECDSA algorithms. While for the ECDH based cipher suites only the ECDH algorithm and the related functions are needed, the ECDHE cipher suites require the key-generation and the ECDSA as well.

There were two main modifications needed for the library for the use under iSense. The first, and more obvious part was the porting from C to the object oriented C++. This

[2] https://github.com/kmackay/micro-ecc

also implied many modifications in the original C macros and global variables. The second, and more problematic part was the modification of the functions in order to fulfill the requirements of an event-driven operating system. The iSense follows the so called 'run-to-completion' way of programming. This means that while a task is running, only the interrupt service routines can be served. Other task cannot get the processor since there is no preemption. This is not a really big problem for general algorithms, nevertheless, it must be considered for long running codes, such as the ECC library. The MicroECC supports 4 different named curves as it is shown in Table 3.1 along with the time which is needed for one calculation. The calculation obviously cannot block the processor for seconds, and it must be divided into smaller tasks. These tasks run for approximately 27 ms in this implementation. For the ephemeral key exchange, I also extended the library with ASN.1 support, since it is needed for the Server Key Exchange message.

Curve name	iSense [s]	Original ARM [s]
secp128r1	1	0.06
secp192r1	3	0.117
secp256r1	7	0.310
secp384r1	22	0.910

Table 3.1: *MicroECC calculation times*

The last column of Table 3.1 also contains the time which was measured by the authors of the original library. The original values are from the GitHub's wiki page, but I scaled them for the processor speed of the iSense (48 MHz originally, 32 MHz for iSense). The difference is significant, but it must be taken into account that the ARM Cortex-M0, which was used by the authors, provides a 'hardware single-cycle (32x32) multiply option'[3] and the library also contains some ARM specific assembly parts for the critical calculations. The mechanism which divides the long running task into quicker tasks also implies latency for the calculation on the iSense.

Since the trend of the time differences is approximately the same for the two cases, the assembly versions for the critical parts (operands for very large integers) can be considered as a possible further improvement.

For the Pre-Shared Key cipher suites, I implemented a manager entity which stores and handles the registered keys. Along with the key, an 'identity' is also stored as a character array. When the client sends the Client Key Exchange, the manager selects the correct key based on the provided identity string. There is only one global manager object, which enables the easy access from both the TLS and the DTLS codes.

3.5 Session management

As it was detailed in Section 2.3.2, the negotiated sessions can be stored and resumed. It is up to the client whether it includes the session ID of the previous session into its Client

[3] http://www.arm.com/products/processors/cortex-m/cortex-m0.php

Hello message or not. In case of the TLS based communication, a web browser stores and uses the sessions while it is running. For these applications the PSK cipher suites cannot be used, which implies that it is important to support the session handling on the server side. It improves the user experience (fast loading of pages) and it also saves the resources of the device. On the contrary, when the DTLS (and the CoAP) is used, the session handling might be pointless, since most of the client applications will not send the session ID in the Client Hello message. As many other parts of the implementation, the session management can be switched off as well, saving approximately 1400 bytes of code size and 92 bytes of memory for each stored session. As for the PSK manager, there is one global instance for the Session Manager, which enables easy access in any situation.

3.6 Accelerating the AES decryption

All of the secured cipher suites use the AES block cipher in different modes. The AES itself is from the Wiselib open source algorithm library. It handles the AES encryption and decryption with 128-bit keys. The AES uses 4 operations:

SubBytes is the only non-linear step of the cipher, where each byte is replaced with another according to a lookup table, called as S-box.

ShiftRows is a byte transposition that cyclically shifts the rows of the state over different offsets.

MixColumns is a permutation operating on the state column by column.

AddRoundKey is a transformation, where the state is modified by combining it with a round key with the bitwise XOR operation.

The cipher is symmetric, it uses the inverse versions of the operations in reverse order for the decryption (InvSubBytes, InvShiftRows, InvMixColumns, the AddRoundKey is the same) [32].

Operation	Encrypt [cycles]	Decrypt (Inv) [cycles]
SubBytes	85	84
ShiftRows	38	36
MixColumns	146	*5138*
AddRoundKey	82	82
Full AES block	2885	*48237*

Table 3.2: *Original Wiselib AES run-time analysis*

After the initial connection speed tests, I measured the calculation times for the AES library itself. Table 3.2 shows the measured values for the different operations at 32 MHz processor speed. The unit is processor cycle, because the Jennic microcontroller provides a Tick timer

for precise measurements. It is really obvious that the bottleneck of the algorithm is the InvMixColumns operation, which runs 35 times slower than the MixColumns.

In the MixColumns step, each column is treated as a polynomial over $GF(2^8)$. For the encryption, each column is multiplied by a fixed polynomial ($c(x) = 3x^3 + x^2 + x + 2$), modulo $x^4 + 1$. This is (practically) the following matrix multiplication for a column 'a' (in $GF(2^8)$):

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

In this $GF(2^8)$ field, the multiplication by 1 means no change, multiplication by 2 means shifting to the left. The multiplication by 3 can be calculated as 'multiplication by 2 plus the original value'. In $GF(2^8)$ the addition is the XOR operation, which means that multiplication by 3 is shifting and then performing an XOR operation with the initial (unshifted) value.

After each shift, it must be checked whether the value is bigger than 0xFF, because in this case an additional XOR operation with 0x1B (it is actually 0x11B but we are working on bytes) is required.

The InvMixColumns is an operation for the same purpose, the difference is that it uses the inverse of $c(x)$, which is $c^{-1}(x) = 11x^3 + 13x^2 + 9x + 14$. It looks like this in the matrix multiplication form:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

In the Wiselib's AES, the MixColumns operation is implemented in a much more efficient way, than the InvMixColumns operation. The MixColumns uses a pre-processor macro for the multiplication by 2, and calculates the multiplication by 3 with an XOR operation. For the InvMixColumns function this part is not that easy, since there are no simple rules for multiplication by 9, 11, 13 or 14. The Wiselib's InvMixColumns uses a general function for $GF(2^8)$ multiplication (this is the xtimes function in [4]). This function is called 16 times in an InvMixColumns calculation and it runs a loop with 8 cycles inside.

The fastest solution for the InvMixColumns operation can be implemented with look-up tables. In this case there would be 4 tables containing all possible multiplication results.

[4] https://github.com/ibr-alg/wiselib/blob/master/wiselib.testing/algorithms/crypto/aes.h

This means that 1024 bytes would have to be stored, but since the code-size is a key factor, this solution is not optimal.

The idea for the optimization is to use the basic operations (multiplication by 2 and the addition) more times after each other to calculate the required values. For instance, the multiplication by 9 can be calculated as: $(((x * 2) * 2) * 2) + x$. Figure 3.3 shows the calculation sequences for each number. As the figure also implies, because all numbers have to be multiplied by 9, 11, 13 and 14, this can be done in a tree-like fashion with re-using the partial results.

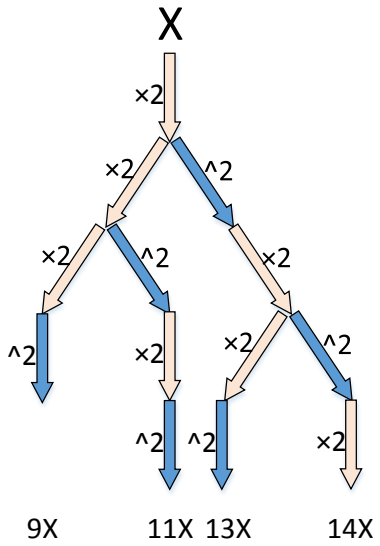

Figure 3.3: *AES decryption – InvMixColumns*

I implemented this solution with pre-processor macros which results 200 bytes larger total code size, but the gain in the calculation time compensates it greatly. Table 3.3 shows the measurement results after the optimization. Comparing the time required for the decryption of 1 AES block, the optimized version works approximately 8 times faster.

The Jennic microcontroller supports the CCM* mode of the AES in hardware. Since the CCM* requires only the encryption part of the AES, the decryption is not implemented. The Jennic API defines functions for the pure AES, which can be used to accelerate the encryption. Table 3.3 shows the time for the hardware based AES encryption. This method is also approximately 8 times faster than the original. In addition, this also saves code-size, because the functions and arrays for the software based encryption are not needed.

3.7 The AES modes

Now, that the AES is reasonably fast enough, the different modes of the block cipher can be introduced. It is a simple task to implement the CBC mode for the AES, since it requires

Operation	SW Encrypt [cycles]	HW Encrypt [cycles]	Decrypt (Inv) [cycles]
SubBytes	85	-	84
ShiftRows	38	-	36
MixColumns	146	-	*484*
AddRoundKey	82	-	82
Full AES block	2885	*366*	*5905*

Table 3.3: *Final Wiselib AES run-time analysis*

only the XOR calculation for buffers. One buffer is 16-byte-long in this case, and the speed of this part can be accelerated with unrolled version of the loop. This unroll can be used as a feature, but it can be disabled as well since it causes some code-size overhead.

I originally started to investigate the use of the CCM mode of operation because the Jennic device's AES co-processor supports this mode, since it is required by the 802.15.4 standard. The hardware API provides the following function for encryption and decryption:

```
PUBLIC bool_t bACI_CCMstar(
        tsReg128 *psKeyData,            //Special type with the key
        bool_t bLoadKey,                //If this is false, the key can be NULL
        uint8 u8AESmode,                //Encrypt or Decrypt
        uint8 u8M,                      //Length of the MAC
        uint8 u8alength,                //Length of the authenticated only data
        int8 u8mlength,                 //Length of the normal input
        tsReg128 *psNonce,              //Special type with the nonce
        uint8 *pau8authenticationData,  //Pointer to the authenticated only data
        uint8 *pau8inputData,           //Pointer to the input
        uint8 *pau8outputData,          //Pointer to the output
        uint8 *pau8checksumData,        //Pointer to the buffer for the MAC
        bool_t *pbChecksumVerify);      //Result of the MAC verification
```

The function takes several input parameters which are detailed in the comments. The important point is that it does not take the length for the nonce. Before the explanation of this, the 'nonce block' must be examined. Figure 2.8 showed the CCM mode of operation and the 'nonce block' is used both for the authentication and the encryption. As Figure 3.4 shows, the first byte of the block contains some flags. The initial bit is always 0. For authentication, the 'A'-bit is 1 if there is authenticated only data, the M field encodes the number of bytes in the authentication field and the L encodes the length of the Length part of this block. This L determines the length of the nonce as well, because the nonce occupies the remaining bytes. For the encryption (and decryption) the A and M is 0, and the L specifies the length of the Counter field (same as the Length, but it starts from 1 and it is incremented after each block).

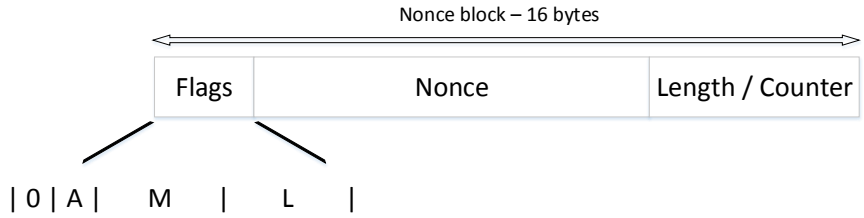

Figure 3.4: *The Nonce block of the AES in CCM mode*

The CCM mode for TLS is defined in RFC 6655 [33] and it declares the nonce for fix 12 bytes, which means that the Length/Counter field should be 3-byes-long. The Jennic API does not take the length of the nonce as a parameter, and there are no notes about this in the documentation. After some black-box like testing, it turned out that it always assumes that the nonce is 13 bytes long. This problem in itself can be solved with an additional 0 for the nonce, but the value of the L field cannot be changed in the flags byte. Because this is an input for the AES, this completely mixes-up the output. All in all, this simple limitation makes the use of the hardware based CCM impossible.

During the testing of the co-processor, it also turned out that it cannot work with data which is larger than 128 bytes, the API call simply returns with an error. This makes sense in case of the 802.15.4 radio, where this is the MTU for the packets. However, since the operation is the same for each AES block again and again, it is not clear why it is restricted to 128 bytes.

Despite the listed hardware limitations, the CCM mode still has the nice feature that it does not require the slow AES decryption. The iSense is compilable for a platform called XMega as well. Since it does not provide the CCM mode, it was implemented in software. I adopted this implementation for the CCM based cipher suites. I also optimized the way how it processes the input data because it was duplicated unnecessarily at a certain point.

3.8 Attacks against the TLS and the DTLS

During the years there were different published attacks against the transport layer security protocols. In general, the security protocols and the implementations are improved based on the published attacks. The different methods can be divided into two parts. There are papers about theoretical and practically exploitable problems. However, often it is only a question of time when a theoretical problem can be use in a practical scenario.

An up to date paper (published in 2013) from Christopher Meyer and Jörg Schwenk collects many TLS attack related papers [34]. As the TLS is the successor of the SSL protocol, many discovered vulnerabilities were solved, when the TLS 1.0 was introduced. The most important weakness of the TLS 1.0 is the IV handling in CBC mode. The problem is that the IV is only random for the first message, and then the last block of the previous ciphertext is used as an IV. This problem was published in 2004 [35] and in 2006 [36] by Gregory

Bard. Finally this attack was deployed in the B.E.A.S.T. tool, and published in [37]. This tool is able to perform even full message decryption. There is no implementation related fix for this problem since it is inside the protocol itself. Due to this massive vulnerability, migration to TLS Version 1.1 has been recommended by IETF.

Some published attacks are not related to this work because the exploited parts are not part of the implementation. One of these attacks uses the compression algorithm to leak information about the plaintext [38]. The current implementation does not use any compression and because of code and memory limitations it is not expected that it will be added in the future. The current version does not include the re-negotiation feature of the protocol which can be attacked as it is published in [39].

The implementations of the cryptographic primitives easily enable side channel attacks. For the TLS the most important side channel attacks are based on time difference measurements. The current wireless sensor networks scenario has its strengths and weaknesses against the timing attacks. If the communication is going over the low-power radio, the measurement of the time differences gets extremely hard because the jitter (caused by the radio) is significant. Because the code runs at low processor speeds, the timing differences can be larger, however, since generally these attacks require numerous connections, the slow connection setup compensates the gain on the timing differences.

The AES in CBC mode uses a 'MAC (then PAD) then encrypt' procedure. This construction can easily leak information. This timing problem is noted even in the RFCs (from TLS 1.2 RFC [17]):

> 'Canvel et al. [40] have demonstrated a timing attack on CBC padding based on the time required to compute the MAC. In order to defend against this attack, implementations MUST ensure that record processing time is essentially the same whether or not the padding is correct. In general, the best way to do this is to compute the MAC even if the padding is incorrect, and only then reject the packet. For instance, if the pad appears to be incorrect, the implementation might assume a zero-length pad and then compute the MAC. This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal.'

The most up do date study was published in February of 2013, as 'Lucky Thirteen: Breaking the TLS and DTLS Record Protocols' [41]. The authors demonstrate that even the 'small timing channel' (which is mentioned in the RFC), can be used to break the security of the TLS and DTLS. The authors propose an algorithm which reduces the timing channel, and I implemented this algorithm in this work. It must be noted that this problem only effects the CBC mode of operation.

The ‘quality’ of the random numbers is critical for security algorithms. However, this paper focuses on the protocols and the cipher suites. For the random number generation, the built in pseudo random number generator of the iSense was used. This is a linear congruential generator which is seeded with a number based on the current internal voltage multiplied by the internal temperature of the device. Since these are basically numbers from the analog to digital converter, the least significant bits are fluctuating. This generator should be changed to a cryptographically secure pseudo-random number generator before any real-life deployment.

Chapter 4

Evaluation

This chapter provides a detailed evaluation for the implemented TLS and DTLS protocols. The first section details the interoperability capabilities. The next section shows the scenarios which were used for the testing. Finally, the third part contains the results, namely: code size, heap usage, handshake time and round trip time.

4.1 Interoperability with other tools and implementations

I tested the implemented protocols in two different ways. The first approach contains tests with different cryptographic libraries. These can be used to test the interoperability and to evaluate the performance of the protocols and the different cipher suites. The other method of testing involves the real client applications (for instance web browsers). These tests produce more 'go – no go' like results.

There are many open source libraries available nowadays. The different solutions provide different cryptographic algorithms. The most commonly used library is the OpenSSL[1], however, interestingly it does not support the DTLS 1.2 protocol. It also lacks of IPv6 support by default, but there are some patches available which can be used to fix this issue.

The OpenSSL is embedded into several different products. The Python language's SSL library also uses the OpenSSL. The Python SSL API handles the IPv6 addresses but it does not provide DTLS related functions. Because of this, I was using a solution called PyDTLS[2] which patches the Python SSL to provide DTLS support. In this work, the different time related evaluations were created with Python scripts with an 'echo server' on the iSense side.

Another cryptographic solution is the CyaSSL[3] (yet another SSL). This library has several modules which must be enabled when it is configured. It supports IPv4 and IPv6 and all

[1] http://www.openssl.org/
[2] https://github.com/rbit/pydtls
[3] http://www.yassl.com/

TLS and DTLS versions. This library is smaller in general than the OpenSSL and this also results in simple and more easily understandable code. An important feature in this SSL implementation is that it supports the AES-CCM based cipher suites, even the ones with ECC key exchange which are only in draft state at the moment. The CyaSSL was used to evaluate the AEC-CCM cipher mode and the PSK key exchange.

The second approach for testing uses specified client applications, namely a web browser for the TLS and a CoAP client for the DTLS. The HTTP server over the TLS protocol can be used with any normal web browser. However, the performance depends on the actually used browser. The reason is for instance that the Google Chrome web browser opens two sockets immediately when it starts to load the web page. It means that the microcontroller has to run two ECC calculations in parallel. The best results can be achieved with the Firefox which opens only one socket and even the supported cipher suites can be influenced.

The testing of the secured CoAP is not as easy as the secured HTTP. There are several CoAP client implementations, but the DTLS support is really rare. I used two solutions to verify the interoperability capabilities. The first one uses the previously mentioned PyDTLS python library for the DTLS support. For the CoAP part, I managed to adopt the client tool from the webiopi library [4]. This library is originally for the Raspberry Pi platform, but with some slight modifications it can be used on a PC with the PyDTLS library.

The second solution is the CoAP client of the Californium project[5]. This is a JAVA based CoAP framework created by researchers from the ETH Zürich. The Californium contains an internal DTLS implementation created by Stefan Jucker [42]. The CoAP client from this library can be also used to communicate with the secured CoAP server.

All in all, it is possible to use the iSense (D)TLS implementation with various client applications. The following parts of this chapter show the performance of the code in different scenarios.

4.2 Experiment setup

The speed of the implementation will be evaluated for two parts, for the handshake and for the communication itself (as round-trip-time). Figure 2.1 showed the two setups for these measurements. In both setups the tester computer is connected to the gateway node through a switch. In the first scenario the gateway runs a (D)TLS echo server. In the second setup the gateway only plays a routing role, and forwards the traffic to another device via the radio interface which runs the echo server. For the radio based communication only the UDP based communication is evaluated.

The TLS and DTLS implementations are completely independent from the IP layer. In order to keep the results consistent the IPv6 was used for all experiments.

[4] http://code.google.com/p/webiopi/
[5] http://people.inf.ethz.ch/mkovatsc/californium.php

4.3 Experiment results

4.3.1 Code Size

Table 4.1 shows the sizes of the different elements in the three main configurations. The sizes for the Handshake and Record protocols show the differences in the required functions. These code blocks are required also when both protocols are active, which results in slightly bigger code size. The heavy part of the implementation is obviously the group of the cryptographic primitives. The size for the ECC library also includes the pre-loaded certificate. This is a self-signed certificate for testing purposes, however, it can be easily changed to a normal one. The ECDHE support was disabled for this measurement and the support for that adds 2kB extra size for the final result in all configurations.

The main point in the table is that because of the shared code of the TLS and DTLS, the support of both protocols only increases the code size by 10-15%. Compared to the other presented implementations [27] [28] [29], on the one hand, this saves a significant amount of code-memory when both protocols are in operation. On the other hand, this approach makes much easier the integration of new cryptographic elements.

The size of the security part can be decreased significantly with the Pre-Shared Key exchange algorithm, since in this case the MicroECC library can be elided, and only a few hundred bytes are needed in exchange for the PSK manager class.

Element	**DTLS**	**TLS**	**TLS & DTLS**
AlertProtocol	0.3 kB	0.3 kB	0.3 kB
HandshakeProtocol	4.3 kB	3.5 kB	4.5 kB
RecordProtocol	2.4 kB	2.5 kB	2.9 kB
TLS		1.7 kB	1.7 kB
DTLS	2.1 kB		2.1 kB
Sessions & states	0.6 kB	0.6 kB	0.6 kB
KDF & HMAC	1.4 kB	1.4 kB	1.4 kB
MD5	1.3 kB	1.3 kB	1.3 kB
SHA	0.9 kB	0.9 kB	0.9 kB
SHA256	1.1 kB	1.1 kB	1.1 kB
AES	3.1 kB	3.1 kB	3.1 kB
MicroECC + Certificate	5.0 kB	5.0 kB	5.0 kB
Total	22.5 kB	21.4 kB	24.9 kB

Table 4.1: *TLS and DTLS code size*

Figure 4.1 shows the impact of the different ciphers for the code size. The baseline for the bar chart is the NULL cipher, since it does not use any encryption/decryption. For the AES in CBC mode, the AES block cipher bears with the largest portion, the CBC mode in itself is nearly negligible. This mode uses the hardware encryption and the software decryption. The CCM mode requires more code-space than the CBC mode, but it provides authenticated encryption. For the true comparison, one of the hash algorithms should be

added to the CBC mode, and in this case the two bars would be similar. However, the situation is not this clear since the hash functions are used at other places (for instance in the PRF algorithm), which means that these cannot be elided for the CCM mode. For the CCM mode the AES block decryption is not used, but the mode in itself is more complicated. The last bar shows the two modes together which basically means that the AES block decryption and some extra 'if conditions' are added to the code.

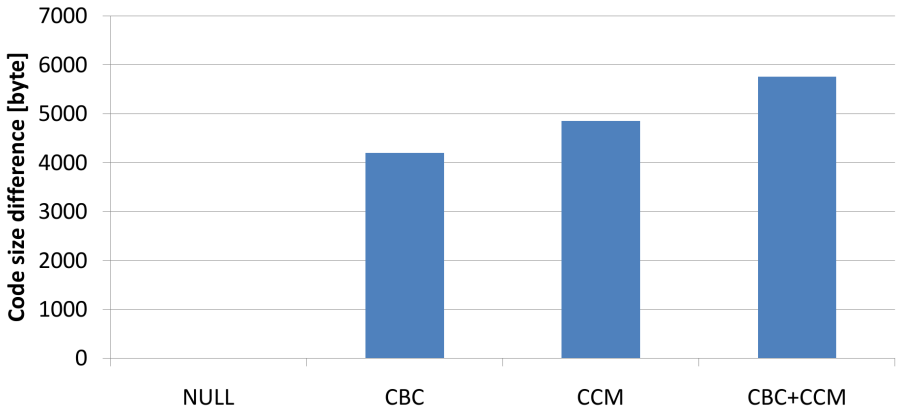

Figure 4.1: *Code size difference for the cipher algorithms*

Figure 4.2 summarizes the code sizes for some different secure protocol stack configurations which can be used in real-life. The network layer is IPv6 for 4 cases and IPv4 for the last bar, showing the complexity difference between the two protocols. The dark-gray parts are the TCP and UDP Protocols, the mid-gray ones are the TLS for HTTPS and DTLS for CoAPS. At the application level (light-gray) I compared the HTTP server and the CoAP server implementations with one simple resource for each.

CoAPS – General This protocol stack uses the same installation as the ones for HTTPS, except the session handling. Namely the ECDH key exchange and the AES in CBC mode which makes in easily interoperable with CoAP clients running over any cryptographic library.

CoAPS – Traditional This setup is 'Traditional' because it uses the PSK key exchange with AES in CCM mode and this combination is used in other CoAP over DTLS implementations for WSNs.

CoAPS – Minimal This case illustrates the smallest possible security layer with the PSK key exchange and the NULL cipher. This stack can be used to authenticate the data, such as measurements from the server to the client.

HTTPS – IPv6 This version uses the ECDH and the AES in CBC mode, making it interoperable with web browsers. It also includes the session handling. As it can be seen, the TCP is significantly larger than the UDP.

HTTPS – IPv4 This version uses the IPv4 which is smaller with approximately 10 kB than the IPv6. This leaves enough code memory for a complex application too. The

layers above the network layer are the same as in the previous case.

The figure shows the fact that the reliability functions at the datagram communication stack are moved from the transport layer to the security and application layers (the UDP is smaller than the TCP, but the DTLS is larger than the TLS and the CoAP is larger than the HTTP). However, the summarized size-gain between the 'CoAPS – General' and the 'HTTPS – IPv6' is still 12%.

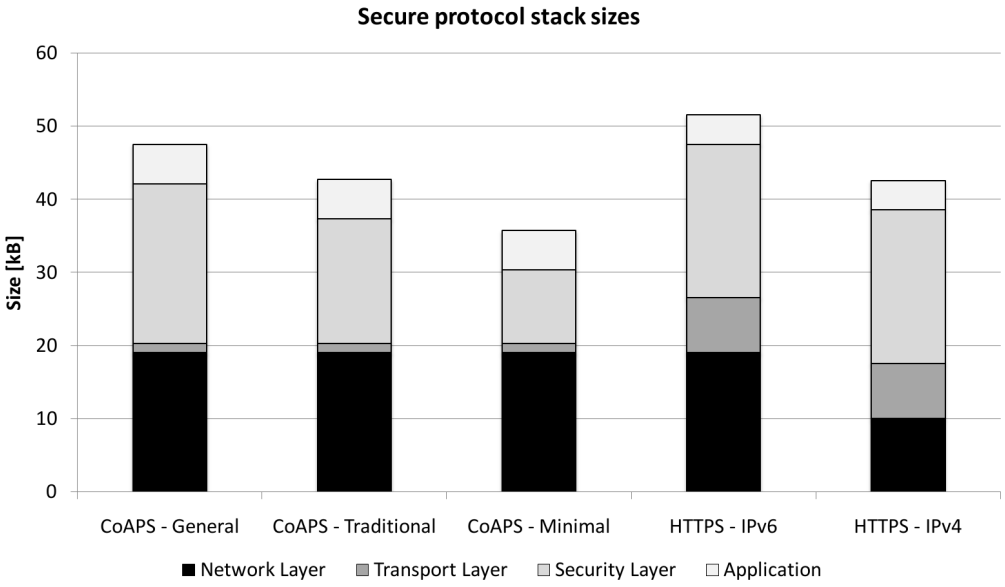

Figure 4.2: *Stack code size summary for TCP and UDP bound application layer protocols.*

4.3.2 Heap usage

In order to get a clear picture about the heap usage of the components, I developed a new plugin for the java based control device application of the iSense operating system (called as iShell). The node sends information via the serial line when the size of the used heap changes and the plugin gathers, summarizes and displays these.

Figure 4.3 shows the results of the heap usage measurement for the TLS (dashed black), DTLS (gray), TCP (black), UDP (dashed gray). For the TLS and DTLS protocols the chart shows the heap requirements for one handshake. The UDP does not use any connection, and in this case the line only shows the standby memory requirement for one listening UDP socket. For the TCP, the figure shows the heap requirements for one new connection. The behavior of the different cases can be compared with regard to duration as well, however because of the frequent memory reports via the serial line, the actual time values cannot be precisely compared to other time measurements in this section.

An important difference between the UDP and the TCP sockets, regarding the heap usage, is that the TCP uses a so-called SendBuffer for the stream based communication. Because

of this buffer, the TCP (and the TLS) reaches higher peaks when the node is sending packets.

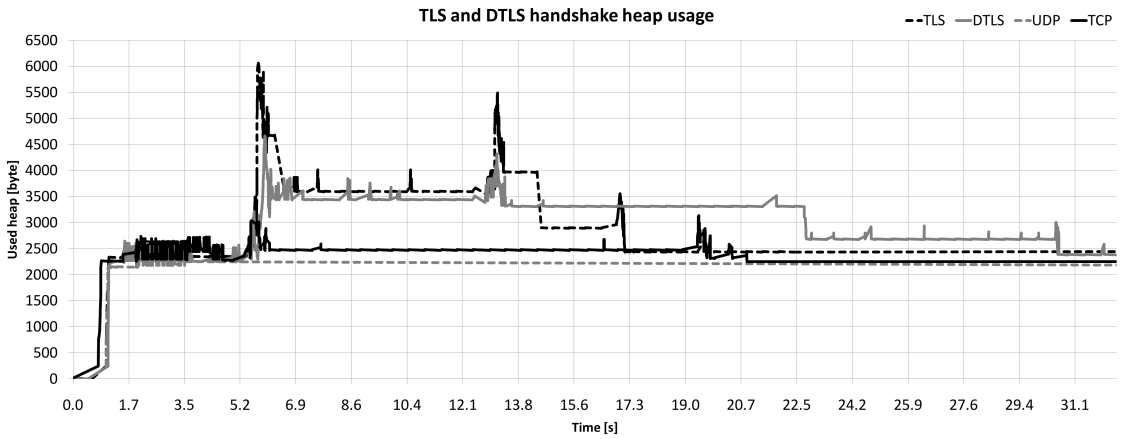

Figure 4.3: *Heap usage for TLS/DTLS (and TCP/UDP for reference).*

All curves start with the startup and initialization of the device and reach the running level in some seconds. The connections are initiated after 5 seconds. The highest peak in the TLS and the DTLS curves is the point when the device sends the certificate to the client. After this peak, the device is working on the ECC calculation for approximately 7 seconds. The second peak shows the end of the ECC calculation. This is the point where the symmetric key derivation takes place and the server sends the final handshake messages. In this figure the behavioral difference between the TLS and the DTLS becomes obvious from here.

After the last handshake messages (t = 13–14), the TLS implementation immediately deletes the objects which were used for the handshake and are not needed any more. The SendBuffer is deleted at the 14th second in the figure, and the connection reaches the normal connected level. Some seconds later (t = 17), the client closes the connection and the server deletes the related objects expect the one which stores the session ID and the master key for possible future session re-negotiation.

The UDP protocol does not provide any reliability for the datagrams. Because of this, during the DTLS handshake, both parties can inform the communication partner about lost messages. Since the server cannot be sure that the last messages have been reached the client, it must be prepared for retransmission for some time after the end of the handshake. This period ends after 22.5 seconds in the figure, and the DTLS deletes the handshake related objects. The DTLS connection reaches the minimum of the required heap at this time. This minimum running level is slightly below the required level for the TLS conncetion. Finally, after the 30th second the connection is terminated.

The TCP connection, which is shown in the figure, is also initiated at the 5th second. The connection is open without any other data exchange until the 20th second, where it is closed.

The figure shows that, except the above detailed message storing difference, the behavior of the TLS and DTLS are similar, and the TLS requires the most memory, out of the 4

compared cases. After the connection has been established, the secured connections require approximately 400 bytes more than the non-secured ones.

4.3.3 Handshake time

The required time to establish the connection was measured for different cases. For these measurements, there is not much difference between the TLS and the DTLS, the figures shows DTLS measurements only. The time was measured for 20 connections in each case and the results are shown in box plots.

The results for the ECDH based key exchange can be seen in Figure 4.4. The time is shown in seconds here, and as it can be seen, the radio does not cause a significant delay, compared to the calculation itself.

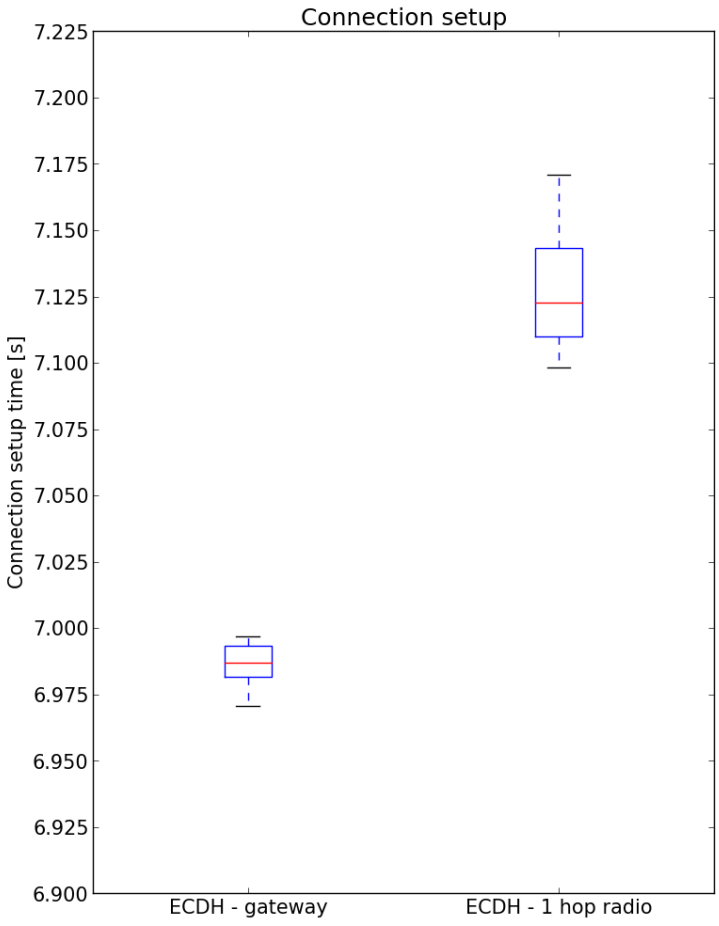

Figure 4.4: *ECDH handshake time*

Figure 4.5 shows the results for the PSK based key exchange and also for the resumed sessions. The time is shown in milliseconds here. The figure demonstrates the strength of the sessions when the ECC based cipher suites are used. However, it can be seen too, that in case of the PSK based cipher suites, the session storing is practically useless. All in all,

the required time for the connections is stable, the deviation is small, especially when the gateway node is used.

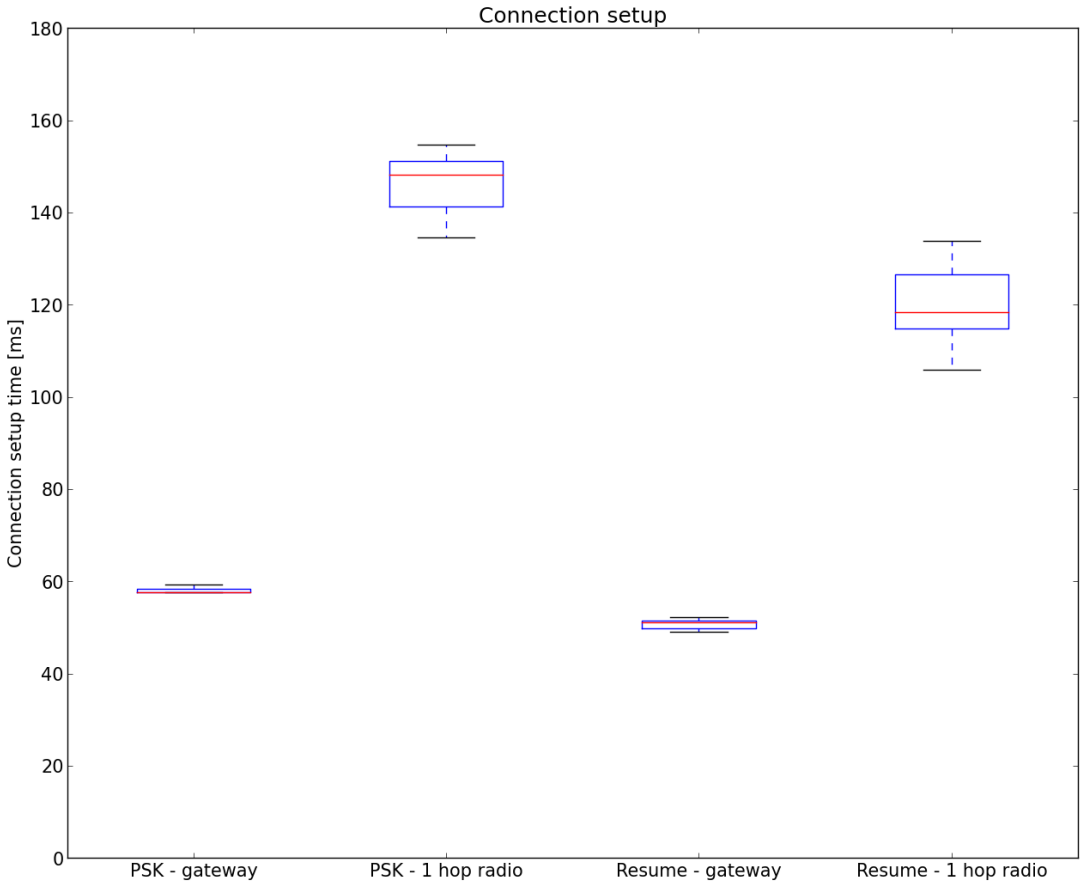

Figure 4.5: *PSK handshake time and resume time*

4.3.4 RTT measurements

This section presents the round trip time measurement results for the implanted protocols. For these measurements, the two scenarios were used from Section 4.2. The results for the TLS and DTLS are shown in different figures, and each figure indicates the performance of the different cipher suites. The plots also show the pure TCP and UDP results for comparison purposes.

The colored areas in the following curves indicate the values between the lower and the upper quartiles, and the lines in the middles show the medians. Before the communication is possible, the handshake must be performed for the secured cases, but the time for that is not included in these plots.

Figure 4.6 shows the RTT results between the computer and the gateway in case of the stream based communication. In this first scenario, the payload length is growing up to 2700 bytes in steps of 100 bytes and the message exchange is repeated 50 times for each length. Since the maximum transfer unit for IPv6 is 1280 bytes, the data cannot be sent

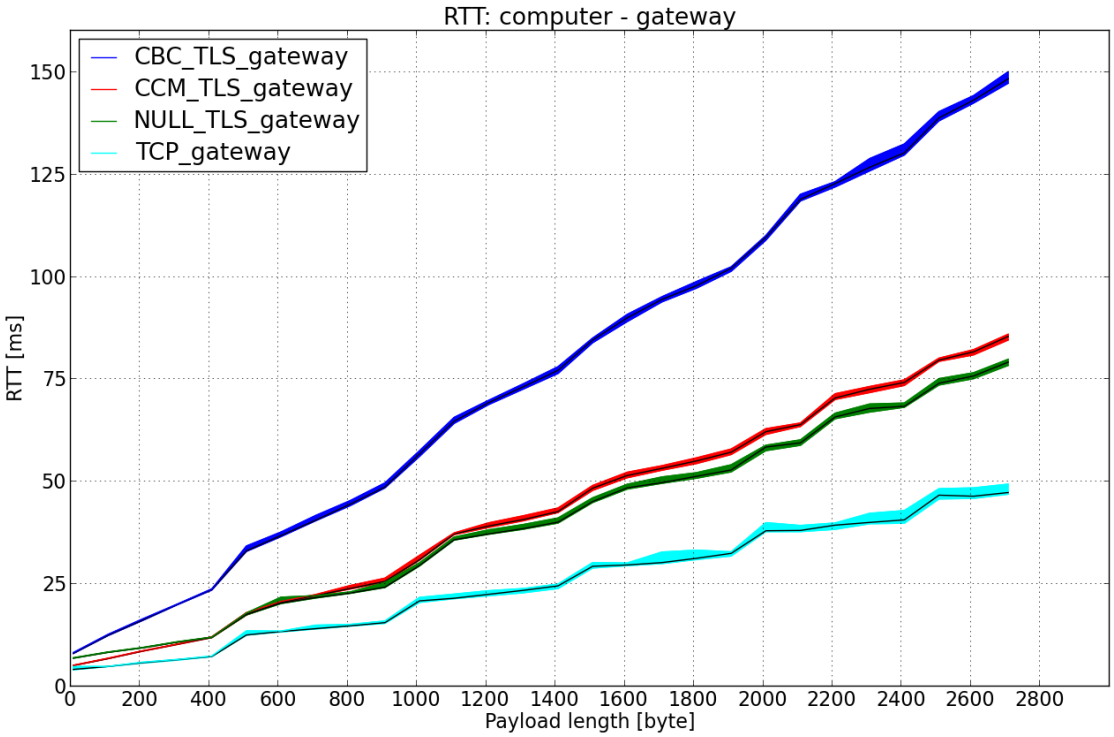

Figure 4.6: *TCP/TLS round-trip-times for the Computer–Gateway-Scenario*

in one packet for the larger payload sizes. The TCP and the UDP handles this differently. The TCP collects the data into the previously mentioned SendBuffer, and creates approximately 500-byte-long IP packets out of it. The transmission window scaling of the TCP is not implemented currently in the iSense. This way the IPv6 packets do not have to be fragmented. All curves show this TCP segmentation, but the position of the jumps are not the same for the pure TCP and the TLS because of the protocol overhead. The figure demonstrates the speed difference of the CBC and the CCM modes, the CCM is approximately two times faster. An interesting result is that the NULL cipher is only slightly faster than the CCM (it is even slower for small packets). The reason is that the MAC algorithm (SHA-1 in this case) works slower than the hardware accelerated CCM mode. The optimization of the existing MAC algorithms was not part of this work, but these should be investigated in the future.

The results for the datagram based communication are shown in Figure 4.7. In contrast with the TCP, the UDP does not care about the size of the packets. It simply pushes the data to the IPv6 layer which results in IPv6 fragmentation after every 1280 bytes. This second figure shows that the IPv6 fragmentation requires longer processing time than the 'fragmentation' in the TCP layer. Except this difference, the behavior of these curves is similar to the TCP based ones. All in all, the datagram based communication is faster than the stream based.

The curves (except for the steps discussed above) are growing in a linear way. The difference between the gradient (between the unsecured and the secured communication) is basically caused by the ciphers and the MAC algorithms. If we elide the steps from the curves, the

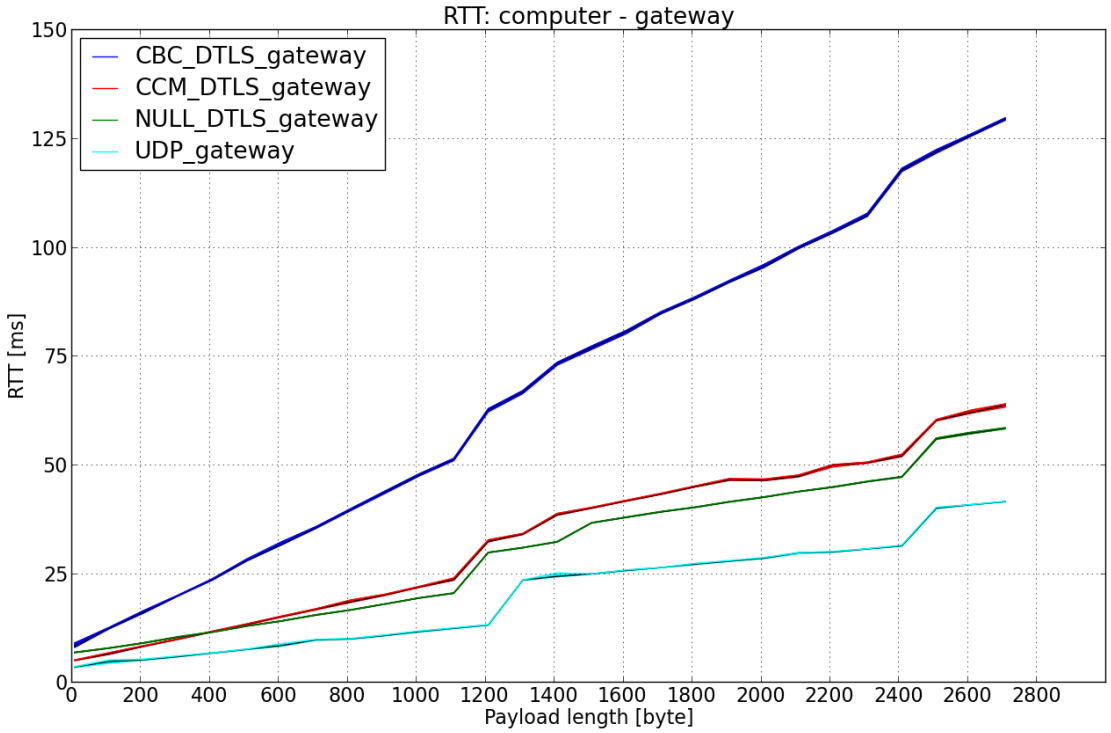

Figure 4.7: *UDP/DTLS round-trip-times for the Computer–Gateway-Scenario*

UDP and the TCP require approximately 0.7-0.8 milliseconds, the DTLS 4 milliseconds and the TLS 4.3 milliseconds per 100 bytes. As a summary I can say that the round trip time remains reasonable for the secure protocol stacks even for larger payload sizes.

Figure 4.8 shows the RTT results for the 1 radio hop scenario. As it was for the previous figures, the RTT was measured after the handshake. In this case the payload length is growing up to 1170 bytes in steps of 10 bytes and the message exchange is repeated 50 times for each length. In this figure, the steps are caused by the fragmentation in the 6LoWPAN layer. The essence of this figure is that the delay of the radio communication dominates the delay of the security parts. While the CCM, NULL and the pure UDP communication cannot be really distinguished, the CBC mode works visibly slower. However, the relative distance of the curves is reducing for the higher payload sizes as well. This relative difference is 44% for 100 bytes, 39% for 500 bytes and 25% for 1000 bytes.

4.4 Evaluation results

This chapter showed the interoperability and performance capabilities of the created implementation. The code size related sections proved the strength of the modular building blocks of the solution, since the required code-memory can be modified on a large scale, depending on the actual use-case. The time related measurements showed that the handshake requires significant time in case of the ECC based cipher suites, but the round trip

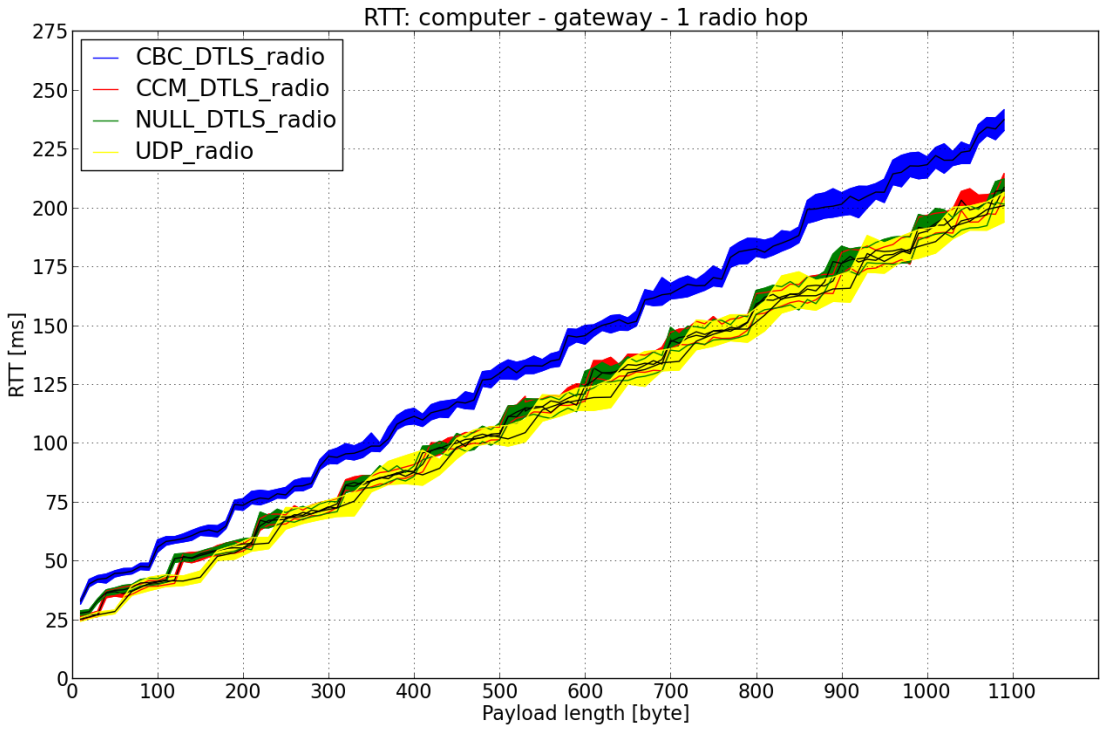

Figure 4.8: *Round-trip-times for the Computer–Gateway–1 radio hop-Scenario*

time is tolerable, and the introduced delay is even negligible when the communication uses the low-power radio.

Chapter 5

Conclusion

The security is getting more and more important for all Internet based applications nowadays. As the wireless sensor networks move towards the Internet of Things, standardized security protocols should be used to protect the data communication.

This paper showed that it is possible to use the TLS and DTLS protocols with various cipher suites in embedded systems. The selected platform for this work was the iSense sensor network operating system, running on the iSense JN5148 microcontroller based devices from the coalesenses GmbH. For the baseline of this work, I used an existing TLS server solution from Michael A. Strebel [30]. As it was detailed before, the code was in bad condition. Actually the final version kept only the basic shape of his work. I restructured, optimized and extended the main building blocks (except the hash functions).

I modified the TLS related codes in order to enable the code-size friendly DTLS implementation. The DTLS code contains all important features, including the cookie exchange mechanism. I added the possibility to enable the session support, and as it was shown during the evaluation, this is important for the ECC based cipher suites. I extended the usable cipher suites from 2 to 10 which enables much more freedom, and it is unprecedented in existing solutions which provide also only 1–2 suites. Many of the provided cipher suites also provide interoperability with commercial solutions.

I adopted the MicroECC library which can perform ECC calculations in approximately 7 seconds, this is 3 times faster than the original ECC solution. The improved version of the AES block cipher works 8 times faster (with hardware acceleration) than the original version in the Wiselib algorithm library.

The code is compilable in a modular fashion which enables the use of the TLS and the DTLS separately and together. This is also true for the different cryptographic algorithms, resulting a scalable solution.

I tested the interoperability capabilities of the implemented protocols with various applications and cryptographic libraries. The secured HTTP server can work together with web browsers, and the interoperability of the secured CoAP was also tested with two different

client side applications. The evaluations showed that the connection setup time is more or less tolerable for the ECC based cipher sites, and fast enough for the pre-shared keys and for the resumed sessions. The achievable communication speed is also reasonable, especially when the low-power radio is also in-use.

As future work, the performance of the hash algorithms should be evaluated and it is expectable that improvable parts will be found. Regarding the ECC calculations, it can be considered to re-write the critical parts in assembly language. From the security aspect, the random number generator must be improved before real-life deployment of the protocols.

All in all, the implemented protocols can be used in various scenarios. It is expected that these will be included into real-life iSense applications when we will reach the age of the Internet of Things.

Bibliography

[1] Y. Zheng, "The Next Generation Network: Issues and Trends," Ph.D. dissertation, Auckland University of Technology, 2008. [Online]. Available: http://aut.researchgateway.ac.nz/bitstream/handle/10292/680/ZhengY.pdf?sequence=4

[2] "Ericsson white paper: More than 50 billion connected devices," 2011. [Online]. Available: http://www.ericsson.com/res/docs/whitepapers/wp-50-billions.pdf

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

[4] S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol," RFC 2401, Nov 1998. [Online]. Available: http:/www.rfc-editor.org/rfc/rfc2401.txt

[5] Z. Shelby and C. Bormann, *6LoWPAN: The Wireless Embedded Internet.* Wiley, 2009.

[6] I. S. Institute, "INTERNET PROTOCOL," RFC 791, Sep 1981. [Online]. Available: http:/www.rfc-editor.org/rfc/rfc791.txt

[7] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460, Dec 1998. [Online]. Available: http:/www.rfc-editor.org/rfc/rfc2460.txt

[8] N. Kushalnagar, G. Montenegro, J. Hui, and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks," RFC 4944, Sep 2007. [Online]. Available: http:/www.rfc-editor.org/rfc/rfc4944.txt

[9] I. S. Institute, "TRANSMISSION CONTROL PROTOCOL," RFC 793, Sep 1981. [Online]. Available: http:/www.rfc-editor.org/rfc/rfc793.txt

[10] C. Wang, K. Sohraby, Y. Hu, B. Li, and W. Tang, "Issues of transport control protocols for wireless sensor networks," in *Communications, Circuits and Systems, 2005. Proceedings. 2005 International Conference on*, vol. 1, 2005, pp. 422–426 Vol. 1.

[11] J. Postel, "User Datagram Protocol," RFC 768, Aug 1980. [Online]. Available: http:/www.rfc-editor.org/rfc/rfc768.txt

[12] C.-Y. Wan, S. B. Eisenman, and A. T. Campbell, "CODA: congestion detection and avoidance in sensor networks," in *Proceedings of the 1st international conference on Embedded networked sensor systems*, ser. SenSys '03. New York, NY, USA: ACM, 2003, pp. 266–279. [Online]. Available: http://doi.acm.org/10.1145/958491.958523

[13] F. Stann and J. Heidemann, "RMST: reliable data transport in sensor networks," in *Sensor Network Protocols and Applications, 2003. Proceedings of the First IEEE. 2003 IEEE International Workshop on*, 2003, pp. 102–112.

[14] T. Dierks and C. Allen, "The TLS Protocol Version 1.0," RFC 2246, Jan 1999. [Online]. Available: http:/www.rfc-editor.org/rfc/rfc2246.txt

[15] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1," RFC 4346, August 2006. [Online]. Available: http:/www.rfc-editor.org/rfc/rfc4346.txt

[16] E. Rescorla and M. N., "Datagram Transport Layer Security," RFC 4347, April 2006. [Online]. Available: http:/www.rfc-editor.org/rfc/rfc4347.txt

[17] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, August 2008. [Online]. Available: http:/www.rfc-editor.org/rfc/rfc5246.txt

[18] E. Rescorla and M. N., "Datagram Transport Layer Security Version 1.2," RFC 6347, Jan 2012. [Online]. Available: http:/www.rfc-editor.org/rfc/rfc6347.txt

[19] "Wikipedia page of the TLS protocoll," accessed: September 18 2013. [Online]. Available: http://en.wikipedia.org/wiki/Transport_Layer_Security

[20] "SSL Pulse," accessed: September 18 2013. [Online]. Available: https://www.trustworthyinternet.org/ssl-pulse/

[21] H. Krawczyk, B. M., and C. R., "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, February 1997. [Online]. Available: http:/www.rfc-editor.org/rfc/rfc2104.txt

[22] "NSA: The Case for Elliptic Curve Cryptography," accessed: September 21 2013. [Online]. Available: http://www.nsa.gov/business/programs/elliptic_curve.shtml

[23] "NIST: Recommended elliptic curves for federal government use," 1999. [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf

[24] N. Bolyard, V. Gupta, C. Hawk, and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)," RFC 4492, May 2006. [Online]. Available: http:/www.rfc-editor.org/rfc/rfc4492.txt

[25] P. Eronen and H. Tschofenig, "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)," RFC 4279, December 2005. [Online]. Available: http:/www.rfc-editor.org/rfc/rfc4279.txt

[26] M. Mittal, "Performance Evaluation of Cryptographic Algorithms," *International Journal of Computer Applications*, vol. 41, no. 7, pp. 1–6, March 2012, published by Foundation of Computer Science, New York, USA.

[27] "Olaf Bergmann – TinyDTLS for Contiki," accessed: September 22 2013. [Online]. Available: http://tinydtls.sourceforge.net/

[28] V. Perelman, "Security in IPv6-enabled Wireless Sensor Networks: An Implementation of TLS/DTLS for the Contiki Operating System," Master's thesis, Jacobs University, 2012.

[29] T. Kothmayr, "A Security Architecture for Wireless Sensor Networks based on DTLS," Master's thesis, Universität Augsburg, 2011.

[30] M. A. Strebel, "Implementierung von TLS für einen Embedded IP Stack," Master's thesis, Universität zu Lübeck, 2013.

[31] M. Varchola, T. Guneysu, and O. Mischke, "Microecc: A lightweight reconfigurable elliptic curve crypto-processor," in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, 2011, pp. 204–210.

[32] J. Daemen and V. Rijmen, *The design of Rijndael*. Springer, 2002.

[33] D. McGrew and D. Bailey, "AES-CCM Cipher Suites for Transport Layer Security (TLS)," RFC 6655, July 2012. [Online]. Available: http:/www.rfc-editor.org/rfc/rfc6655.txt

[34] C. Meyer and J. Schwenk, "Lessons Learned From Previous SSL/TLS Attacks - A Brief Chronology Of Attacks And Weaknesses," *IACR Cryptology ePrint Archive*, vol. 2013, p. 49, 2013.

[35] G. V. Bard, "The Vulnerability of SSL to Chosen Plaintext Attack," Cryptology ePrint Archive, Report 2004/111, 2004, http://eprint.iacr.org/.

[36] ——, "A Challenging But Feasible Blockwise-Adaptive Chosen-Plaintext Attack on SSL," in *SECRYPT 2006, PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON SECURITY AND CRYPTOGRAPHY, SET'UBAL*. INSTICC Press, 2006, pp. 7–10.

[37] T. Duong and J. Rizzo, "Here come the XOR Ninjas," 2011.

[38] J. Kelsey, "Compression and Information Leakage of Plaintext," in *Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002, Revised Papers*, ser. Lecture Notes in Computer Science, vol. 2365. Springer, 2002, pp. 263–276. [Online]. Available: http://www.iacr.org/cryptodb/archive/2002/FSE/3091/3091.pdf

[39] M. Ray and S. Dispensa, "Renegotiating TLS," PhoneFactor, Inc., Tech. Rep., Nov 2009.

[40] B. Canvel, A. P. Hiltgen, S. Vaudenay, and M. Vuagnoux, "Password Interception in a SSL/TLS Channel," in *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, ser. Lecture Notes in Computer Science, vol. 2729. Springer, 2003, pp. 583–599. [Online]. Available: http://www.iacr.org/cryptodb/archive/2003/CRYPTO/1069/1069.pdf

[41] N. J. AlFardan and K. G. Paterson, "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols," in *IEEE Symposium on Security and Privacy*, 2013, pp. 526–540.

[42] S. Jucker, "Securing the Constrained Application Protocol," Master's thesis, ETH Zurich, 2012.