Zurich University of Applied Sciences
School of Engineering
Institute of Embedded Systems (InES)

Bachelor Thesis

Secure Wireless Communication for Hearing Aids

Authors:
Leo Rudin
Nathalie Achtnich

Supervisors:
Dr. Simon Künzli
Dr. Michael von Tessin

Advisor:
David Lorenz

Submitted on
August 16, 2023

Study program:
Bachelor Degree in Computer Science

Abstract

Today, hearing aids communicate with a wide variety of devices via wireless connections. Like all networks, these connections are at risk of being compromised by an attacker when left unprotected. Until now, the partner company of this thesis, a large hearing instrument manufacturer, has used a proprietary security protocol to secure the connection. However, this company's new generation of hearing aids will have more powerful hardware, which raises the question of using standardized security protocols anew. This paper aims to contribute to the clarification of this question. Since the bottleneck of communication overhead during data exchange will remain, this work focuses on the optimization of standardized security protocols with respect to this overhead. In a first phase, the theoretical foundations necessary for understanding the selected security protocols and overhead reduction are established. The security protocols are then weighed against each other with regard to a prototype, and TLS and TLS/QUIC are identified as the most promising. The implementing libraries chosen are wolfSSL for TLS and TLS/QUIC and lwIP for the underlying TCP/IP stack. Subsequently, the underlying transport framework for the TLS variant is implemented in a prototype and wolfSSL is tested in a client-server setup. The existing prototype, which also includes an interceptor task, can be used as a basis for further investigations. The use of the wolfSSL library has proven to be rather cumbersome, which is why its use in a further examination is not recommended.

Zusammenfassung

Heutzutage kommunizieren Hörgeräte mit den verschiedensten Geräten über drahtlose Verbindungen. Ungeschützt stehen diese Verbindungen, wie alle Netzwerke, in der Gefahr, von einem Angreifer kompromittiert zu werden. Bisher verwendete das Partnerunternehmen dieser Arbeit, ein grosser Hörgerät-Produzent, ein proprietäres Sicherheitsprotokoll, um die Verbindung abzusichern. Die neue Hörgerät-Generation dieser Firma wird aber über eine leistungsfähigere Hardware verfügen, was die Frage nach dem Einsatz von standardisierten Sicherheitsprotokollen neu aufwirft. Die vorliegende Arbeit will einen Beitrag dazu leisten, diese Frage zu klären. Da das Nadelöhr des Kommunikations-Overheads während des Datenaustauschs bestehen bleiben wird, fokussiert diese Arbeit auf die Optimierung standardisierter Sicherheitsprotokolle betreffs dieses Overheads. Dafür werden in einer ersten Phase die theoretischen Grundlagen etabliert, die für das Verständnis der ausgewählten Sicherheitsprotokolle und der Overhead-Reduktion nötig sind. Für die Erstellung eines Prototyps werden dann die Sicherheitsprotokolle gegeneinander abgewogen und TLS und TLS/QUIC als die vielversprechendsten identifiziert. Als implementierende Bibliotheken werden wolfSSL für TLS und TLS/QUIC und lwIP für den darunterliegenden TCP/IP-Stack gewählt. Anschliessend wird das unterliegende Transport-Gerüst für die TLS-Variante in einem Prototypen umgesetzt und wolfSSL in einem Client-Server-Setup getestet. Der bestehende Prototyp, der auch eine Vorrichtung zum Abfangen von Paketen umfasst, kann als Grundlage für weitergehende Untersuchungen genutzt werden. Die Verwendung der wolfSSL-Bibliothek hat sich als eher umständlich erwiesen, weshalb von einem weiteren Einsatz abgeraten wird.

Preface

Performing this thesis has allowed us to gain deep insights into the field of embedded systems and their security. Not only did we learn a lot about security protocols and their implementation by extensive research and hands-on experience in the matter, but we also had the opportunity to work with seasoned partners from business whose expertise was invaluable for our professional growth. We would therefore like to thank our point of contact from the hearing instrument manufacturer that we worked with, Dr. Michael von Tessin, for his relentless support and sharing of expert knowledge throughout the semester. We further want to thank our supervisor Dr. Simon Künzli from ZHAW, who made the partnership with the manufacturer possible in the first place and provided us with many handy hints regarding the technical as well as the organisational aspects of our work. A special thanks goes to our advisor David Lorenz, who spent hours on reviewing our code and thus helped us out of trouble multiple times. We hope that our joint forces may help to advance the secure use of hearing aids worldwide.

Contents

Chapter 1

Introduction

1.1 Background

The industrial partner of this thesis is an internationally active hearing instrument manufacturer specializing in the development and production of hearing aids. The possession of several brands makes the company one of the largest providers in the hearing aid sector worldwide.

Nowadays, modern hearing aids are getting more and more connected. Besides having the right and left component of the hearing aids connected in order to exchange information, hearing aids are also able to communicate with other devices like PCs and smartphones. This for instance allows an audiologist to calibrate and configure the hearing aids to their patients' needs or updating the device firmware. Users can also use their own smartphone to change various settings on their hearing aid.

This connectivity makes the hearing aid a viable target for cyber attacks, and as such, they are subject to various threats to all of the three elements of the Confidentiality - Integrity - Availability triad:

– Confidentiality: A malicious actor might eavesdrop on integrated microphone to gather valuable information or extracting health related data of the user.

– Integrity: A malicious actor might manipulate data that is exchanged between the hearing aid and other devices to play loud sounds that could further damage the user's ears.

– Availability: A malicious actor might render the hearing aid non-functional and therefore lower or completely hinder the user's hearing ability.

As a consequence, our hearing instrument manufacturer decided to require authentication over an end-to-end secured channel for very sensitive connection types. This for example includes connections between the hearing aid and the audiologist's device.

After evaluating various standardized security protocols regarding their message overhead and computational overhead, it was found that the hardware constraints that came with the current generation of hearing aids could not satisfy the requirements of those protocols. On these grounds, our industrial partner was forced to implement their own proprietary security protocol.

While they did a formal verification of their proprietary security protocol in collaboration with ETH Zürich [1], relying on a non-standard protocol always imposes a higher risk. In contrast to standardized protocols, the implementation and protocol details of the proprietary protocol are not publicly available and cannot be verified by a wider audience, which increases the potential risk of implementation bugs and protocol issues.

The next generation of hearing aids introduces significant enhancements in terms of CPU processing power and memory capacity. This also reopens the discussion of using standardized protocols as their requirements could now be satisfied by the next generation's hardware.

1.2 Goals

The goal of this bachelor thesis is to evaluate the potential of various standard security protocols on the basis of the new hardware to be used in the next generation of hearing aids of our industrial partner. Security protocols that are particularly promising include, but are not limited to, TLS, DTLS, and IPsec. Identification of other possible protocols will be part of the research scope. The focus of attention is to be placed on the communication overhead resulting from the data traffic and on how to minimize it while retaining the security features, namely confidentiality, integrity, and availability. If a trade-off has to be made, its implications on the security level must be explored as well. Apart from this, the underlying protocol stack should be taken into account concerning additional overhead caused by communication protocol headers. Throughout the protocol stack, standardized solutions are to be preferred.

Following the evaluation, a simple prototype should be implemented that enables the verification of one or multiple security protocols that were identified as auspicious on hardware that is similar to the actual hardware being used in the next generation's hearing aids. Based on the assessment of this experimental setup, especially regarding the communication overhead, we seek to work out the suitability of a standardized security protocol for the hearing instrument manufacturer's future hearing aids.

1.3 Outline

This thesis is structured as follows: After having stated the problem and the goal of this thesis in the introduction, a theoretical part follows in Chapter 2 where we provide the bedrock for the subsequent sections: The target system, i.e. the hearing aid of the next generation, and the prototype hardware and software simulating this hearing aid for the purposes of this thesis are described, followed by an overview of the five security protocols evaluated in this work. In the same chapter, we take a closer look at the underlying internet and transport layers and some possibilities to optimize their functionality for embedded applications. Chapter 2 is closed with a survey of the cryptographic cipher suites supplied by the target system. In Chapter 3, we undertake an evaluation of the security protocols presented in Chapter 2 with regard to their communication overhead and come to a decision which protocol will be implemented in the prototype. Besides the protocols, implementation libraries as well as cipher suites are also balanced against each other such that a conclusion can be drawn on which to use in the prototype implementation. In Chapter 4, we describe the implementation of the prototype, how we designed its architecture and made use of tools and libraries to build a small network with measurement capability. After that, the results of the implementation phase are depicted in Chapter 5. Concluding this work, we discuss the findings of the previous chapter and give an outlook on further research possibilities in Chapter 6.

Chapter 2

Foundations

In this chapter, we lay the foundations needed for a thorough comprehension of the matter, namely the specifications and constraints of hearing aids of our industrial partner regarding their wireless connectivity plus the innovations introduced with the next generation, the security protocols that could be optimized to work on embedded devices as well as some compression technologies enabling reduction of the overhead generated on the transport and the internet layer. In a last part, we present an overview of the cipher suites that are offered by the target system.

2.1 Target System and Constraints

For the hearing aids of our industrial partner, like for many other technological assistive tools, wireless connectivity is indispensable. It is used for establishing connections to a variety of endpoints: remote controls, fitting software of the audiologist, smartphones for telephony and media, the cloud and more. Furthermore, state synchronization and audio streaming between the left and the right hearing aid also take place over a wireless connection.

For these purposes, the hearing aids employ a frequency band of 2.4 GHz which is the standard that Bluetooth also uses. As a communication protocol, Bluetooth Low Energy (BLE) is applied, which is optimized for low power target systems. For some specific targets, other communication protocols are used but are not in the scope of this thesis.

The next generation of our manufacturer's hearing aids will come with a bundle of new and/or enhanced features. While the general system structure remains the same, there will be major changes in the electronics and in the embedded software. The chip which handles wireless connections will be replaced by its newest edition. The device hardware will thus allow for much more extensive computational operations whereas the frequency band for the BLE connections will stay the same. The performance bottleneck therefore consists in the wireless connection and is even exacerbated when multiple connection tasks run in parallel. Hence, optimization of the power and memory consumption of the security protocols is subsidiary while the principal focus lies on the reduction of the communication overhead.

The chips used in the subsequent hearing aids also impose some restrictions on the security. The cryptographic hardware blocks are limited to ECC/RSA encryption for asymmetric cryptography, AES-128/256 for symmetric cryptography, SHA-2/3 for hashing and TRNG for generating random numbers. The choice of the cipher suite may impact the traffic overhead as well, which will be discussed in Chapter 2.5.

2.2 Prototype Hardware and Software

2.2.1 Evaluation Board

The prototype hardware that is provided for implementation and evaluation purposes for this thesis is the MIMXRT685-EVK evaluation board by NXP, a microcontroller of the RT600 family [2]. It features an Arm Cortex-M33 processor which is a next generation core based on the ARMv8-M architecture with low power consumption and which can operate at frequencies of up to 300 MHz. Two hardware co-processors enable hardware acceleration for cryptography. The RT600 provides up to 4.5 MB of on-chip SRAM accessible by both CPUs. Two general purpose DMA (direct memory access) engines can be configured to be used by a specific controller, e.g. the M33 CPU. Several interfaces are available, including a FlexSPI flash interface, a high-speed SPI interface, a I3C bus interface and eight configurable universal serial interface modules. There is also a Cadence Xtensa HiFi 4 Audio DSP engine which provides support for efficient audio and voice encoding/decoding execution but which will not be used in the course of this research study.

2.2.2 Software Application

Together with the prototype hardware, a minimum viable software product was delivered that builds and runs on the RT685-EVK board. Its directory structure is shown below, with files not directly relevant for our case being left out. The individual components are described afterwards.

```
hd-sec-protocols
├── toolchain
│   └── armgcc.cmake
├── mvp-lib-embos
│   ├── embOS_CortexM
│   └── embOS_Sim
├── mvp-servicehost
│   ├── BoardSupport
│   │   └── RT685EVK
│   │       ├── MCUXpresso
│   │       ├── SEGGER
│   │       ├── SetupBoard
│   │       ├── SetupEmbOS
│   │       └── Startup
│   └── Source
│       └── ServiceHost.cpp
├── mvp-app-hello
│   └── Source
│       └── Application.cpp
└── CMakeLists.txt
```

toolchain. This folder contains information about how to compile and link the application as well as how to download the application via the GDB J-Link server to the evaluation board. The `armgcc.cmake` file sets up the C/C++ and ASM compiler depending on the compiling system, Linux or Windows. `rt685evk_download.gdb` contains instructions for GDB about the GDB server location to download the executable file to the board.

mvp-lib-embos. The `mvp-lib-embos` directory contains the embOS real-time operating system (RTOS) from SEGGER [3] that comprises a configuration for Cortex-M CPUs and for Windows, of which the former is relevant for our case. EmbOS, a priority-controlled multitasking system, is optimized for high speed and low memory consumption. As an RTOS, it leverages preemptive scheduling for CPU time distribution among tasks, i.e. the task with the highest priority runs as long as it is not suspended either by calling a blocking function or by another task with higher priority. On embOS, all tasks are threads, which means that all tasks can access the same memory locations. In Section 4.1, we describe how we take advantage of the embOS multitasking system to set up our client/server prototype pair.

mvp-servicehost. The `ServiceHost` contains the `main` function where the application is started. Here, the operating system is initialized, the tasks are created and the kernel is started. It also contains specific support functions depending on the target platform. For example, the `printf` function allows writing output to the serial connection between the board and the developer's host machine.

mvp-app-hello. Here resides the actual user program. For now, there is a single "Hello World" function using the ServiceHost for printing the statement out to the console.

CMake. CMake, the build tool used for this project, operates with compiler independent configuration files that enable the generation of environment-specific Makefiles. In our case, CMake creates build files to run the Ninja build system. These independent configuration files, called `CMakeLists.txt`, are organized in a hierarchical structure depending on the directory structure of the project. Usually, all libraries and executables have their own `CMakeLists.txt` file containing instructions regarding the build of this entity, as well as the root folder containing a `CMakeLists.txt` that directs the configuration of the overall project and the inclusion of subdirectories.

2.3 Security Protocols

Currently, numerous security protocols are available, although not all of them are well-suited for use in embedded devices, which have limited RAM and memory space, along with other constraints. In the following subsections, we will delve into five security protocols that could be customized to function in embedded devices: TLS, DTLS, IPsec, COSE, and WPA. Providing this context is essential to gain a better understanding of the specific workings of these protocols as we compare their suitability for our task later on.

2.3.1 TLS

The Transport Layer Security protocol (TLS) [4], formerly known as Secure Socket Layer protocol (SSL), is one of the most widely used protocols to protect the authenticity, integrity, and confidentiality of data sent over the Internet. The standard was proposed by the Internet Engineering Task Force (IETF) and is mainly used for securing HTTP connections but has a variety of other applications. Different versions exist, most famously TLS 1.2 and TLS 1.3. While TLS 1.3 is considered more secure due to its enhanced cryptographic suites and mechanisms, it has not suffered considerably from performance decline [5][6]. Its communication overhead is also remarkably smaller compared to TLS 1.2 because the initialization vector does not have to be transmitted with every message anymore. This makes TLS 1.3 the version of choice, and all information below refers to TLS 1.3.

As its name implies, the protocol is settled on the transport layer, although it does not fit perfectly in the OSI model: It actually runs on top of a transport protocol, however, applications use the TLS protocol as if it were a transport layer [7]. TLS requires a reliable underlying transport protocol like TCP. The cryptographic means applied in the data exchange between two endpoints are usually agreed upon in the course of the TLS procedure. The TLS procedure consists of two phases, called the *TLS handshake* and the *TLS record layer*, which are described in the sections below.

TLS Handshake. The TLS handshake enables authentication of the communication endpoints as well as the establishment of a symmetric key that is used later on for encrypting the application data. The steps of the handshake process are shown in Figure 2.1. In the following, we will look at this process in more detail and see how the connection is thereby secured.

```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

FIGURE 2.1: TLS handshake and TLS record protocol. The steps with an asterisk are optional. [4]

The steps of the TLS handshake protocol can be grouped in three phases: Key exchange, server parameters, and authentication.

1. **Key exchange.** This phase comprises the `ClientHello` and the `ServerHello` messages. In the `ClientHello`, the client initiates the connection to the server

and, among other information, lists the cipher suites that it supports, preparing the establishment of a shared secret. The server responds with a `ServerHello` containing the selected cipher suite and other potential parameters relevant for securing the connection, like keying material. In the case of session resumption, the `ClientHello` and the `ServerHello` messages include information about the pre-shared key instead of agreeing on a new cipher suite.

2. **Server parameters.** Following the `ServerHello`, the server sends another message called `EncryptedExtensions`, containing responses to the `ClientHello` extensions that are not needed to establish a shared secret. This is the first encrypted message – from here on, every message is encrypted with the key derived from the shared secret. If client authentication is desired, the server also sends a `CertificateRequest` to the client.

3. **Authentication.** Finally, the authentication phase begins by the server sending its `Certificate`, followed by the `CertificateVerify` which consists of a signature over the hash value of the entire conversation so far. After this, the server concludes with the `Finished` message which transmits a MAC over the hash value of the whole conversation (including the `CertificateVerify`). Then the client sends its `Certificate` and `CertificateVerify` if it was requested by the server before. When the server receives and accepts the client's `Finished` message, the handshake phase has successfully been completed.

TLS Record Layer. Now that the integrity, the authenticity and the confidentiality of the the channel between the two endpoints has been established, the TLS record layer is used for exchanging encrypted data. Since this thesis aims at reducing the overhead particularly of the record layer, we will elaborate on its packet format in more detail.

In general, the record layer protocol fragments data blocks into `TLSPlaintext` records of 2^{14} bytes or less and compresses it without loss. It adds a MAC for integrity, encrypts the data for confidentiality, and prepends the record header. The resulting packet format is visualized in Figure 2.2.

The header of the packet is made up of three fields: the content type, the legacy protocol version, and the fragment length. The content type signals the higher-level protocol used to process the transmitted data. The legacy protocol version has its only purpose in achieving backward compatibility with older TLS versions and must always be set to a specific value signifying TLS 1.2 for TLS 1.3. Its content must be ignored under all circumstances, however. The length field describes the length of the following data fragment. In total, the header fields sum up to 5 bytes.

The tail consists of padding bytes in case of a block cipher and a mandatory authentication tag with a default size of 16 bytes since TLS 1.3 only supports AEAD (Authenticated Encryption with Associated Data) ciphers. It is possible, though not common, to reduce or truncate the authentication tag to 4 bytes which leads to a trade-off concerning integrity. Therefore, the tail contributes a minimum of 4 bytes (when using a stream cipher and a maximally reduced authentication tag size) and a maximum of 32 bytes (in case of full padding and a maximum authentication tag size).

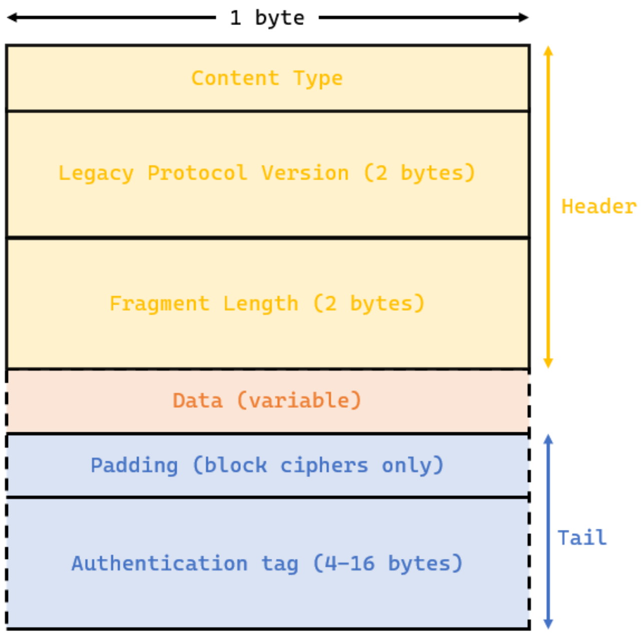

1 byte
Content Type
Legacy Protocol Version (2 bytes)
Fragment Length (2 bytes)
Header
Data (variable)
Padding (block ciphers only)
Authentication tag (4-16 bytes)
Tail

FIGURE 2.2: TLS record layer packet format.

2.3.2 DTLS

The Datagram Transport Layer Security (DTLS) protocol is based on the TLS protocol, which was presented in Chapter 2.3.1. But in contrast to TLS, DTLS is able to run on transport protocols which are not stream oriented like TCP. DTLS version 1.3 is standardized in RFC 9147 [8], which specifies the usage of DTLS over the UDP transport protocol. However, DTLS is not limited to operate on UDP exclusively, as there are RFCs which specifiy the usage on other non-stream oriented transport protocols like SCTP. DTLS version 1.2 is based upon TLS 1.2 and version 1.3 on TLS 1.3.

TLS expects a reliable, stream-oriented underlying transport protocol which means that DTLS adapts the TLS protocol to circumvent the following issues that come with operating over protocols other than TCP:

1. TLS expects sequence numbering as well as the prevention of packet-loss which is implicitly provided by TCP. DTLS therefore adds explicit sequence numbers to the DTLS record layer and provides a re-transmission timer to handle packet loss. Either communicating party has a timer which may expire upon sending a message to the other party. If expired, the message is re-transmitted and the other party knows that their packet was lost during transmission and performs a re-transmission as well.

2. TLS does not allow re-ordering of packets. DTLS solves this by adding the before mentioned sequence numbers where the receiver can immediately check if the received packet contains the expected sequence number.

3. As TLS handshake messages tend to get rather large, DTLS provides its own fragmentation and reassembly functionality within handshake messages.

4. Datagram transport protocols are potentially vulnerable to Denial-of-Service

(DoS) attacks. DTLS counters this by borrowing the stateless cookie technique which is also used in the IKE protocol of IPSec (see Chapter 2.3.3). In short, the server responds upon a `ClientHello` with a cookie, which the client then uses to re-transmit the `ClientHello message` with the cookie included, which the server then verifies.

DTLS also provides detection for record replay. However this feature is not mandatory since duplicate messages may also appear due to routing errors and not for malicious purposes.

DTLS Handshake. DTLS leverages the TLS handshake flows and messages, as described in Chapter 2.3.1, with some adaptations. Handshake message headers contain additional fields to handle message loss, reordering, and fragmentation. The aforementioned retransmission timers were introduced to handle packet loss, and a new ACK message type was introduced, which allows for reliable delivery of handshake messages.

DTLS Record Layer. Figure 2.3 illustrates the `DTLSCiphertext` structure of a DTLS record layer message that follows after the DTLS handshake has been performed. The DTLS 1.3 record layer structure generally provides more flexibility compared to the DTLS 1.2 record layer which is the reason only version 1.3 is presented in this document.

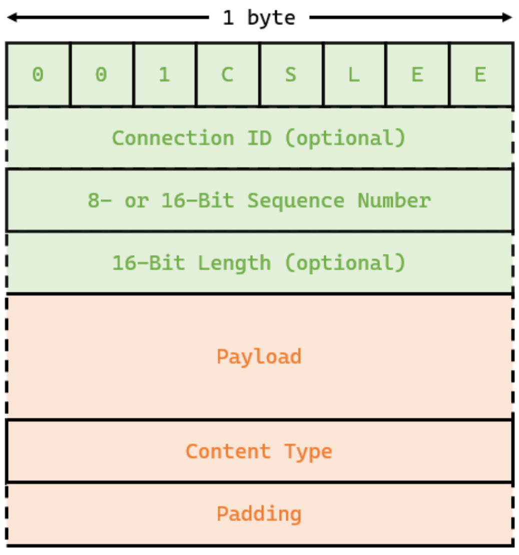


FIGURE 2.3: Structure of a DTLS version 1.3 record layer message.

The structure first starts with three fixed bits that distinguish the record layer from the previous version 1.2. The five following bits control different parts of the message:

- **C:** Indicates the presence of the optional Connection ID. The connection ID is an extension to the DTLS protocol which allows the receiver to select the appropriate security association to correctly process the packet. If not used, the security association is determined using the IP address and port of the peer which could sometimes change due to network address translation (NAT) procedures.

– **S:** Indicates the length of the sequence number, either 8 bit (0) or 16 bit (1).
– **L:** Indicates the presence of the optional length field. If the length field is omitted, the data is consumed until the end of the underlying transport layer datagram.
– **EE:** Contain the least significant two bits of the connection epoch value.

Afterwards follows the variable length payload as well as the content type which has a length of 1 byte. The `ContentType` field is used to distinguish the type of data that is transmitted. For application data, the value 23 is used. In the best case, the header would generate an overhead of 3 bytes excluding potential message authentication codes. This would include the first header byte containing the bit flags, the 8 bit sequence number and the 8 bit content type.

It should also be noted that, according to RFC 6347 section 1.3 [9] which specifies DTLS 1.2, the use of stream cipher is banned. Therefore there will be additional overhead generated by potential initialization vectors and padding.

2.3.3 IPsec

IPsec [10] is a suite of security network protocols consisting of following protocols: Authentication Header (AH) protocol, Encapsulating Security Payload (ESP) protocol, and the Internet Key Exchange (IKE) protocol. While the ESP protocol provides confidentiality, integrity and authentication, the AH protocol only provides integrity and authentication, which makes the AH protocol not suitable for our use case and is therefore not discussed further.

While most of the other protocols presented in this document operate on higher layers within the OSI model, IPsec protocols operate on the internet layer and are directly protecting IP packets. This provides more flexibility regarding the usage of higher layer protocols. There are neither restrictions on what can be transported as payload nor a tight coupling with other protocols.

An implementation of IPSec can either be provided directly by the operating system's IP stack or as "bump in the stack", where IPSec is added a as an additional component within the OSI layer, operating independently and processing IP packets from the internet layer [11].

IKE Protocol. The IKE protocol, which is specified in RFC 7296 [12], is used to establish and maintain Security Associations (SAs) between two connecting hosts and perform mutual authentication. A Security Association is an agreement between two hosts on how to apply IPSecs security mechanisms for IP packets that travel in one direction. It for instance contains the algorithms that should be used to encrypt the payload as well as the necessary keys to do so. Security Associations are saved and maintained in a Security Association Database (SAD).

There exist two versions of the IKE protocol, version 1 and 2. Version 2 was developed by the IETF in 2005 and version 1 was marked obsolete. Reasons for the new version were the low performance of IKEv1, missing protection against DoS attacks due to the exchange of stateful information and the overall complexity, as the first version was spread over several RFCs. IKEv2 is now only specified in one RFC and is able to perform a exchange within just 4 messages.

When a host A wants to send a packet to host B, it first checks the Security Policy Database (SPD) if sending packets to host B requires IPSec protection. If yes, the database would contain a Security Policy (SP) which also includes the SA that should be used. If no, the IKE protocol will be used to establish the SA for sending packets from host A to host B.

The IKEv2 protocol consists of two phases: Phase one generally establishes common parameters between host A and B and phase two establishes the specific SA to exchange messages using either the ESP or AH protocol. The initiator (host A) therefore starts with an `IKE_SA_INIT` message that contains proposals for the encryption algorithms to be used as well as a Diffie-Hellman parameter and a cryptographic nonce. The responder (host B) then chooses one of the proposals and responds with their decision as well as its Diffie-Hellman parameter and cryptographic nonce. Now both parties calculate their respective keys. Host A then responds with with a `IKE_AUTH` message which is already protected by the previously calculated keys and authenticates itself to host B. Additionally, A also already provides proposals on how A wants to communicate with B (ESP or AH) and the algorithms to use. B verifies this, authenticates itself as well, and responds with the chosen proposal. During phase 2, the mutual security parameter index (SPI) that together with the IP address uniquely identifies the used SA on both sides, is transmitted as well [13] [14].

ESP Protocol. The ESP protocol [15] can be used to apply integrity, confidentiality and authentication to all IP packets that are transferred over an IPsec protected channel. ESP requires a symmetric cipher for encryption and all packets that are transferred should contain all relevant information to correctly decrypt the transferred data. To achieve this, ESP adds the ESP header and trailer to each outgoing IP packet as illustrated in Figure 2.4. The receiver of an ESP packet then uses this information to correctly decrypt/check a received packet and passes it up to higher layers.

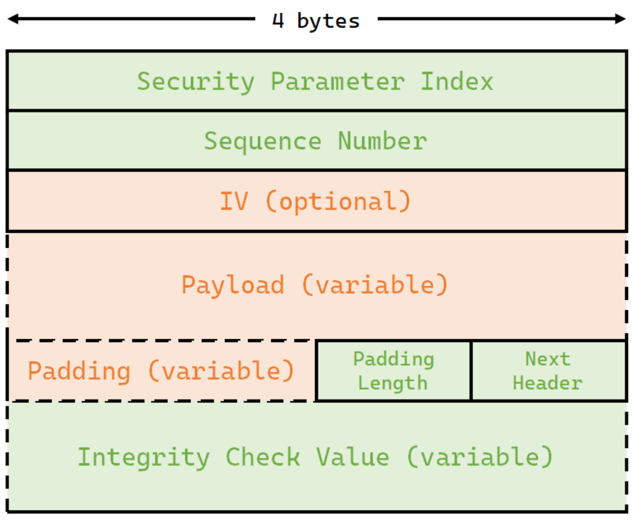


FIGURE 2.4: ESP header and trailer format.

The receiver will use the value that is transmitted within the Security Parameter Index (SPI) field to find the Security Assocation (SA) to determine the algorithms and keys that are needed to process the incoming ESP packet. Since IPsec operates on the IP layer where there is no guarantee that packets are arriving in the same order intended by the sender, the header also contains a sequence number which is used to avoid replay attacks. The sequence number is maintained for an SA and is incremented for each packet until the value reaches its maximum size of 32 bits. Afterwards, a new SA has to be established that starts the counter again at zero. Next follows the actual payload with an optional initialization vector in front in case a block cipher is used, which would also mean that the optional padding field is used to fill up the block. The Next Header field is used to mark which header immediately follows the ESP header. Finally, the integrity check value (ICV) guarantees the authenticity of the received packet and is added after the payload.

Transport Modes. IPsec can operate in two distinct modes: Transport and tunnel mode. In transport mode, as illustrated in Figure 2.5, the ESP header is inserted between the IP header and the subsequent header. The overhead in this mode would be 9 bytes without an integrity check value which always varies depending on the chosen algorithm. Of course, the initialization vector and potential padding also need to be considered but these two values could be omitted entirely by using stream ciphers instead of block ciphers.

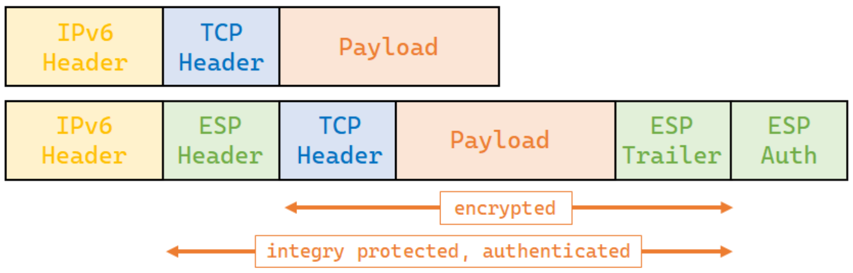


FIGURE 2.5: Processing of a IP packet by IPsec in transport mode.

In tunnel mode, as illustrated in Figure 2.6, a completely new IP header is created and positioned in front of ESP header while the ESP header is positioned in front of the original IP header. This also means that the complete original IP packet including the original header is now subject to IPSec's protection mechanisms. For this reason, this mode is extensively used by VPN services as this also hides the sender and receiver of the IP packet. This, of course, increases the overhead by the length of the respective IP protocol version, 40 bytes in case of IPv6 or 20 bytes in case of IPv4, which results in a total overhead of at least 49 and 29 bytes, respectively.

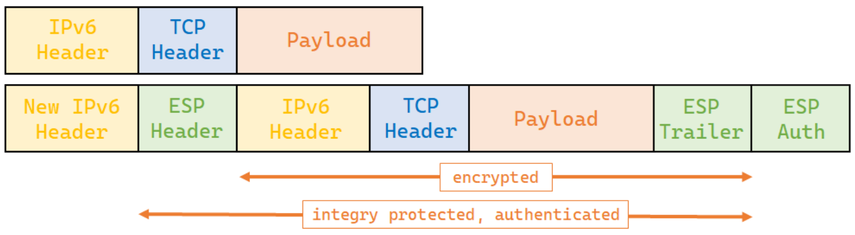


FIGURE 2.6: Processing of a IP packet by IPsec in tunnel mode.

2.3.4 QUIC

QUIC is not solely a security protocol but, more specifically, a transport protocol tightly coupled with the TLS protocol (see Chapter 2.3.1) and specified in RFC 9000 [16]. It usually comes with HTTP/3 on the application layer, which is specified in RFC 9114. The primary motivation to create a completely new transport protocol were the problems with the TCP protocol that is currently used as a transport protocol to transmit HTTP/2 traffic [17]. HTTP/2 is the next version of the HTTP protocol, and is specified in RFC 7540. One of the main enhancements of HTTP/2 is the introduction of multiplexed connections that make use of the same TCP connection to perform multiple requests. While this improved the overall performance and experience as there is no need anymore for each request to finish until the next request can go out, this introduced the same so-called "head of line blocking" problem but on the transport instead of the application layer.

TCP Head of Line Block. TCP is a reliable transport protocol. This means that packets which are transmitted between two connecting endpoints are guaranteed to be delivered in the correct order. If one of the packets is lost, the whole transmission process is being halted and every packet that directly follows has to wait. In cases of networks with high packet loss, HTTP/1 would even outperform HTTP/2 as more physical TCP connections exist compared to the latter. QUIC addresses this problem by introducing virtual streams that independently operate on one physical reliable connection between to parties.

QUIC Protocol Stack. Illustrated in Figure 2.7 is a comparison between the HTTP/2 and QUIC stack. The blue parts indicate elements that belong to the QUIC stack.

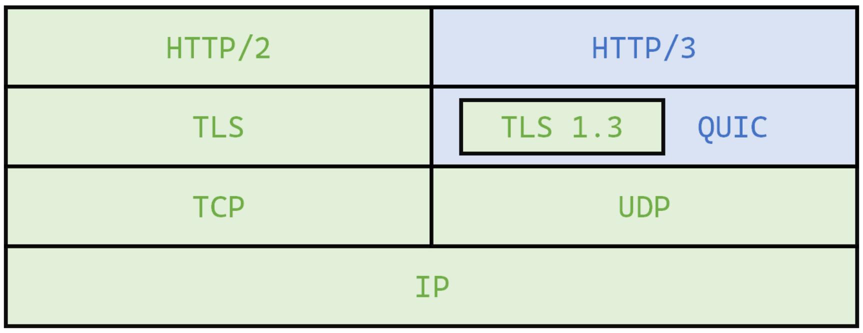


FIGURE 2.7: QUIC compared to HTTP/2 protocol stack.

The TLS protocol is deeply integrated into the QUIC protocol, which means that QUIC messages are always transmitted in a secure way. Whenever a QUIC handshake is negotiated between two endpoints, a TLS handshake is performed as well. QUIC also specifically requires TLS 1.3 and cannot operate on lower versions. UDP is used as the actual underlying transmission protocol. Every QUIC message frame is transmitted over a regular UDP packet. This ensures the best possible hardware compatibility as the introduction of an actual standalone transport protocol would mean huge efforts to update all network components that may sit between two connecting hosts.

QUIC Handshake. In principle, QUIC makes use of the TLS handshake as described in Chapter 2.3.1. However, QUIC brings down the number of round trips to one or zero, depending on the handshake type. This is achieved mainly by allowing multiple packets within the same UDP packet. QUIC also takes responsibility for transporting the handshake messages with their own `CRYPTO` frame, and therefor adapts the TLS handshake protocol.

- 1-RTT handshake. This type is used when a new connection between two endpoints is established and is visualized in Figure 2.8 as an example. Every line represents a QUIC packet. The cryptographic parameters are negotiated using the `INITIAL` packet, that contains `CRYPTO` frames. The `STREAM` frame transport encrypted data, which takes place at the earliest on the last line in the server response in Figure 2.8, making it a potential 0.5-RTT handshake.

- 0-RTT handshake. This type of handshake is used when a connection has previously been established and is resumed. The packets that are sent are identical to the ones in the 1-RTT handshake, but instead of waiting for the server to respond, the client can already send data along with the initial packet.

```
Client                                                   Server

Initial[0]: CRYPTO[CH] ->

                                 Initial[0]: CRYPTO[SH] ACK[0]
                       Handshake[0]: CRYPTO[EE, CERT, CV, FIN]
                                 <- 1-RTT[0]: STREAM[1, "..."]

Initial[1]: ACK[0]
Handshake[0]: CRYPTO[FIN], ACK[0]
1-RTT[0]: STREAM[0, "..."], ACK[0] ->

                                          Handshake[1]: ACK[0]
         <- 1-RTT[1]: HANDSHAKE_DONE, STREAM[3, "..."], ACK[0]
```

FIGURE 2.8: Example of a QUIC 1-RTT handshake. [16]

QUIC Record Layer. The QUIC standard defines a version independent short and long header in RFC 8999 [18]. RFC 9000, which specifies the usage of QUIC over the UDP protocol, specifies that data has to be transmitted using QUIC packets. An UDP packet may contain one or more QUIC packets. Each packet either uses a long or a short header. Long headers are used during connection establishment and short headers afterwards. The 1-RTT packet is the only packet that uses the short header format and is used after the version and 1-RTT keys were negotiated. As 1-RTT packets omit the packet length within the header, they can only be transmitted in one UDP packet. If transmitted together with other packets, they have to be at the end of the packet chain.

QUIC packets contain one or more different types of frames. Frames are used to actually transport protocol relevant data. User data is transmitted using `STREAM` frames. Illustrated in Figure 2.9 is the complete structure of a QUIC 1-RTT packet that contains one `STREAM` frame to transmit data but would not be necessarily limited to just one frame. The green parts show the 1-RTT packet header and the blue parts the header of the `STREAM` frame.

- **HF (Header Form):** Indicates which header type is used. In case of the 1-RTT packet this bit is always set to 0 which stands for the short header format.

- **FB (Fixed Bit):** Fixed bit that is always set to 1.

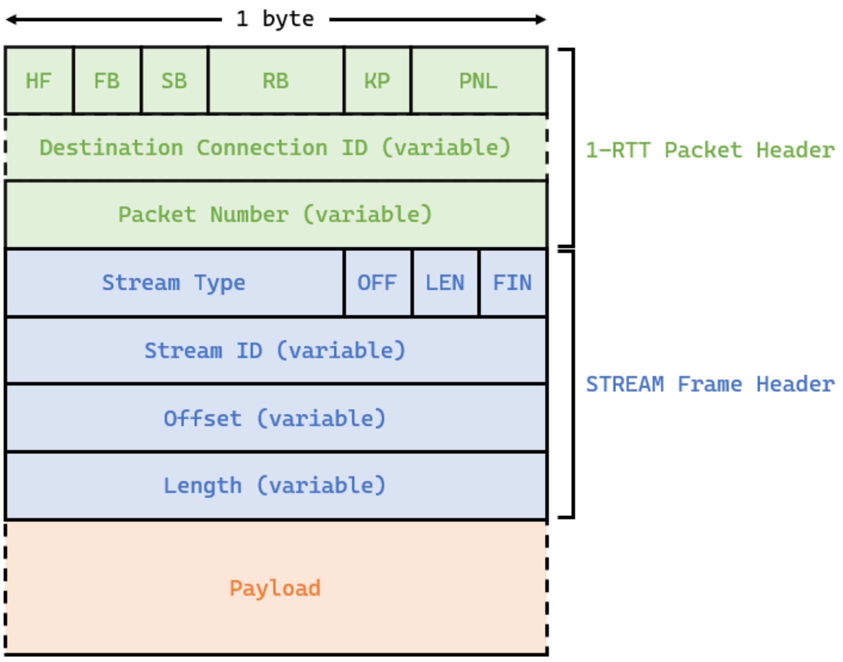


FIGURE 2.9: Structure of a 1-RTT QUIC packet containing a `STREAM` frame.

- **SB (Spin Bit):** Optional feature that allows on-path observers to measure the time between the spinning of this bit.
- **RB (Reserved Bits):** Two bits that are reserved but serve no further purpose.
- **KP (Key Phase):** Can be set in case a key rotation occurs.
- **PNL (Packet Number Length):** Indicates the length of the packet number field in bytes + 1. For instance, the binary value of 0b11 would indicate that the packet number field has a length of 4 bytes.
- **Destination Connection ID:** Optional field that could be used to mitigate changes on lower protocol layers and to ensure that the packet still arrives at the intended location. For instance, if the packet is changed due to network address translation (NAT) procedures, a receiver may still recognize the package.
- **Packet Number:** Number that identifies the packet and is used to acknowledge packets.
- **Stream Type:** The 6-bit stream type always contains the binary value 00001 representing the `STREAM` frame type.
- **OFF:** Indicates if the offset field is present or not. If missing, transmitted data is considered as payload starting from offset 0.
- **LEN:** Indicates if the length field is present or not. In case the length field is omitted, the payload is read until the end of the packet containing packet.
- **FIN:** Indicates if the frame marks the end of a virtual stream.

- **Stream ID:** Identifies the stream this data belongs to. There may exist multiple streams which are transmitted over a physical QUIC connection. A stream may either be unidirectional or bidirectional.
- **Offset:** Byte offset within the payload of the `STREAM` frame.
- **Length:** Byte length of the `STREAM` frames payload.

The Destination Connection ID, Stream ID, Offset and Length fields are all encoded using QUICs variable length integer encoding. This encoding is used in most fields that contain integer values to avoid the presence of additional fields indicating the length of the integer field.

2MSB	Length	Usable Bits	Maximum
00	1	6	$2^6 - 1$
01	2	14	$2^{14} - 1$
10	4	30	$2^{30} - 1$
11	8	62	$2^{62} - 1$

TABLE 2.1: All four possible integer lengths with their encoding.

The two most significant bits indicate how long the integer value actually is. For instance, if the binary value 0b00 is used, the field has a total length of 8 bits. Excluding the two most significant bits, 6 bits are usable for the integer value itself which allows for a range of 0 to $2^6 - 1$.

This means that in the best possible case, `STREAM` frames have a total overhead of 4 bytes which would include the first byte which contains the header flags, the 8 bit packet number, the byte indicating the stream type and the 8 bit long stream id. This excludes potential message authentication codes.

2.3.5 COSE

COSE [19][20] stands for CBOR Object Signing and Encryption and is a relatively young security solution with its first RFC publication dating from 2017 [21]. CBOR (Concise Binary Object Representation) is a standard that extends the JSON format to allow for binary data and other formats to be encoded and is well suited as a message encoding format for IoT purposes. COSE makes use of CBOR for serializing the data to be encrypted and sent.

COSE Message Structure. In contrast to the other security protocols discussed here, COSE operates on the Application Layer on a per-message basis. Each message has the same structure:

- Protected header (as a CBOR "bstr" type)
- Unprotected header (as a CBOR "map" type)
- Payload

The protected header is used for parameters that need to be cryptographically protected. The presence of both buckets is required in every message. If one bucket is empty, it has to be encoded as a zero-length string or map.

COSE Header Structure. The header parameters are not fixed but there exist some common COSE header parameters that are used in a majority of all messages. These parameters are introduced below:

- **alg**. Designated cryptographic algorithm to use.
- **crit**. Critical header parameters that must be understood by the receiving application.
- **content type** of payload.
- **kid**. The key identifier describes a piece of data that can be used as input to find the needed cryptographic key.
- **IV**. Full Initialization Vector.
- **Partial IV**. This field is used to carry a value that causes the IV to be changed for each message.

The first two parameters are placed in the protected header bucket whereas the latter three belong in the unprotected header bucket.

COSE Message Types. COSE messages use a layer concept to differentiate cryptographic concepts such as encryption and signing. There are (at least) seven message types:

- **Encrypt0**. Encrypted structure with a single recipient. Key is implicitly known.
- **Encrypt**. Encrypted structure with (potentially) multiple recipients.
- **MAC0**. MAC authentication structure with a single recipient.
- **MAC**. MAC authentication structure with (potentially) multiple recipients.
- **Sign1**. Signed message with a single signature.
- **Sign**. Message signed by multiple entities.
- **COSE_Key/COSE_KeySet**. Structure for establishing a common secret and/or further specifications of how to apply a cryptographic algorithm.

These message structures can be nested to achieve confidentiality, integrity as well as authenticity of the messages at the same time.

2.3.6 WPA

As its name suggests, Wi-Fi Protected Access (WPA) was developed for securing Wi-Fi connections [22]. It is the successor of WEP (Wired Equivalent Privacy) and has evolved from WPA over WPA2 to its most recent version WPA3, which came with some improvements concerning security. Earlier versions used the Temporal Key Integrity Protocol (TKIP) to encrypt data but starting from WPA2, TKIP was exchanged for the more secure Advanced Encryption Standard (AES). WPA operates in two modes: personal mode, which uses a pre-shared key for authentication, and enterprise mode, which works with an authentication server [23].

WPA was designed specifically for wireless computer networks. Since the connection to the hearing aid runs on Bluetooth Low Energy, WPA would have to be adapted for our case. Our research has not yielded any results regarding a BLE implementation of WPA, hence we discard this option and exclude WPA from further investigation.

2.4 Transport and Internet Layer

Most of the protocols that were discussed in Chapter 2.3 expect the IP protocol on the internet layer and the User Diagram Protocol (UDP) or the Transmission Control Protocol (TCP) on the transport layer. While some of them do not technically require these protocols, libraries that implement security protocols are usually tightly coupled with IP and TCP/UDP protocols. After an overview of the internet protocol IPv6 and the transport protocols TCP and UDP, this chapter discusses possibilities of reducing the overhead that stems from these protocols. Furthermore, Chapter 2.4.5 presents the possibilities of transmitting IP packets over Bluetooth low Energy networks, which is currently being used as wireless transmitting technology for the hearing aids.

2.4.1 IPv6

The Internet Protocol version 6 (IPv6) is the latest version of the IP protocol and specified in RFC 8200 [24]. One of the main advantages of IPv6 compared to IPv4 is the much larger IP address space. Instead of just 2^{32} possible addresses, IPv6 allows the creation of 2^{128} addresses. The larger address space makes it highly suitable for utilization in the IoT sector where it is also subject in many standards. IP compression methodologies described in Chapter 2.4.4 and the operation of the IP protocol over BLE described in Chapter 2.4.5 both require IPv6.

IPv6 Header. The IPv6 header was slightly modified compared to the IPv4 header. Fields were removed or renamed and new fields were added.

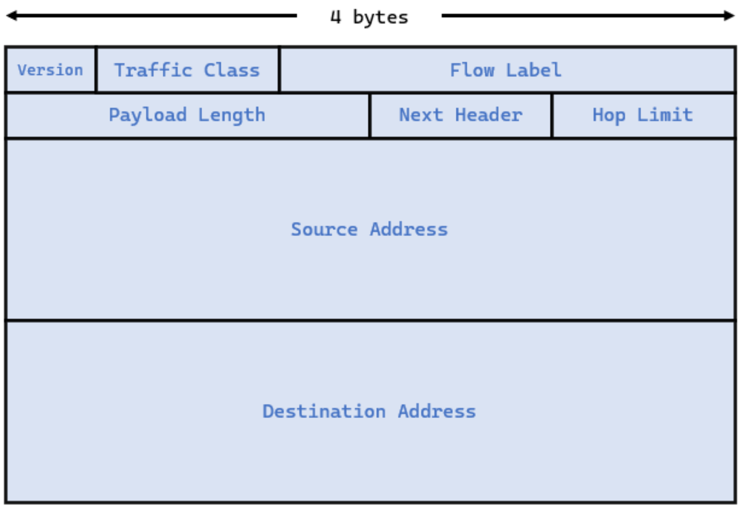


FIGURE 2.10: IPv6 header layout.

As to be seen in Figure 2.10 the IPv6 header has a total length of 40 bytes and consists of following fields:

– **Version:** Static field that always contains the binary value of 6.

– **Traffic Class:** Can be used by the network to perform traffic management.

– **Flow Label:** Allows the identification of a single flow of packets between a sender and receiver and is specified in RFC 6437.

– **Payload Length:** The length of the payload without the header. Extension header that may follow are also considered as payload.

– **Next Header:** The next header field allows to specification of a next header that immediately follows the IPv6 header. The value corresponds to a protocol number as specified by the IANA organization.

– **Hop Limit:** Is decremented by one if the packet passes through a node (router). Once the value reaches zero, the packet is discarded.

– **Source and Destination Address:** The 128 bit source and destination address. Detailed specification for the addressing architecture is specified in RFC 4291.

2.4.2 TCP

The Transmission Control Protocol (TCP), defined in RFC 9293 [25], is a transport-layer protocol responsible for establishing and maintaining a connection between two hosts over a network [26]. As such, it serves the same purpose as the UDP protocol, but ensures that no packet is lost during transmission, maintains the order of arrival and provides error check mechanisms. This makes it suitable for many applications, for example HTTP and HTTPS connections. TCP divides the data stream into packets and passes them to the IP protocol layer after it has equipped each packet with a header containing relevant information about the sender, receiver and connection. The format of the TCP header is shown in Figure 2.11 with its fields being explained hereinafter.

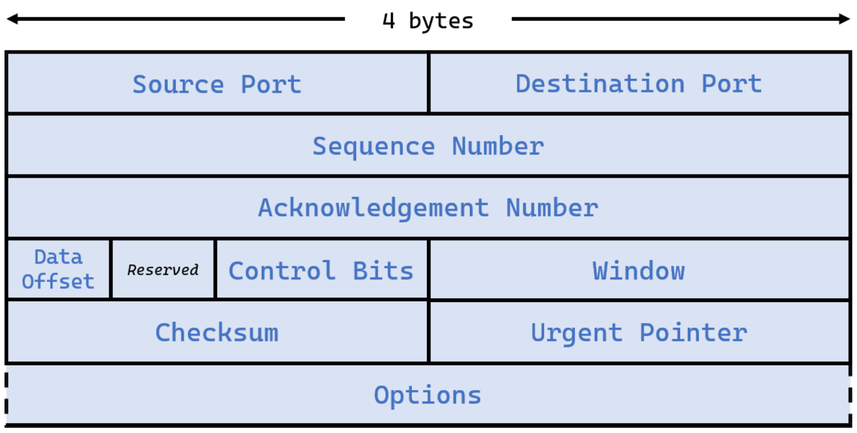


FIGURE 2.11: TCP header layout.

– **Source and Destination Port.** 16-bit numbers denoting the sender and the receiver port, respectively.

- **Sequence Number.** Each byte of data that is sent over a TCP connection has a unique sequence number. This field carries the sequence number of the first data byte in this packet.
- **Acknowledgement Number.** This field acknowledges the reception of a specific data quantity by containing the sequence number expected next.
- **Data Offset.** Indicates where the data begins.
- **Control Bits.** Flags that control the use of some header fields.
- **Window.** Number of data bytes that the sender is willing to accept.
- **Checksum.** The checksum is used to control errors and ensure the integrity of the sent data.
- **Urgent Pointer.** This field indicates which of the data bytes in the payload are urgent. It is to be interpreted as an offset from the sequence number. This field is only sent when the URG flag is set. It is not advisable anymore to use this mechanism whatsoever [25].

2.4.3 UDP

The User Datagram Protocol (UDP) is, in contrast to TCP, an unreliable and connectionless protocol and therefore requires no prior handshake. This makes it suitable for performance critical applications such as voice or video transmission where occasional packet loss is usually accepted. Illustrated in Figure 2.12 is the header format for a single UDP packet.

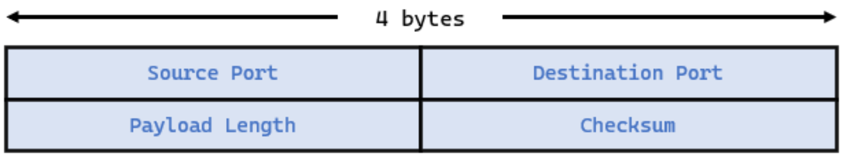


FIGURE 2.12: UDP header layout.

- **Source and Destination Port.** Port number of the sender and receiver of this UDP packet.
- **Payload Length.** Indicates the length of the payload in bytes which directly follows the header.
- **Checksum.** If used, the checksum field provides the possibility to perform error checking of the received payload. If unused, it is filled with zeros.

2.4.4 Header Compression

Header compression allows the reduction of the overhead that is created by the protocol headers. This chapter presents some of these compression methodologies.

Robust Header Compression. Robust Header Compression (ROHC) is a efficient compression scheme for UDP/IP and ESP/IP headers as defined in RFC 3095 [27].

It was developed specifically with situations in mind where packets get lost or damaged easily and should provide robust compression even under those aggravated circumstances. The fundamental idea is to only transmit information that has changed since the previous packet. This leaves only five fields that cannot be compressed away completely, and one of the remaining ones only needs to be transmitted occasionally, and two of them – IP ID and RTP timestamp – can be predicted by the sequence number. Therefore, the sequence number and the UDP checksum have to be transmitted at all times, and functions have to be established that predict the IP ID and the RTP timestamp from the sequence number. These functions need to be updated once in a while to ensure the correct prediction of the fields. Unfortunately, ROHC is computationally too complex and still generates too much traffic overhead for embedded devices [28] which erases it as a compression option for our case.

LOWPAN_IPHC 6LoWPAN was a working group of the Internet Engineering Taskforce (IETF) that defined multiple standards to operate the IPv6 protocol over low-rate wireless personal area networks (LR-WPAN). LR-WPAN networks are defined by the technical standard IEEE 802.15.4 [29] which specifies the physical layer and the media access control sublayer of such networks. LR-WPAN networks focus on low-data-rate for resource constrained devices with no battery or very limited battery consumption. The MTU (maximum transfer unit) is reduced to just 127 bytes which leads to only 80 bytes of payload data on the MAC sub-layer. As a consequence, it is required to optimize the overhead generated by IPv6 headers. Therefore, stateless header compression was first standardized in RFC 4944 [30] using the LOWPAN_HC1 (header compression 1) encoding format that is used to compress IPv6 headers and the optional LOWPAN_HC2 (header compression 2) encoding format, which allows compressing some transport layer protocols like UDP [28]. As these compression methods are stateless, the receiver has to be able to derive all IPv6 header fields from the data which is present in the received frame. This makes the LOWPAN_HC1 and LOWPAN_HC2 encoding formats insufficient in most practical use cases of IPv6 over LR-WPAN networks.

For this reason, a new stateful header compression encoding format LOWPAN_IPHC (IP header compression) was standardized in RFC 6282 [31], as illustrated in Figure 2.13.

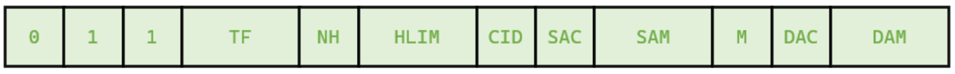


FIGURE 2.13: LOWPAN_IPHC encoding.

Instead of transmitting the full IPv6 header, it is replaced with the 2 byte long encoding format followed by the IPv6 header fields either compressed or uncompressed depending on which bits are set in the encoding scheme. In any case, the header fields have the same order as they do in the IPv6 header format [24]. The version and payload length fields are always fully elided, as the former is a static value that never changes and the latter can be derived from lower layers such as the IEEE 802.15.4 header.

The encoding starts with a dispatch value to indicate the presence of a IPv6 header that was compressed using the LOWPAN_IPHC encoding format, and is followed by bit flags that define how certain IPv6 header fields are compressed:

– **TF (Traffic Class, Flow Label):** Defines how the traffic class and flow label fields are compressed.

– **NH (Next Header):** Indicates if the next header field is either not compressed and fully carried in line or compressed using the LOWPAN_NHC encoding discussed in Section 2.4.4.

– **HLIM (Hop Limit):** Indicates if the hop limit field is either carried fully in-line or the field is elided and takes one of the predefined values of 1, 64 or 255.

– **CID (Context Identfier Extension):** Indicates if the context identifier extension is used. If used, an additional 8 bit context identifier extension field follows the DAM field.

– **SAC (Source Address Compression):** If set to 0, stateless compression is used for the source address. If set to 1, sateful context-based compression is used.

– **SAM (Source Address Mode):** Defines, together with the SAC flag, how the source address should be compressed.

– **M (Multicast Compression):** Indicates if the destination address is a multicast address or not.

– **DAC (Destination Address Compression):** If set to 0, stateless compression is used for the destination address. If set to 1, stateful context-based compression is used.

– **DAM (Destination Address Mode):** Defines together with the M and SAC flag, how the destination address is compressed.

If the CID flag is set to 1, the 8 bit context identifier extension is added after the DAM field. The extension encoding is illustrated in Figure 2.14.

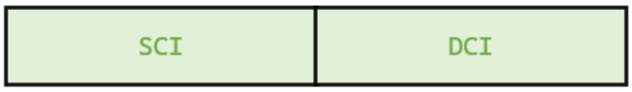


FIGURE 2.14: Context Identifier Extension encoding.

The specification expects that a context was shared between the sender and receiver. The extension field contains two context identifiers: One for the source address (SCI or source context identifier) and one for the destination address (DCI or destination context identifier). These contexts are then used to compress and decompress either of the source and destination address fields. The specification however does not specify how these contexts should be shared and maintained nor the content of such contexts.

LOWPAN_NHC If the NH bit within the previously discussed LOWPAN_IPHC encoding scheme is set to 1, the next header is compressed using the LOWPAN_NHC encoding format. The next header field within the IPv6 header is also omitted, as the type of next header can now be determined with the NHC ID. The NHC ID is a variable length bit pattern that identifies the type of header that is compressed. Together with the remaining bits to control the compression, the LOWPAN_NHC encoding format is formed with a size of one byte. Following this byte are the fields of the next header, either compressed, uncompressed or completely elided depending on the bit settings. They have the same order as they would in their original header structure.

One requirement to use the LOWPAN_NHC encoding format is that each preceding header is compressed using either the the LOWPAN_IPHC or LOWPAN_NHC encoding format. The standard therefoer specifies two specific encoding formats to compress a selected set of IPv6 extension headers and the UDP header. The encoding for IPv6 next headers is illustrated in Figure 2.15.

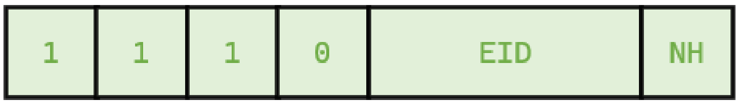


FIGURE 2.15: LOWPAN_NHC encoding.

The type of IPv6 next header that is compressed is encoded in the 3 bit long EID field. If the NH bit is set to 1 again, the following header is compressed using LOWPAN_NHC again. IPv6 extension headers are mostly carried unchanged after the LOWPAN_NHC encoding byte. The main reason to encode IPv6 extension headers is to allow their usage with the UDP compression format discussed next. Two exceptions exist: If the NH bit is set to 1, the next header field is fully elided because it can be derived from the LOWPAN_NHC byte that follows the currently compressed next header. Furthermore, in contrast to an uncompressed extension header, the length field indicates the number of bytes that immediately follow the length field instead of indicating the size in units of 8 bytes. This reduces unnecessary wasted space within the header.

The standard furthermore defines the compression of UDP headers as illustrated in Figure 2.16. This byte is again follows by the compressed or uncompressed UDP header fields depending on the bit settings.

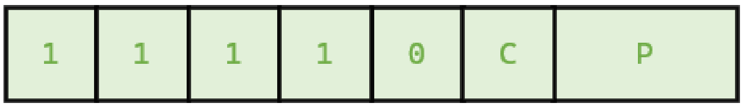


FIGURE 2.16: LOWPAN_NHC UDP encoding.

- **C (Checksum):** Indicates if all 16 bits of the checksum value are carried in-line or the checksum is fully omitted. Elding the checksum requires some sort of other verification, either provided by the lower 6LoWPAN layer or by other protocols such as IPSec.
- **P (Ports):** Either the full 16 bits for both port numbers are carried in-line or for either the destination or the source port the first 8 bit are set to the fixed value of 0xf0 and the remaining 8 bits are carried in-line. Finally, it is also possible to set the first 6 bits of source and destination port to the fixed value of 0xf0b and only the remaining 4 bits are carried in-line.

IPSec Header Compression. Shahid Raza et al. [32] presented a 6LoWPAN IPSec extension for the header compression methodologies standardized in RFC 6282 [31].

RFC 6282 currently specifies two types of header compression methods based on the LOWPAN_NHC encoding format described in previous Chapter 2.4.4. This includes the compression of IPv6 headers where the `EID` field describes the type of next header that is compressed but without the IPSec ESP and AH headers. Currently, binary value 0b101 and 0b110 are reserved and therefor unused. The paper proposes three ways to use the two free slots to compress IPSec headers:

1. One of the two reserved slots is used to indicate that the following header is either an AH or ESP header that is encoded using the LOWPAN_NHC encoding format. The type is determined by the NHC ID within the encoding.
2. Use both slots to differentiate between either AH and ESP header encoding. This would allow omitting the NHC ID in the following header but this would go against the specification in RFC 6282.
3. Define a third IPSec header LOWPAN_NHC encoding format which would take up one more byte but would allow to differentiate how upper layers are compressed.

As the AH protocol is irrelevant in the context of this document, only the ESP LOWPAN_NHC encoding is illustrated in Figure 2.17.

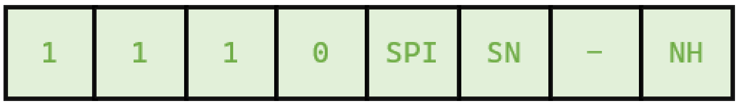


FIGURE 2.17: LOWPAN_NHC ESP encoding.

- **SPI:** If set to zero, the SPI field is omitted and a default pre-defined value is used instead.
- **SN:** If set to zero, the sequence number field length is reduced from 32 to 16 bits.
- **NH:** Indicates if the following header again is compressed using the LOWPAN_NHC encoding format.

In the best case scenario, this reduces the overhead by 6 bytes. However, if the ESP format is used, upper layer protocols such as UDP cannot be compressed as the 6LoWPAN gateway has no possibility to expand the UDP header as it is encrypted. This would require the definition of a algorithm for ESP that could handle both, compression and encryption of UDP headers. This also hinders the leverage of the third proposal of introducing a third IPSec header LOWPAN_NHC encoding format, to allow more flexibility regarding compression of upper layer protocols. Due to encryption, the same compression features would need to be implemented in the ESP module and this option is therefore also excluded from further discussion.

Another draft standard proposed by S. Raza et al. [33] presents yet another LOWPAN_NHC format illustrated in Figure 2.18 for AH and ESP headers and allows the reduction of one more byte.

Here, the SPI and SN fields control what is carried in-line for their respective fields. The SPI field is either completely elided and a default value is used or only the least

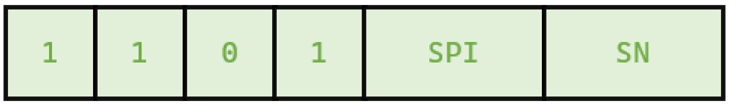


FIGURE 2.18: LOWPAN_NHC ESP encoding.

significant 8 or 16 bits are transmitted. In case the flag is set to 0b11, all 32 bits of the SPI field are carried in-line. The SN flag indicates four different lengths: 8, 16, 24 or 32 bits of the sequence number are carried in-line. Using this encoding, in the best case scenario, the overhead would be reduced by 7 bytes.

TCP/IP Header Compression The TCP/IP header compression was suggested in 1990 by Van Jacobson (therefore also called Van Jacobson compression) and specified in RFC 1144 [34]. The basic idea behind it is not to send information that does not change but to store it instead – and if it changes, only send the difference to the most recently stored value. This results in a TCP/IP header of three bytes in the best case (see Figure 2.19).

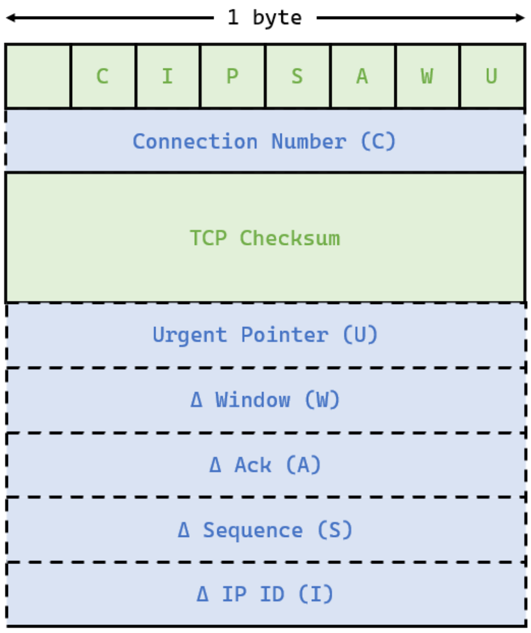


FIGURE 2.19: Compressed TCP/IP header. [35]

As it can be seen in Figure 2.19, the compressed header consists of a change mask (byte 0) and 2 bytes of TCP checksum, the only information that cannot be omitted under any circumstances. Sometimes, the transmission of additional fields is required, which is signaled in the change mask. The remaining header fields are explained in the following.

– **Change Mask.** Every bit (except the first) in the change mask signifies if a specific field in the header is present or absent. The meaning of the letters can be determined by the letters in brackets in the fields below.

– **Connection Number.** Information on where the last saved copy of an uncompressed packet is stored that can be used for (re-)synchronization.

– **Window, Ack, Sequence.** These fields only contain the difference to the last value, if needed at all.

– **IP ID.** This is the ID of the packet. Unlike the other fields, when the flag in the change mask is not set, the ID is automatically assumed to be incremented by one.

An implementation of the Van Jacobson header compression can be found in the Point-to-Point Protocol, which is a data link layer protocol.

2.4.5 IPv6 over BLE

Bluetooth Low Energy (BLE) has many similar characteristics to the link layer of IEEE 802.15.4 networks. RFC 7668 [36] therefore makes use of the IPv6 optimizations specified by the Lo6WPAN working group like the header compression methologies presented in Section 2.4.4 and applies it to the BLE stack.

Protocol Stack. Figure 2.20 illustrates the protocol stack that is used to operate IPv6 over BLE networks. Green parts indicate existing components that stem from the Bluetooth standard and blue parts indicate new parts.

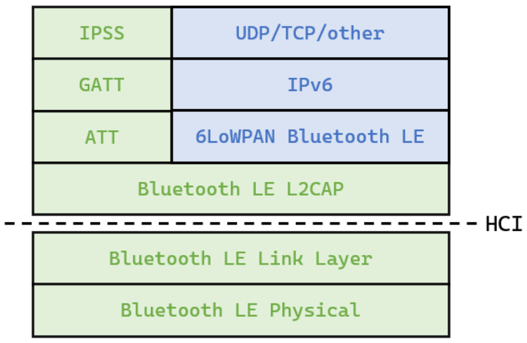


FIGURE 2.20: Bluetooth Low Energy (BLE) stack with IPv6 support.

For IPv6 operation over BLE networks, the standard relies on Bluetooth version 4.1 and the Internet Protocol Support Profile (IPSP) version 1.0. IPSP includes the Internet Protocol Support Service (IPSS), which allows the discovery IP-enabled devices. This means the IPv6 stack works in parallel to the GATT stack which enables the discovery of nodes that support IPSS. Between IPv6 and L2CAP sits the 6LoWPAN Bluetooth LE layer which applies optimizations to the IPv6 protocol. The L2CAP layer provides functionalities such as multiplexing, fragmentation and reassambly of packets and communicates with lower layers using the host control interface, which separates higher layers that are usually operated on the host with lower layers that are usually operated on the Bluetooth controller. As the L2CAP layer already provides fragmentation and reassembly functionality, the optimizations specified in the 6LoWPAN standard regarding fragmentation are not used and the full 1280 bytes are transmitted. It is recommended, however, to not exceed 1280 bytes to avoid path MTU discovery procedures.

Topology and Roles. Contrary to IEEE 802.15.4 networks, BLE networks only supports star topologies and no mesh topologies. The BLE standard currently defines two roles which are relevant for IPv6 to operate over BLE: The BLE central role and the BLE peripheral role. A BLE central and BLE peripheral role can either be a 6LoWPAN Border Router (6LBR) or a 6LoWPAN Node (6LN) according to the 6LoWPAN specification. Illustrated in Figure 2.21 are two scenarios how the BLE network can operate. In the scenario on the left, the network is connected to the internet. In case a 6LN wants to send or receive data from the internet, the traffic needs to be routed over the 6LBR component which acts as a router. In other scenarios, the network may be operated completely isolated, as illustrated on the right. In any case, communication between two 6LN components need to be routed over the 6LBR component.

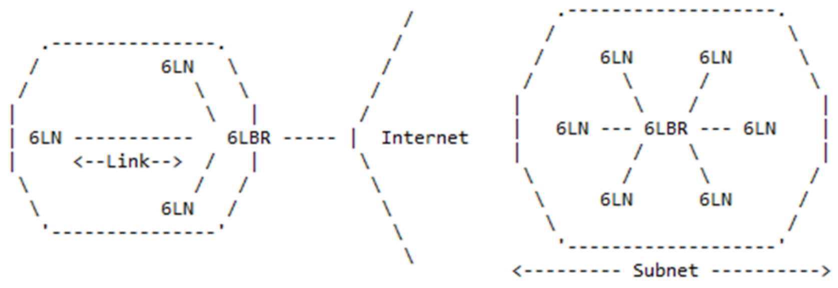


FIGURE 2.21: Possible network topologies for Bluetooth LE networks. [36]

IPv6 Stateless Address Autoconfiguration. Each Bluetooth device is identified by a 48-bit device address. The Bluetooth standard specifies different address types for which the RFC recommends to use the private address type. Using this device address, a 64-bit interface identifier (IID) is generated by inserting 0xFF and 0xFE in the middle of the device address. The IID is then prepended by fe80::/64 to form the link-local IPv6 address for this Bluetooth device.

Header Compression. Header compression is required for all IPv6 traffic that travels over BLE networks. The specification differentiates between stateless and stateful header compression using the LOWPAN_IPHC encoding format. For link-local addresses, stateless header compression can be used and source and destination must be completely elided, meaning the SAC flag is set to 0 and the SAM flag to 0b11.

In case source or destination address are non-link-local addresses, the standard requires the use of stateful header compression. This is accomplished with a Address Registration Option (ARO) where either a 6LN or 6BLN component must share their their respective non-link-local address with each other using a Neighbor Advertisement (NA) message. If the source or destination address is the latest address, which was communicated using ARO, source or destination addresses must be completely elided.

2.5 Cipher Suites

Cipher suites are a set of cryptographic algorithms that secure a connection over a network between two endpoints [37]. Thus, they make up the beating heart of network security and their choice is of crucial importance because a cryptographically broken algorithm can jeopardize the CIA triad. A cipher suite usually comprises a key exchange algorithm, an encryption algorithm, and a MAC algorithm, with the first two covering confidentiality and authenticity and the last one providing for integrity.

As already mentioned in Chapter 2.1, the cipher suites that can be used on the target system are limited due to hardware constraints. We are therefore going to focus on those cipher suites that are supported by our target system and will shortly explain how they work and what their implications on the communication overhead are.

2.5.1 ECC/RSA

These algorithms come under asymmetric cryptography (also called public-key cryptography). Asymmetric cryptography is used for exchanging keys and thereby establishing a common secret as well as for signing data and the verification of this signature. The mechanism of asymmetric cryptography relies on a key pair of which one key is public and the other one private to the specific host. The keys are matching insofar that only the private key can be used to decrypt something that was encrypted with the public key. For establishing a common secret with this host, a client will use the host's public key for encryption so that only the host possessing the corresponding private key can decrypt it.

RSA (Rivest–Shamir–Adleman) is one of the oldest asymmetric cryptographic algorithms [38]. It rests upon the factorization of large prime numbers and the now-a-day standard uses 2048-bit keys. **ECC (Elliptic Curve Cryptography)**, on the other hand, is based on elliptic curves over finite fields and one of the more recent inventions in asymmetric cryptography. It is much more efficient than RSA in terms of key size and computational power, as a greater cryptographic strength can be achieved with a shorter key. For example, a 256-bit ECC key attains equivalent security as a 3072-bit RSA key [39]. None of their properties affect the communication overhead, though, because the key exchange usually takes place only once in a client-server session (or even fewer in case of session resumption using pre-shared keys).

As of today, RSA and ECC algorithms are considered to be secure, even when taking into account quantum computers. Nevertheless, both RSA and ECC rely on the factorization principle that can theoretically be cracked by the Shor algorithm, which implies that some day, quantum computers might become powerful enough to solve the factorization problem even for 2048-bit keys [40][41].

2.5.2 AES-128 and AES-256

AES (Advanced Encryption Standard) [42] is a symmetric cryptography algorithm where the same key is used for encrypting and decrypting the data. It succeeded DES (Data Encryption Standard) which was first broken in 1997. AES is FIST-approved [43] and considered one of the most secure crypto algorithms today. As of 2023, it is believed that it is even quantum resistant [44]. Its protection varies with the key length, however, which can be 128, 192, or 256 bits and is appended to its name. In security protocols, they serve as a means to encrypt the data that is to be exchanged between two endpoints as soon as a shared secret is established by the use of an asymmetric cipher.

AES is a block cipher algorithm that operates on data chunks of 128 bits [45]. These chunks are separately converted into cipher-text chunks and then joined back together. Instead of using the same key for every encryption, a key schedule algorithm will compute round keys from the initial key that are then used for encrypting the data in the specific round. Depending on its mode of operation, AES might require padding and/or an initialization vector which would lead to a larger communication overhead. Some of these modes allow for authentication besides encryption, e.g. GCM (Galois-Counter-Mode) and CCM (Counter with Cipher Block Chaining-Message Authentication Code), which are also AEAD (Authenticated encryption with associated data) schemes that provide for integrity and authenticity not only of the payload but also of the (plain-text) header [46]. Because of their enhanced security, AEAD ciphers are preferred by TLS, and they also have a faster execution time than algorithm pairs implementing encryption and authentication separately [47]. That's why we focus only on AEAD modes of AES here, namely GCM and CCM.

GCM and CCM are both based on counter mode (CTR), which means that they use one or multiple counters for encryption that are steadily incremented by one. In TLS 1.3, both modes derive the initialization vector from the cryptographic context established during the handshake phase and then XOR it with the sequence number of the specific packet, thus guaranteeing its uniqueness for each message [48]. This prevents the need to send the initialization vector with every message during the record phase, as it was still the case in TLS 1.2 [49][50]. Despite their commonalities, the internal workings of the modes differ markedly. We will not go into any details here and will only mention two differences that are relevant for the goal of our thesis. First, GCM encrypts the messages before it computes and adds a MAC, as opposed to CCM, which adds the MAC before it encrypts the message [51]. Secondly, while CCM uses CMAC as a MAC algorithm, GCM makes use of the GHASH hash function [52]. These differences will be reconsidered and evaluated in Chapter 3.3.1.

2.5.3 SHA-2 and SHA-3

As its name implies, SHA (Secure Hashing Algorithm) is a family of functions used for hashing data and thereby providing integrity. A hash function takes some data as input and outputs a fixed-sized string that has no inherent relation to the original input. In that sense, hash functions are one way as it is virtually impossible to derive the original data from its hash value. Security protocols use hash functions to calculate a unique MAC value, thus ensuring the integrity and authenticity of the data, as well as to derive random secret keys from a shared secret [53].

Hash functions can be evaluated on the basis of three cryptographic characteristics: pre-image resistance, second pre-image resistance, and collision resistance [54]. **Pre-image resistance** denotes the difficulty to find an original input when given a hash digest value. If a hash algorithm is pre-image resistant, it means that it is hard even for a high-performance computer to succeed with a brute-force attack. **Second pre-image resistance** means the difficulty to find a (different) message with the same hash value as a given message. Lastly, **collision resistance** is an even stronger property, demanding that it is hard to find *any* two different messages that output the same hash value. These characteristics are achieved when the hash function's output values are all equally likely to be produced (uniformly distributed), when a small change in the input data results in a large change in the digest value, and when the output value has a reasonable size (default size in TLS 1.3 AEAD MACs: 16 bytes). This output value has to be transmitted with every hashed message for verification purposes, thus contributing to the communication overhead. The implications of a hash value size reduction are discussed in Chapter 3.3.2. As for the general collision resistance, SHA-2 algorithms are commonly considered to be in a good position because they are designed according to the Merkle-Damgard construction, which aims exactly at enhancing this property [55]. Hash functions of the Merkle-Damgard construction design are vulnerable to freestart collision attacks, however [56]. Since these attacks are hardly feasible due to the requirement of selectable initialization vectors, though, they do not pose an actual threat. SHA-3 algorithms, on the other hand, are designed according to sponge construction. Since SHA-3 is relatively new, finalized only in 2019, and differs drastically from SHA-2 in its internal workings, it has not been studied as well as SHA-2 yet. But findings so far show that SHA-3 is about as collision-resistant as SHA-2 [56].

Chapter 3

Evaluation

Having laid the foundation, we will conduct an evaluation of the security protocols presented in the previous chapter and of the libraries implementing them. This concerns implementations of the security protocols itself but also of the underlying transport protocol. Subsequently, the implications of a chosen cipher suite and operation mode will be discussed. According to the goal of this thesis – minimization of the overhead in hearing aid communication – the evaluation primarily takes into account how the traffic overhead can be reduced by using suitable protocols and implementations.

3.1 Security Protocols

As we have seen in Chapter 2.3, different security protocols come with different conceptual approaches that have differing effects on the performance of a communication channel. An overview of the communication overhead that is generated by the security protocols evaluated in this thesis can be found in Table 3.1.

The rows in Table 3.1 represent the security protocols, with QUIC as some kind of exception, as it can be treated as a security protocol as well as a transport protocol. The column "Protocol Overhead" denotes the overhead generated by the security protocol itself whereas the two columns to the left designate the underlying layers which contribute additional overhead. Except for TLS over TCP, the "Transport Protocol" column always indicates the UDP overhead since all the other security protocols run on top of UDP. The column "Total Overhead" shows the minimal overhead of all the layers added together, that is, the expectable overhead on the communication channel in the very best case (note that the application layer is not included here). We will now explain how the overhead of every security protocol comes about, starting with a preliminary remark on the composition of the IPv6 overhead.

Internet Layer (IPv6). We specifically chose the IPv6 as protocol on the internet layer, as RFC 7668 described in Chapter 2.4.5 currently only allows IP version 6. The real absolute worst case scenario would of course be 40 bytes which would relate to the size of the IPv6 header described in Chapter 2.4.1. But as the operation of IPv6 over BLE always requires header compression, we consider the worst case with 6LoWPAN header compression in mind. As the version field and payload length are always elided the worst case for the compressed IP packet is 37 bytes including the 2 bytes from the LOWPAN_IPHC encoding format. Contrary, if every field is compressed using the best possible method, eliding Version, Traffic Class, Flow Control,

Protocol Name	**Internet Layer (IPv6)**	**Transport Layer (UDP/TCP)**	**Protocol Overhead**	**Total Overhead**
IPsec	Best case: 2/3 bytes Worst case: 37 bytes	-	Uncompressed: 13 bytes Compressed: 8 bytes	Best case: 10 bytes
TLS over TCP	Best case: 3 bytes Worst case: 37 bytes	Uncompressed: 20 bytes Compressed: 3-5 bytes	Standard: 9 bytes Best case: 7 bytes	Best case: 10 bytes
TLS over QUIC	Best case: 2 bytes Worst case: 37 bytes	Uncompressed: 8 bytes Compressed: 2 bytes	Best case: 8 bytes	Best case: 12 bytes
DTLS	Best case: 2 bytes Worst case: 37 bytes	Uncompressed: 8 bytes Compressed: 2 bytes	Best case: 7 bytes	Best case: 11 bytes
COSE	Best case: 2 bytes Worst case: 37 bytes	Uncompressed: 8 bytes Compressed: 2 bytes	Best case: 10 bytes	Best case: 14 bytes

TABLE 3.1: Overhead of security protocols.

PayLoad Length, Hop Limit, Next Header and both Source and Destination Address, the IPv6 header generally has a total length of 2 bytes which consists of just the LOWPAN_IPHC encoding format.

IPsec. IPSec has two values for the best case for the internet layer column. It is not specified, which transport protocol is used since IPSec sits below the transport layer and could therefor transport any kind of payload without using a transport protocol. In case the UDP protocol would not be used, the `NH` bit would be set to 0 as only the next header will not be compressed using the LOWPAN_NHC encoding format. This also means that the Next Header field has to be carried in-line after the LOWPAN_IPHC, which would add the additional overhead of 1 byte resulting in 3 bytes of overhead. Chapter 2.4.4 describes non-standardized ways to compress IPv6 headers. In the best case, this could reduce the ESP header overhead by 7 bytes, as the Security Parameter Index (SPI) field is fully elided and the Sequence Number (SN) field is reduced to 1 byte. Combining this with the 1 byte long IPv6 next header LOWPAN_NHC encoding and the 1 byte long ESP LOWPAN_NHC encoding, the actual reduction would be 5 bytes. Assuming a 4 byte long ICV value (of which the implications are discussed in Chapter 3.3.2) and a stream cipher with which the Initialization Vector (IV) and the necessary Padding could be avoided, the overhead results in 8 bytes instead of the uncompressed 13 bytes. If the ESP LOWPAN_NHC encoding is used, the IPv6 could again set the `NH` bit to 1 and the next header field could be elided, which would mean that the total overhead results in 10 bytes. Also note that using the ESP LOWPAN_NHC compression would mean, that the UDP LOWPAN_NHC compression cannot be used as the compressed UDP packet cannot be expanded as it will be encrypted.

TLS over TCP. The default size of the TLS protocol overhead is composed of 5 header bytes and 4 bytes of authentication tag (see Chapter 2.3.1). The legacy protocol version field, which covers 2 bytes, could be omitted as it does not carry any information accessed in the course of the TLS connection. Therefore, an optimized TLS header would consist of 3 bytes, resulting in a protocol overhead of 7 bytes. On the IPv6 layer, the NHC cannot be applied, as there is no specification which defines how TCP headers are compressed using the LOWPAN_NHC, thus extending the best case to 3 bytes. The TCP layer, if uncompressed, adds another 20 bytes of header but could be reduced to 3-5 bytes with Jacobson compression – comprising the whole TCP/IP header. The best case of 3 bytes would be given if also the sequence number and the acknowledge number, together with the other optional fields mentioned in Chapter 2.4.4, were completely omitted. The best case of the entire overhead would therefore be 3 bytes (TCP/IP) plus 7 bytes (TLS) = 10 bytes.

TLS over QUIC. When the TLS protocol is used together with QUIC, the protocol overhead is not the same as with TLS over TCP because TLS is not used on top of QUIC. The overhead resulting from QUIC header is 4 bytes, the short header for data transmission contributing 2 bytes and the `STREAM` frame another 2 bytes. Together with the potential message authentication code (MAC) of 4 bytes, the best case overhead for the QUIC protocol is 8 bytes. On the UDP layer, 8 bytes are appended when using an uncompressed header, and 2 bytes when the header is compressed with LOWPAN_NHC encoding format. This leads to the optimal case of 12 overhead bytes in total.

DTLS. In case of the DTLS protocol, the DTLS header generates an overhead of 3 bytes in the best case. The total overhead of 7 bytes is result of the additional message authentication tag (MAC) overhead of 4 bytes. When using LOWPAN_NHC compression, the overhead generated on the UDP layer can also be limited to 2 bytes. This sums up to the total overhead to 11 bytes.

COSE. When assuming the CIA triad, every COSE message has to be wrapped in 3 message layers: one for encryption (`Encrypt` message type), one for authentication (`Sign` message type), and one for integrity (`MAC` message type). Every message layer has its own header buckets, which even make up at least two bytes in the case of an empty bucket because the encoding of a zero-length string or map already takes one byte. For three message layers, this results in 6 header bytes plus 4 bytes of authentication tag. As with DTLS, NHC can be used on the IPv6 layer and LOWPAN_NHC on the UDP layer, either layer generating 2 bytes of overhead in the best case.

Conclusion. On the basis of these circumstances, several observations concerning the suitability of the evaluated security protocols for our case can be made. Firstly, the IPsec protocol, though showing one of the lowest overall overhead in the table, does not meet our requirement of standardization throughout the protocol stack if compression is applied on the IPsec header. If the header is not compressed at all, the resulting overhead of 16 bytes exceeds all the other protocols by far. A similar situation is given with COSE, whose overhead even in the best case surpasses all the other protocols. This eliminates IPsec and COSE from our range of potential security protocols generating a minimal overhead.

So, we are left with TLS over TCP, TLS with QUIC, and DTLS. Their overhead is almost identical in size. Nevertheless, their workings differ significantly. DTLS assumes an unreliable connection lying underneath. It can be assumed, that the underlying transport protocol on the target device will be reliable so there is no reason, to accept the cryptographic disadvantages, that come with DTLS. This disadvantages may require additional effort which would not be needed when using a security protocol, that expects a reliable transport protocol. Therefore, the winner is TLS which not only exposes a relatively small overhead but is also more easily integrated with the target system preconditions. Using TLS over QUIC also poses an interesting option though, as the protocol provides flexibility regarding header size and is not dependent on a reliable transport protocol at all, as QUIC is transport and security protocol in one. Furthermore, QUIC allows the transportation of multiple packets over the same UDP packet which may increase the throughput. For this reason, it is planned to also verify the QUIC protocol on the evaluation board.

3.2 Implementation Libraries

In this chapter, we inspect several libraries implementing two different layers in the protocol stack: First, five libraries implementing TLS are evaluated, and in a second part, we assess three libraries for the TCP/IP stack. Each evaluation is concluded with a decision on a specific library that will be used in the prototype.

3.2.1 TLS/QUIC Implementations

When implementing TLS as a security protocol, several existing libraries are eligible. Since the library must be suitable for embedded devices, the selection of potential libraries is narrowed down to a handful of libraries that are promoted to be particularly space- and/or power-saving. The libraries that we identified as promising are listed and compared in Table 3.2. All other TLS libraries like openSSL were excluded from further research due to their large code footprint. quant, another library that was brought up for the sake of its support of embedded systems, was discarded because it is still in research and not ready for production use.

As it can be seen in Table 3.2, most of the libraries offer their source code for free, expect in the case of emSSL [57], where it is only available once a license has been purchased by SEGGER. However, SEGGER provides evaluation packs with the precompiled source code under their friendly license agreement to explore the libraries features. EmSSL and wolfSSL [58] both have restrictions on their use which becomes apparent from their licensing system: While wolfSSL requires disclosing the source code unless a commercial license is purchased, emSSL demands acquiring a license as soon as it is used for commercial purposes. The other libraries have no such restrictions.

The most popular library in our sample is mbed TLS [59], followed by wolfSSL, at least when deeming GitHub stars a measure for popularity. WolfSSL, picoTLS [60], and picoquic [61] support TLS 1.3 whereas mbed TLS and emSSL only support up to TLS version 1.2. However, mbed TLS states in their roadmap, that they start with the implementation of TLS version 1.3 in 2023 CQ3 [62]. SEGGER on the other hand told us upon request, that they are also working on a TLS version 1.3 which is about to be completed within this year. As a bonus, wolfSSL also has support for QUIC. The code footprint of the libraries is rather small, as this was a criterion to be evaluated at all, except for picoquic which is based on openSSL via picoTLS.

Documentation is available for all of the evaluated libraries, but to different extents. While picoTLS and picoquic are yet rather poorly documented, wolfSSL, mbed TLS and emSSL come with extensive manuals and examples.

When it comes to hardware acceleration, wolfSSL, mbed TLS, and emSSL explicitly state in their documentation that they support hardware cryptography whereas no such information was found in the documentations of picoTLS and picoquic. Hardware cryptography support is a relevant feature for our case since the hearing aid of our manufacturer is capable of accelerating computationally intensive cryptographic operations and thereby contributing significantly to a better performance.

Finally, to be able to test both TCP and QUIC as transport protocols, we require a library that offers some freedom regarding the transport protocol stack. Ideally, it supports TCP as well as QUIC, which is only the case with wolfSSL and picoTLS. Mbed TLS and emSSL do not have a statement about their support of QUIC, leading us to the assumption that they only support TCP to date. Picoquic, on the other hand, is a QUIC (only) implementation – as its name implies –, leaving no room for TCP aspirations.

Conclusion. These considerations make it clear that all of the libraries have their benefits and their drawbacks. EmSSL puts itself forward by its vendor, whose products have been used by our hearing instrument manufacturer before. Nonetheless, emSSL does not offer TLS 1.3 support yet, which is likely to surpass the lower TLS versions in persistence and might still be relevant by the time the next hearing aid generation is released. This constraint excludes emSSL, mbed TLS as well as picoquic, the latter not being appropriate anyway due to its rather big code footprint. Eventually, wolfSSL was favoured over picoTLS because of the availability of comprehensive documentation. However, wolfSSL comes with a license model that is less convenient for our hearing instrument manufacturer since it either requires the publication of source code or the chargeable acquisition of the license.

	wolfSSL	**mbed TLS**	**emSSL**	**picoTLS**	**picoquic**
Company	wolfSSL	-	SEGGER	-	-
Source Code Available	Yes (GitHub)	Yes (GitHub)	No	Yes (GitHub)	Yes (GitHub)
License	GNU General Public License v2.0 / Commercial License	Apache License 2.0	SEGGER Software License / Friendly License	MIT / BSD-2-Clause License	MIT
Popularity (GitHub Stars)	1.9k	4.1k	-	460	416
Supported Versions	TLS v1.0-1.3; DTLS v1.1-1.3; QUIC	TLS v1.0-1.2; DTLS v1.0-1.2	TLS v1.0-1.2	TLS v1.3	QUIC based on picoTLS which implements TLS v1.3
Code Footprint	Small size: 20-100kB	Rather small code footprint	Low footprint	OpenSSL / "minicrypto" with smaller footprint	OpenSSL on picoTLS – rather big footprint
Documentation Available	Manual, examples, tutorials	Documentation	Wiki, documentation	GitHub wiki only	Not much documentation yet
Hardware Cryptography Support	Yes	Yes	Yes	Not stated in documentation	Not stated in documentation
Lower Protocol Flexibility	TCP/QUIC	TCP only	TCP only	TCP/QUIC	QUIC only

TABLE 3.2: Comparison of TLS libaries.

3.2.2 TCP/IP Implementations

Since the decision on the security protocol stated in Chapter 3.1 favoured two alternatives featuring different underlying transport protocols, a prototype allowing for testing these two options will require the implementation of the transport protocol as well. What's more, we are interested in the overall communication overhead generated by the transport protocol as well as by the security protocol. Therefore, we conduct a rudimentary evaluation of three libraries implementing the TCP/IP stack in this chapter. Table 3.3 gives an overview over the TCP/IP libraries picoTCP [63], lwIP [64], and emNet [65]. All of the libraries evaluated here claim to be optimized for resource-constrained embedded systems, which is why all of them are expected to have a low code footprint and are not assessed on this feature in Table 3.3.

	picoTCP	**lwIP**	**emNet**
Company	Intelligent Systems / Altran	-	SEGGER
Source Code Available	Yes (GitHub)	Yes (GitHub)	No
License	GLP license for free use	BSD License	Licenses for commercial and non-commercial use
Documentation	Wiki	Wiki and Documentation	Wiki and Documentation
Lower Layer Flexibility	Yes	Yes	Yes
Compression Support	Yes	Yes	No
Up-to-date	Last commit in 2019	Last commit last week	Actively maintained by SEGGER

TABLE 3.3: Comparison of TCP/IP libaries.

While picoTCP and emNet are maintained and promoted by a company, lwIP completely relies on a network of committed developers. The source code of picoTCP and lwIP is freely accessible on GitHub and can also be used for free as the corresponding licenses are permissive. SEGGER, on the other hand, only provides emNet in the form of a pre-compiled library and on request when using it for free for non-commercial purposes. Its license model is less permissive when using emNet for commercial use. Documentation is available for all of the three libraries, with lwIP [66] and emNet [67] documentations being more extensive than picoTCP's [68].

With regard to lower layer flexibility, all of the three libraries allow to implement one's own driver, which makes them all convenient for our intention of putting an abstraction between our prototype and the real physical layer. However, compression support using 6LoWPAN is only supplied by picoTCP and lwIP, emNet does not mention 6LoWPAN in the whole of its documentation. At the same time, picoTCP does not seem to be maintained actively at the time being as the last commit to their GitHub repository dates back to 2019.

Conclusion. With these considerations in mind, we chose lwIP as our underlying transport protocol because it fulfills all of our demands regarding availability, flexibility, maintenance, and documentation. EmNet is kept as a backup option in case lwIP causes any problems.

3.3 Cipher Suite

In contrast to the security protocol and the implementation library, the evaluation of a cipher suite is not part of this thesis. Nevertheless, for the implementation of a prototype, a specific cipher suite has to be chosen. That's why some elementary properties of the available ciphers are taken into consideration here, forming the basis for a preliminary choice of a cipher suite. In a second step, we balance security against performance, reflecting on the security implications when the authentication tag of the MAC algorithm is reduced in size.

3.3.1 Choice of ciphers

As mentioned in Chapter 2.5, the ciphers provided by the target system are RSA/ECC as asymmetric crypto ciphers, AES-128 and AES-256 as symmetric crypto ciphers, and SHA-2 and SHA-3 as MAC algorithms. Since the handshake process is not part of the evaluation in this thesis, the asymmetric crypto ciphers will be excluded from our considerations, as they belong to the handshake phase.

We have defined five criteria that either contribute to the security of a connection, or have an impact on the performance of a (embedded) system, or that are required by our project-specific context. The latter is reflected in the first two criteria, TLS 1.3 and authentication tag size, while the others represent more general security concerns.

- **TLS 1.3.** As other TLS versions have been discarded, we look for a cipher suite supported by TLS 1.3. This does not rule out any of our algorithms of interest per se, but excludes some flavors of them: For instance, some AES modes of operation have been dropped by TLS 1.3, e. g. AES-CBC mode, since TLS 1.3 concentrates on AEAD algorithms. TLS 1.3 has also dropped RSA as a means for establishing a shared secret due to its security risks [69]. The five cipher suites generally supported by TLS 1.3 are (with asymmetric crypto algorithms removed from the suite) [70]:
  - TLS_AES_256_GCM_SHA384
  - TLS_CHACHA20_POLY1305_SHA256
  - TLS_AES_128_GCM_SHA256
  - TLS_AES_128_CCM_8_SHA256
  - TLS_AES_128_CCM_SHA256
- **Authentication tag size.** The authentication tag that is attached to every data packet on the record layer should be no longer than 4 bytes, which is the minimum size allowed in TLS 1.3. This was stipulated by the hearing instrument manufacturer in order to minimize the communication overhead on the record layer. The trade-off between security and overhead reduction is worked out

in Chapter 3.3.2. AEAD ciphers usually add a tag of 16 bytes, and some of them do not allow shorter tags by standard, e.g. ChaCha20-Poly1305. AES modes with AEAD do not inherently inhibit authentication tag size reduction, basically allowing sizes between 4 and 16 bytes[71].

– **Security vs. performance.** Usually, the longer the cryptographic key and the hash digest output, the less feasible a security breach and the stronger the protection of the data. At the same time, the longer the key and the hash digest, the longer it takes to compute the output, in general. What does that mean regarding the AES and SHA algorithms? AES-128 takes a 128-bit secret key, which is obviously more vulnerable to brute force attacks than AES-256 with its 256-bit key. Still, even a 128-bit key is secure against attacks by state-of-the-art technology, including quantum computing [72]. That's why it is of little use to employ AES-256 in embedded devices where computational resources are limited.
  A similar situation emerges with SHA-256/SHA-384/SHA-512 (SHA-2 algorithms of different digest sizes) and their SHA-3 counterparts. While algorithms with larger digest size provide stronger collision resistance, even SHA-256 offers enough integrity protection whilst requiring less power and memory [73]. Therefore, from a security point of view, AES-128 with SHA-256 would be sufficient for our purposes.

– **Operation mode.** As of today, GCM is the state-of-the-art block cipher mode and usually preferred over CCM because it is Encrypt-then-MAC (EtM) whereas CCM is MAC-then-Encrypt (MtE), EtM generally being considered as the more secure approach than MtE [74] (even though this does not really hold true for CCM [51]). Additionally, GCM is faster than CCM: While CCM takes 2 AES operations per block, GCM only takes one, and GCM can also be parallelized, but CCM cannot [75]. CCM, on the other hand, is implemented in hardware more easily and efficiently, as CCM only takes one AES block as opposed to two blocks for GCM. Moreover, the hash function implemented in GCM, GHASH, has known weaknesses that enables attacks faster than brute force [76]. This makes the use of GCM with short authentication tags inadvisable. CCM has no such known weaknesses, leaving brute-force attacks as the best method to crack it. However, in May 2023, an RFC draft has been released, called "Galois Counter Mode with Secure Short Tags" [77]. It suggests using AES-GCM with another hash function which would bypass the weaknesses of GHASH.

Conclusion. After having considered the points outlined above, we opted for the TLS-AES128-GCM-SHA256 cipher suite. This was mainly due to the reasons stated under "Security vs. performance" and GCM being the AES operation mode most widely deployed. In its current state, GCM has some significant flaws which does not make it suitable for use in a real-world scenario. Nevertheless, we suppose that these flaws are going to be eradicated sometime in the near future. For the considerations concerning the authentication tag reduction in the next subsection, we assume a close-to-ideal MAC algorithm with brute force being the fastest attack.

3.3.2 Implications of Authentication Tag Reduction

This sub-chapter examines the effects of a reduced authentication tag size on the integrity of a message. We make two main observations here, namely how often the key should be changed, and how the collision resistance (more precisely: the second pre-image resistance) is impaired when using an authentication tag of 4 bytes. Other influences as key size or the specific hash function in use are not considered here since they have no direct relation with the problems analyzed in this thesis (i.e. communication overhead) and, as such, can easily be replaced or adapted when needed. In addition, the key size is assumed to be the default size of AES-128, therefore this feature should not cause any deviation from official research findings in our setup.

Key Usage. For AEAD algorithms, RFC 5116 [78] determines the size of the authentication tag that ensures the integrity of the data and is appended to every packet on the record layer. This size is fixed to 16 bytes (128 bits) in the case of our chosen algorithm, AES-128 with Galois-Counter mode. This means that there is no version of TLS that supports the reduction of the authentication tag by standard and the reduction will result in an altered version of TLS. Nonetheless, NIST has issued a paper containing recommendations on the use of GCM including the optimal usage of authentication tags shorter than 16 bytes [71]. These recommendations imply, among others, that the GCM key should be re-generated frequently and that the size of the ciphertext should be as small as possible. Table 3.1 shows the maximum number of times a given key should be used for decryption of a ciphertext of a given length (including Addidional Authenticated Data (AAD)) when operating with a 32-bit authentication tag. (It has to be noted, though, that these NIST recommendations have been challenged because of missing explanations on the security level they are supposed to induce, among other criticism [77].) When assuming 10 bytes of AAD (overhead of TLS over TCP, see Chapter 3.1) and 128 bits of ciphertext, as AES-128 operates on 128-bit blocks, the combined length would result in $26 < 2^5$ bytes, thus enabling the usage of the same key for $2^{22} = 4.194.304$ times at most. What does this mean specifically for our TLS secured connection running on top of BLE? (We focus on the throughput of the connection here because a packet has to be sent before it can be decrypted and the process of transmitting takes considerably more time than the decryption itself.) We assume a BLE throughput of about 0.3 Mbps [79] and a packet size of about 30 bytes. Thus, the number n of packets sent every second is:

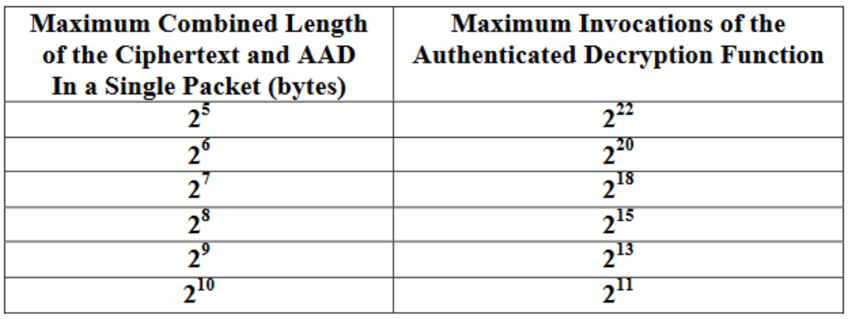
$$n \approx \frac{0.3\frac{Mb}{s}}{30 \cdot 8\ \text{bit}} = 1250\,\frac{\text{packets}}{s}$$

And the maximal usage period t of one key is:

$$t = \frac{4194304\ \text{packets}}{1250\,\frac{\text{packets}}{s}} \approx 3355\,s \approx 56\ \text{min}$$

If 1250 packets are sent every second, the key will have to be changed approximately once an hour.

Second Pre-Image Resistance. For prevention of feasible attacks on an existing BLE connection between a hearing aid and a client, we are interested in second pre-image resistance (also called weak collision resistance), which means the resistance against finding a second message with an identical digest value as a given message. For brute-force attacks, the time complexity of this task is $O(2^n)$, with n being the bit



Maximum Combined Length of the Ciphertext and AAD In a Single Packet (bytes)	Maximum Invocations of the Authenticated Decryption Function
2^5	2^{22}
2^6	2^{20}
2^7	2^{18}
2^8	2^{15}
2^9	2^{13}
2^{10}	2^{11}

FIGURE 3.1: Maximum key usage with regard to ciphertext plus AAD size when using 32-bit auth tag. [71]

length of the authentication tag [80]. The average cycle rate that it takes to process 1 byte with AES-GCM is assumed to be 100 cycles/byte. The processing of a packet of 30 bytes would then cost 3000 cycles, which means one try takes around 1 μs on a 3.0 GHz CPU. An authentication tag length of 32 bit implies that there are 2^{32} possibilities how this tag might look like. How long does it take an attacker, then, to try all of these possibilities on the CPU mentioned above?

$$\frac{1}{1 \cdot 10^{-6} s} \cdot x = 2^{32}$$

$$x \approx 4295s \approx 1.2h$$

After 1.2 hours, an attacker would have found a different message producing the same digest value with a probability of about 100%. However, a TLS connection can still be integrity-secured with a tag size of 32 bit if appropriate measures are taken. For example, one could set the connection inactivity timeout to less than one hour, such that a packet will only be received within a shorter time span than it would take to check all 2^{32} possibilities.

Chapter 4

Prototype Implementation

In this chapter, we describe the process of the implementation phase of our work. First, an overview of our vision regarding the architecture of the prototype is given, followed by two chapters on the integration of the external libraries lwIP and wolfSSL into our project and their adaptations according to our project-specific needs. Lastly, we will mention the tools that we worked with and have proven to be very useful in the debugging and monitoring process.

4.1 Architecture

Figure 4.1 illustrates the high-level architecture of the prototype and the interaction between each component.

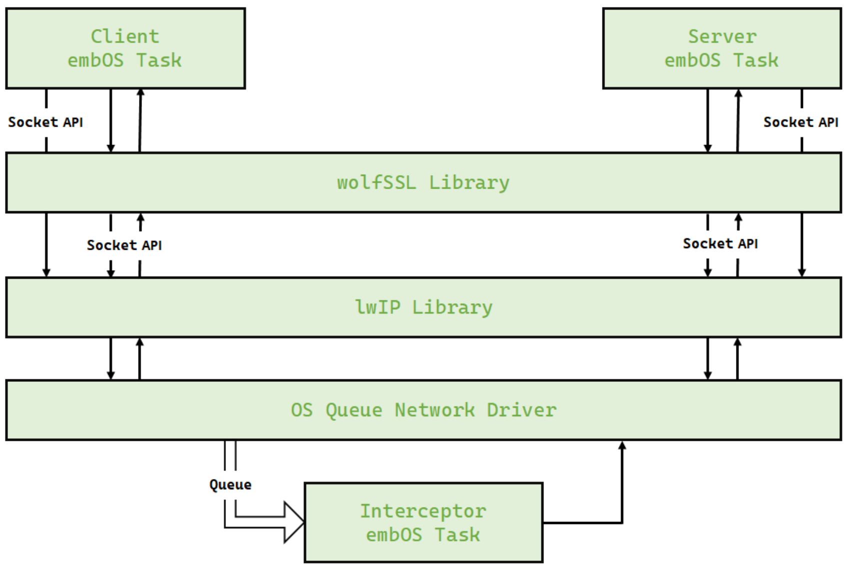


FIGURE 4.1: Target architecture of the prototype.

The prototype consists of three embOS tasks which all run on the same board. The design has a close resemblance to a virtual loopback interface with the main difference that the traffic is led over an implemented network interface which, instead of leading the packets on a real physical layer, leads it onto an embOS queue that is used to send packets between the tasks.

- **Client embOS Task:** Represents the active client that is communicating with the server task. In a real world scenario, this could for instance be a smartphone or a computer with the hearing aid calibration software that wants to communicate with the hearing aid.
- **Server embOS Task:** The server represents the passive hearing aid that waits for new arriving client connections and should handle them accordingly.
- **Interceptor embOS Task:** As the name suggests, the interceptor task intercepts the raw IP traffic that travels between the client and server task. It therefore waits in a blocked state until new packets are written to the embOS queue. Once a packet arrives in the queue and neither the server or the client need to perform further operations, the interceptor removes the packet from the queue and starts analyzing the packet. After all analyzing work is done, the interceptor calls the input function of the network interface to signalize the TCP/IP stack that a new packet has arrived which causes the packet to travel back up the network stack.

4.2 IP/TCP Implementation: lwIP

lwIP [81] is an implementation of the TCP/IP stack in C that is targeted at embedded systems. Its goal is to reduce resource usage but still providing a full implementation of the TCP/IP stack. Furthermore, an implementation of the well-known socket interface is provided, which is required to run the wolfSSL library. Besides the IP and TCP implementation, other network protocols are supported as well such as UDP, DHCP, and DNS, to name a few.

We start by describing how lwIP was integrated into our project and more specifically with our operating system embOS. After that, our implementation of the queue network driver is presented, followed by the initialization of lwIP.

4.2.1 Integration into MVP Project

The source code of lwIP is available on GitHub. The repository was cloned and the stable release tag `STABLE-2_1_3_RELEASE` was checked out and the contents of the `src` folder were copied to the new `mvp-lib-lwip` folder in the root of the project. This folder was then included in the CMake build process in the root `CMakeLists.txt` file:

```
add_subdirectory("mvp-lib-lwip")
```

Illustrated below is the folder structure of the lwIP project.

```
mvp-lib-lwip
├── src
│   ├── api
│   ├── apps
│   ├── core
│   │   ├── ipv4
│   │   └── ipv6
│   ├── include
│   └── netif
└── CMakeLists.txt
```

The `api` folder contains the raw, sequential and socket API to interact with the the library and its component. `apps` contains various service implementations like an HTTP server that are irrelevant for our context. Within the `core` folder lies the heart of the implementation as well as implementations for both the IPv4 and IPv6 protocol. `include` contains all header files of the library, and finally, the `netif` folder contains implementations for network interfaces and also functionalities for the 6LoWPAN standard discussed in Chapter 2.4.4.

All relevant C source files are referenced in the `CMakeLists.txt` file which causes them to be compiled during the build process. Not all files were included, for example all files regarding the Point-To-Point protocol were excluded, as they are not required for our prototype.

4.2.2 Integration with embOS

lwIP can operate in two modes which are controlled via the `NO_SYS` pre-processor flag [82]. Operating with the `NO_SYS` flag set to 1 disables the sequential and socket API and only the raw API may be used as the former APIs rely on OS features such as semaphores, mailboxes and threads while the latter instead works with callback functions. As our chosen TLS/QUIC implementation relies on the socket API, using lwIP in this mode is not feasible which required the integration with embOS.

Following files had to be defined or implemented for this purpose:

– **lwipopts.h** General settings file that has to be defined in any case. All settings are based on the `opt.h` header file and are also documented and can be overwritten with the `lwipopts.h` header file. In our case this is mainly used to set the `NO_SYS` flag to 0 and disable irrelevant lwIP components to reduce the code overhead.

– **sys_arch.c** All functions that lwIP uses to talk with the underlying operating system. The functions are defined in the `sys.h` header file within the *include* directory and there their purpose is documented as well.

– **sys_arch.h** Header file for the previously mentioned `sys_arch.c` file. Is included before the OS integration function definitions in the `sys.h` header file so the implementations in `sys_arch.c` are taken instead.

– **cc.h** Describes the compiler and processor to lwIP. This includes in `typedef` definitions for the signed and unsigned integer data types of 8, 16 and 32-bit size as well as printf formatters that should be used for these types. Also included are macros to forward debug output to the board specific `printf` function to allow displaying debugging output from the lwIP library which is described in Chapter 4.2.2.

Debug Output. lwIP provides an extensive printf/build-in debug functionality. This for instance prints the IPv4 or TCP headers in a readable structured format which is useful to investigate issues.

As the board provides a custom `printf` function, all debug output from lwIP had to be routed to this function. To achieve this, a new C function has been created in `ServiceHost.cpp` called `lwip_debug` which additionally adds the name of the currently executing embOS task and delegates the output to `BoardSupport::Vprintf`. Furthermore within the `cc.h` header file, the `LWIP_PLATFORM_DIAG` and `LWIP_PLATFORM_ASSERT` macro had to be overridden so that they instead call the `lwip_debug` function instead of the standard `printf` function.

Debuggers can be enabled or disabled on a component basis with their respective `XXX_DEBUG`. For instance, if only debug statements related to the socket interface should be enabled the `IP_REASS_DEBUG` flag has to be set to `LWIP_DBG_ON`.

4.2.3 Queue Network Driver

lwIP allows the creation of custom network driver to integrate the stack with your own hardware [83]. This means in our case, that the outgoing IP packet should be written on the embOS queue and there should be an input method that allows passing an incoming packet back up the stack.

lwIP requires the implementation of three functions for each network interface: `myif_init` to initialize the network interface, `myif_link_output` to transmit raw packets on to the link layer without modifying their content and `myif_output` to add link headers that may be needed in order to pass the packet on to the link layer.

Initialization Function. The initialization function receives a pointer to the `netif` struct that should be initialized. The `netif` struct is used to represent network interfaces within lwIP.

```c
err_t osqueueif_init(struct netif *netif) {
    netif->name[0] = 'q';
    netif->name[1] = 'd';

    netif->output = osqueueif_output;
    netif->linkoutput = osqueueif_link_output;
    netif->mtu = 1500;
    netif->flags |= NETIF_FLAG_LINK_UP;

    netif->hwaddr[0] = 0x11;
    netif->hwaddr[1] = 0x22;
    netif->hwaddr[2] = 0x33;
    netif->hwaddr[3] = 0x44;
    netif->hwaddr[4] = 0x55;
    netif->hwaddr[5] = 0x66;
    netif->hwaddr_len = 6;

    return ERR_OK;
}
```

lwIP requires that each network interface receives its own identifier which in our case was set to "qd" which stands for queue driver. Next the pointers to the respective output functions are set, the MTU is set and the corresponding bit that indicates the NETIF_FLAG_LINK_UP state is set. lwIP checks this flag and only sends IP packet once the flag was set to this state. Lastly, the hardware address of the interface is set. In our case, this is not a real address as we just want to imitate the presence of a real network driver.

Output Functions. The `osqueueif_output` function straight away calls the `osqueueif_link_output` function since there is no need to add link layer headers.

```c
err_t osqueueif_output(struct netif *netif, struct pbuf *p, const
    ip_addr_t *ipaddr) {
    return osqueueif_link_output(netif, p);
}
```

The `osqueueif_link_output` functions calls the `OS_QUEUE_Put` function of embOS which copies the contents of the packets payload to the queue. The `pbuf` struct is the representation of a packet within lwIP.

```c
err_t osqueueif_link_output(struct netif *netif, struct pbuf *p) {
    OS_QUEUE_Put(getQueuePtr(), p->payload, p->len);
    return ERR_OK;
}
```

Input Method. Contrary to the other functions, the input method is responsible for calling the lwIP input function that is set on the `netif` struct. The input function is set during lwIP initialization phase (see Chapter 4.2.4) and allows to notify lwIP of an incoming packet.

```c
void osqueueif_input(void* pData, int pDataLength) {
    struct pbuf* pPbuf = pbuf_alloc(PBUF_RAW, (u16_t) pDataLength,
    PBUF_RAM);
    memcpy(pPbuf->payload, pData, pDataLength);
    pPbuf->len = (u16_t) pDataLength;

    struct netif* pNetIf = getNetifPtr();
    pNetIf->input(pPbuf, pNetIf);
}
```

The `osqueueif_input` function is called by the interceptor task. The interceptor directly passes the received payload that was read from the queue to the input function as well as the payloads length. Here the data is converted back to a `pbuf` struct and together with the `netif` struct passed to the input function-pointer on the `netif` struct.

4.2.4 Initialization of lwIP

lwIP can either be initialized by using the dynamic host configuration protocol (DHCP) or by directly passing a static network definition.

```c
static struct netif queuenetif;
static ip4_addr_t ipaddr, netmask, gw;
...
IP_ADDR4(&ipaddr,  192, 168, 1, 100);
IP_ADDR4(&netmask, 255, 255, 255, 0);
IP_ADDR4(&gw,      192, 168, 1, 1);
netif_add(&queuenetif, &ipaddr, &netmask, &gw, NULL, osqueueif_init,
    ip4_input);
netif_set_up(&queuenetif);
netif_set_default(&queuenetif);
tcpip_init(NULL, NULL);
```

In our case we defined a static 192.168.1.0/24 network. The `netif_add` function lets lwIP know about the network interface. The function receives a pointer to the memory location of the `netif` struct and information about the network interface. The IP address 192.168.1.100 is assigned to the network interface. Finally, the pointer tot he init function of the network interface is passed as well as the input function that should set on the `netif` struct.

4.3 TLS/QUIC Implementation: WolfSSL

For our prototype, we chose wolfSSL [58] as a TLS implementation library (see Chapter 3.2.1). WolfSSL is written in ANSI C and comes with a bunch of features that can be easily customized using pre-processor macros. The centerpiece of the library is the integrated wolfCrypt library [84], which is responsible for the cryptographic processes like encryption and signing. WolfCrypt has been FIPS 140-2 validated by the NIST (National Institute of Standards and Technology), ensuring that the library meets the cryptographic standard issued by the U.S. government [85]. WolfSSL is at the cutting-edge of cryptographic development, not only offering support for progressive ciphers like ChaCha20 but also experimenting with post-quantum cryptography. These use cases, of course, go far beyond our thesis. To thrive in the cyber-security business, however, it is crucial for a company to have its finger on the pulse of the time, guaranteeing the adherence of its products to the most current security requirements.

In the first three subsections, we will describe the workings of the wolfSSL library, that is, how the library is structured, how a secure TLS connection is established, and how messages are sent and received. Following the more theoretical parts, we will give account of our hands-on experience when using the library: We begin by depicting how a simple client-server pair using AES-GCM as a cipher can be implemented with wolfSSL. After that, the integration of the wolfSSL library with our project-specific setup and prerequisites will be addressed, and eventually, we will describe how we approached the topic of callback functions for customizing the record layer.

4.3.1 Library Structure

Below, the tree of the most important directories inside the wolfSSL library is depicted. At first, it can be a little bit confusing due to the re-use of names. The rest of the folders either contain examples, tests, or context-specific support files (e.g. Docker).

```
wolfssl
├─src // source code files of the library
│  ├─ssl.c
│  └─...
├─wolfcrypt
│  └─src // source code files of the crypto library
│     ├─aes.c
│     └─...
└─wolfssl // header files of the library
   └─wolfcrypt // header files of the crypto library
```

4.3.2 Establishing a Connection

The wolfSSL library encapsulates the TLS elements (and processes) like the `TLS-CipherText` or even the `Handshake` struct well in its objects and functions, which is very convenient when setting up a standard TLS connection as described here. The central wolfSSL objects are the WOLFSSL_CTX struct on the one hand and the WOLFSSL struct on the other hand. The WOLFSSL_CTX forms the context of the wolfSSL connection and encapsulates information about the certificates, the keys and the cipher suite to be used, and more. Its initialization is shown below.

```
/* Create and initialize WOLFSSL_CTX */
WOLFSSL_CTX * wolfSSL_CTX_new( WOLFSSL_METHOD * )
    // e.g. WOLFSSL_METHOD = wolfTLSv1_3_client_method()
```

To feed the context object with the certificate(s) and the key(s), several functions are at disposal, depending on the format of the aforementioned components. One can either read them from a file or, what is more reasonable in our case with a prototype without a file system, from a char buffer. For example, the certificate can be read from a pem file with the function `wolfSSL_CTX_use_certificate_file()` or from a buffer with `wolfSSL_CTX_use_certificate_buffer()`. The cipher suite can be set with `wolfSSL_CTX_ set_cipher_list`.

In the meantime, the WOLFSSL object is responsible for the TLS connection itself and the communication with the opposite endpoint. It receives a pointer to the WOLFSSL_CTX and thus has access to the properties set on it before. The following code snippet shows the initialization and usage of the WOLFSSL object.

```
    /* Create a WOLFSSL object */
    WOLFSSL * wolfSSL_new( WOLFSSL_CTX * )

    /* Attach wolfSSL to a socket initialized before */
    wolfSSL_set_fd(ssl, sockfd);

    /* Client only: connect to wolfSSL on the server side */
    int wolfSSL_connect( WOLFSSL * ssl )

    /* Server only: accept a client TLS connection */
    int wolfSSL_accept( WOLFSSL * )
```

The `wolfSSL_connect()` function on the client side initiates a TLS handshake between the client and the server. The prerequisite for this function call is an already established underlying transport connection between the client and the server. When the server accepts this connection in `wolfSSL_accept()`, a secure TLS connection with the defined parameters has successfully been established and the client can start transmitting data to the server.

4.3.3 Sending and Receiving Messages

There exist several functions for sending and receiving messages in wolfSSL. The most common ones are are listed below:

```
    /*read sz bytes from the connection into the data buffer */
    int wolfSSL_read( WOLFSSL * ssl, void * data, int sz )

    /* write sz bytes from the data buffer to the connection */
    int wolfSSL_write( WOLFSSL * ssl, const void * data, int sz )
```

These functions can be used on either end of the connection. The return values of `wolfSSL_read()` and `wolfSSL_write()` state the number of bytes of data that were successfully received or sent, respectively. This means that they can also indicate a partial transmission success.

More options in the context of receiving messages are provided by the following functions:

- `wolfSSL_pending()`: Returns the amount of bytes that are waiting in the data buffer to be read.
- `wolfSSL_peek()`: This function can be used as a read function that does not modify or delete the data buffer which contains the data to be read.
- `wolfSSL_recv()`: This function does the same as `wolfSSL_read()` but enables the use of specified flags for this operation.

4.3.4 Running a Client-Server Pair with AES-GCM

The wolfSSL library offers a multitude of examples, each of which demonstrates the implementation and behaviour of a specific wolfSSL feature. Examples that were relevant for this thesis are stored in the following places:

```
wolfssl
 └─examples
 │  └─client // client example with config options
 │  └─server // server example with config options
 └─IDE
 │  └─GCC-ARM // client-server example using ECDHE-ECDSA and custom
 │  │ transport
 │  └─INTIME-RTOS // client-server example with threads
 └─wolfcrypt
    └─test // huge file with many example usages configurable
      by macros
```

A separate repository exists on GitHub that purely demonstrates use cases of the wolfssl library [86]. The tree below shows the applications that were useful for our purpose:

```
wolfssl-examples
 └─tls
 │  └─client-tls13.c // simple client example with TLS 1.3
 │  └─server-tls13.c // simple server example with TLS 1.3
 │  └─client-tls-ecdhe.c // client example using specific cipher
 │  │ suite
 │  └─server-tls-ecdhe.c // server example using specific cipher
 │    suite
 └─crypto
    └─aes
       └─aesgcm-file-encrypt.c // example on how to use AES-GCM
         encryption
```

The crucial act consists in choosing the optimal example to start from. We started by getting a simple client-server pair using AES-GCM up and running on Ubuntu, for which we made small adjustments to the wolfSSL TLS 1.3 client-server pair [87][88]. The code including our adaptations can be found in our GitHub repository. In order to enable the use of the AES-GCM cipher suite, the pre-processor macro `HAVE_AESGCM` has to be defined.

Principally, this client-server pair running on Ubuntu could be transferred as is onto our prototype after the integration described in Chapter 4.3.5 had taken place. The only major adaptation consisted in using the buffer functions instead of the file functions when loading the keys and certificates as embOS does not provide a file system.

4.3.5 Integration into MVP Project

In order to use the wolfSSL library in the MVP project, the library must be integrated into the project generally and, in a second step, made compatible with lwIP. For our project, we used wolfSSL's stable version 5.6.0.

General Integration. WolfSSL includes a pre-configured CMakeLists file, which facilitates the integration into our project (see Chapter 2.2 for the architectural conditions of our project). The CMakeLists file does not only define which subdirectories to link (depending on the preprocessor macros that are set) but also conducts checks on the system variables available and runs test and benchmark cases per default. As the wolfSSL is a stand-alone library, we tried to avoid making changes to any of its source code even for integration. Instead, we inserted definitions relevant for wolfSSL into a separate `CMakeLists.txt` file. In a first step, this concerned the configuration of several macros. To exclude the examples and the crypto tests from the build, we added the following lines:

```
    set(WOLFSSL_EXAMPLES "no")
    set(WOLFSSL_CRYPT_TESTS "no")
```

Since our target system does not allow the use of pthreads, we also defined the following macro to make wolfssl run in a single thread:

```
    set(WOLFSSL_SINGLE_THREADED "yes")
```

Furthermore, the nature of our embedded target system required the explicit disabling of shared libraries.

```
    set(BUILD_SHARED_LIBS "no")
```

Finally, we tried to exclude wolfSSL's `options.h` file that is generated automatically by running CMake with the command below. An explanation for the motivation behind this step will be given later, as well as an alternative to avoid the errors resulting from the inclusion of `options.h`, as the command below did not work.

```
    set(BUILD_DISTRO "yes")
```

Another measure that has to be taken is adding `__clang__` as a compile definition. This is necessary to align wolfSSL's configuration with the one of embOS, which pretends to have the GNU compiler run in Clang mode.

Last but not least, the location of the header files from wolfSSL and the ones to be included from embOS – wolfSSL uses `RTOS.h` – must be declared:

```
    target_include_directories(wolfssl PUBLIC
        $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/../
        mvp-lib-embos/embOS_CortexM/Inc>
        $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/Include>
)
```

Beside the additional `CMakeLists.txt` file, we also created a header file with our project specific compile definitions, under `mvp-lib-wolfssl/Include/user_settings.h`. The central part of the content of this file is echoed below.

```
#undef WOLFSSL_EMBOS
#define WOLFSSL_EMBOS

#undef NO_RC4

#undef WOLFSSL_TLS13
#define WOLFSSL_TLS13

#undef WOLFSSL_AESGCM
#define WOLFSSL_AESGCM

#undef USE_CERT_BUFFERS_1024
#define USE_CERT_BUFFERS_1024
```

The un-definition of `NO_RC4` must be made because the flag is automatically set by `options.h`, which would result in a redefinition error. As stated above, we did not manage to exclude `options.h` from the build by setting a suitable flag. And unfortunately, we also did not succeed in including the `user_settings.h` file by adding the macro `WOLFSSL_USER_SETTINGS` in the `CMakeLists.txt` file even though this is the procedure described in the wolfSSL documentation.

Then, the directory of wolfSSL can be incorporated into the project with the `add_subdirectory()` command in the root `CMakeLists.txt` file.

After all of the steps outlined above, one will realize a redefinition error arising from the `types.h` file where XMALLOC, XFREE, and XREALLOC, previously defined in `settings.h` when using the embOS macro, are defined again. From our point of view, this is not an expected behaviour, and we could only solve it by meddling with the wolfSSL source code: On line 514 in `types.h`, we now also check for the definition of `WOLFSSL_EMBOS` before entering the elif block.

Finally, to use wolfSSL in the `ServiceHost`, wolfSSL must be included in the `target_link_ libraries()` statement in the `CMakeLists.txt` file of `mvp-service-host`.

Integration with lwIP. First, some statements need to be included in order to make known that wolfSSL uses lwIP. This concerns the new `CMakeLists.txt` file for wolfSSL as well as the `user_settings.h` file:

```
/* mvp-lib-wolfssl/CMakeLists.txt */
target_include_directories(wolfssl PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/
    mvp-lib-lwip/src/include>)

/* mvp-lib-wolfssl/Include/user_settings.h */
#undef WOLFSSL_LWIP
#define WOLFSSL_LWIP
```

When attempting to integrate the wolfSSL and the lwIP libraries into the same project and to include them in the same file, we ran into several redefinition errors that are outlined below, together with the solutions that we applied to eradicate them.

- `timeval`. WolfSSL uses the time header file provided by the system which includes a definition for `timeval` whereas lwIP uses a custom `timeval` struct in `sockets.h`. As suggested in the comment on lines 515ff in `sockets.h`, we set `LWIP_TIMEVAL_PRIVATE` to 0 and included `sys/time.h` in `cc.h` to force the use of the system time header file for lwIP sockets.

- `BYTE_ORDER`. The byte order is defined by the system as well as by the lwIP library. Since both of them declare the byte order to be little endian, we added a `#ifndef BYTE_ORDER` check before the definition of `BYTE_ORDER` in `cc.h`.

- Several components from `errno.h`. This happened because we worked with the lwIP `errno.h` previous to the integration of wolfSSL, which is not a necessity. So we simply added `#ifndef EDEADLK` before the definition of `LWIP_PROVIDE_ERRNO` in `lwipopts.h` to prevent the use of lwIP's `errno.h` when the `EDEADLK` struct stemming from `errno.h` has been defined before.

After that, our project could be compiled without any errors left. However, when trying to run the client-server example as described in Chapter 4.3.4, we ran into a major error that blocked the whole system, inhibiting even the launching of the operating system. This only happened when calling the certificate/key buffer functions. The Ozone debugger eventually revealed that a Cortex-M HardFault exception was thrown. Before the code freeze of this thesis, it did not become clear what this fault was caused by. One hint may be given by the memory consumption that rockets when the concerning functions are included in the code.

4.3.6 Callbacks on Record Layer

The wolfSSL library provides a functionality to customize the handling of packets sent on the record layer. It is called "Atomic Record Processing callbacks", implementing the idea to deploy callback functions that take care of encryption/decryption and MACing/verification of the messages. Since the goal of this thesis is the reduction of the communication overhead, this functionality could be used to shorten the authentication tag size that is 16 bytes per default. Maybe further adaptations could be undertaken in these callback functions but this was not examined in this work for time reasons.

The prototype callback functions that are mentioned in the wolfSSL documentation [89] are listed below.

```c
    typedef int (*CallbackMacEncrypt)(WOLFSSL* ssl,
        unsigned char* macOut,const unsigned char* macIn,
        unsigned int macInSz,int macContent, int macVerify,
        unsigned char* encOut, const unsigned char* encIn,
        unsigned int encSz,void* ctx);

    typedef int (*CallbackDecryptVerify)(WOLFSSL* ssl,
        unsigned char* decOut, const unsigned char* decIn,
        unsigned int decSz, int content, int verify,
        unsigned int* padSz, void* ctx);
```

It took a while until we found out that these functions and also the example that is provided in `test.h` are assumably not applicable to our case because the cipher suite that we chose – AES-GCM – is Encrypt-then-MAC (EtM) while the mentioned callback function is MAC-then-Encrypt (MtE). Moreover, the only example uses

AES-CBC as a cipher, which works differently from AEAD algorithms, especially since it does not involve an authentication tag but a regular MAC (HMAC in the example). The callback functions for EtM are named `CallbackEncryptMac` and `CallbackVerifyDecrypt`, the latter taking identical parameters like `CallbackDecryptVerify`, which is why only the former – that also differs only in two arguments from its counterpart – is listed below:

```
    typedef int (*CallbackEncryptMac)
        (WOLFSSL* ssl, unsigned char* macOut, int content,
        int macVerify, unsigned char* encOut,
        const unsigned char* encIn,
        unsigned int encSz, void* ctx);
```

One can see that the difference to `CallbackMacEncrypt` consists in the missing arguments `macIn` and `macInSz`. (The definitions of all types associated with these callbacks can be found in `ssl.h`, starting from line 3149.)

Having found out which callback function to use, we faced another difficulty: The use of the EtM functions is documented nowhere. Due to this lack of documentation, it also does not clearly emerge why two different callback function types (MtE vs. EtM) exist at all, given their similarity in the function signature. Some hints regarding the purpose of the function arguments are given by the documentation of `wolfSSL_CTX_SetMacEncryptCb` that sets the MtE callback function to a context object: The `encIn` buffer contains the plaintext to be encrypted while the `encOut` buffer should hold the ciphertext after encryption. It seems that the `macOut` buffer should be used for storing the generated MAC, which would be the authentication tag with AES-GCM. We can only assume that the parameters of the EtM functions have comparable purposes.

It was then our main concern where to come by the authentication tag in the MAC-decryption function. The documentation of the `wc_AesGcmDecrypt` function that is called inside `CallbackVerifyDecrypt` states that the parameter `authTag` should already contain the authentication tag so that it can be compared with the result of the MAC operation. However, the callback function does not provide an argument that would look anything like an authentication tag. The only other option that we could conceive of was to use the `macOut` argument on the encryption-MAC side. Unfortunately, though, the `macOut` buffer is a pointer of type byte, which means it is only 8 bytes. This would not suffice for an authentication tag of default size 16 bytes. Using it in the `wc_AesGcmEncrypt` function then led to error -173 indicating the passing of a malformed argument. When not passed as a parameter, the buffer is still affected by the `wc_AesGcmEncrypt` function and is filled with numbers that also appear in the ciphertext, which would make sense because `macOut` is defined as pointing to an index close to the output message address (see `tls13.c` line 3176). We then manually verified that using it as authentication tag in the decrypt function does not result in matching authentication tags. It is therefore clear that the `macOut` buffer does not contain the authentication tag we are looking for. It has to be noted, though, that these experiments were conducted with the MtE callback functions. This may have contributed its own share to the unsatisfying results.

To be able to test the EtM callback function in the first place, some further measures need to be taken as it seems that some implementations are missing in the wolfSSL source code. In the file `tls13.c`, which, among others, is responsible for preparing the messages to be sent over TLS 1.3, only the case of MtE is handled (lines 3175–3182) with an analogous EtM case missing. There exists an implementation of EtM but only when using TLS 1.2 (`internal.c`, lines 20.968–20.976). We then tried to implement the case of EtM for TLS 1.3 by creating an `else if (ssl->ctx->EncryptMacCb)` case in the `tls13.c` file, however, this caused a segmentation fault in the context object on the server side when running the program. This is most probably an indicator of a flawed parameter passing in `tls13.c` on our part which would have to be investigated further.

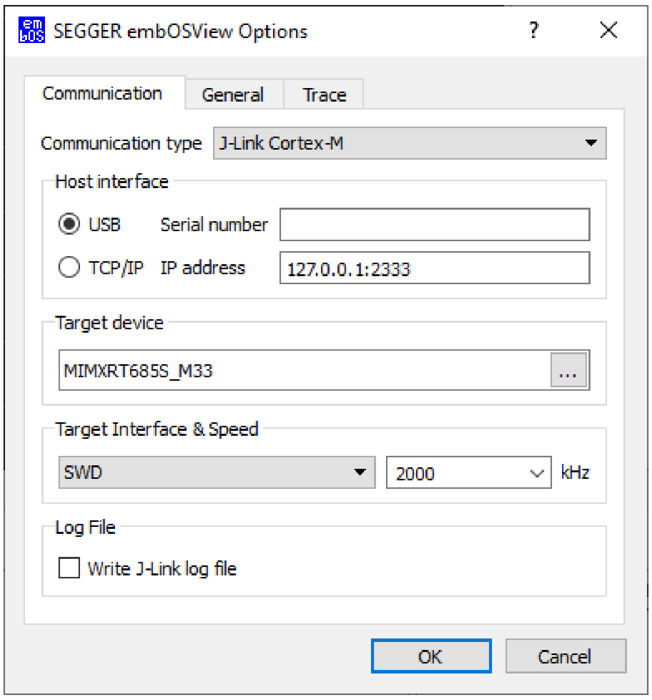
After all these struggles with gaining a more profound comprehension of wolfSSL's record layer callbacks, we had to accept that using them in our prototype would blow the time frame of our Bachelor thesis.

4.4 Debug and Monitoring Tools

4.4.1 embOSView

embOSView is a product by SEGGER, which is tightly integrated with embOS and allows insights into the currently running operating system. Especially useful is the overview of all tasks which are currently running with their assigned priority. Due to the nature of a RTOS, a task may only run if all other tasks are currently suspended which could also result into a deadlock situation if no task is able to run. embOSView provides information why a task is currently suspended which is useful when trying to diagnose a deadlock situation.

Illustrated in Figure 4.2 are the settings to be configured in order to connect to the running operating system. In our case, the connection runs via the USB port. Depending on the state of the operating system, it may not be possible to connect. This is a strong indicator that the embOS kernel did not start at all or is in a faulty state.


SEGGER embOSView Options

Communication | General | Trace

Communication type: J-Link Cortex-M

Host interface
- (●) USB — Serial number:
- ( ) TCP/IP — IP address: 127.0.0.1:2333

Target device: MIMXRT685S_M33

Target Interface & Speed: SWD — 2000 kHz

Log File
- [ ] Write J-Link log file

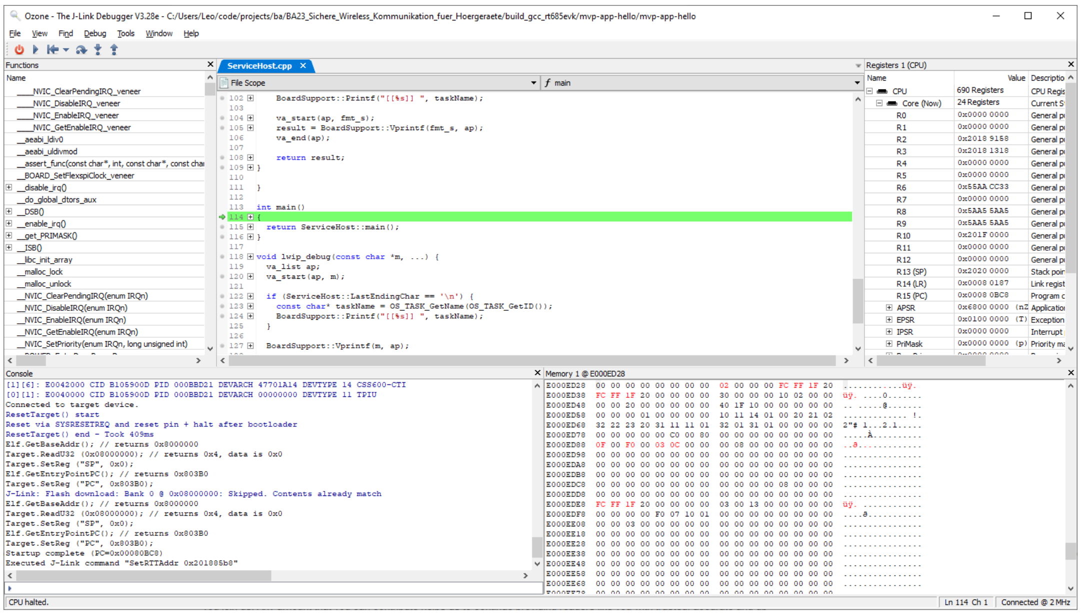
OK | Cancel

FIGURE 4.2: Settings for our prototype.

4.4.2 Ozone Debugger

The Ozone Debugger is another product by SEGGER which allows debugging programs by using the J-Link debug probes. It includes various features which are usually included in a debugger for embedded systems such as the code view to check where the program is currently running at, a memory view to glimpse at the memory's contents at all times, a disassembly view to inspect the assembly code that was created by the C or C++ instructions, viewing of all CPU registers as well as setting debug breakpoints and guards that react if a variable changes. As our NXP board runs a J-Link debug probe that is also used by the GDB J-Link debug interface, it is easily possible to debug applications using the Ozone Debugger tool.

Once the tool starts, the target device to be debugged has to be selected, which in our case is `MIMXRT685S_M33`. For the connection settings, the target interface should be set to `SWD` with a target speed of 2 MHz. The host interface should be USB. If USB is selected, the list which shows the emulators that are connected via USB should list an entry `J-Link LXPpresso V2` which can then be selected. As a next step, the program file should be selected, which in our case is in the build output folder `build_gcc_rt685evk` in the `mvp-app-hello` folder. There, the `mvp-app-hello` file has to be picked out. Optional settings should be left unchanged. Once done, the green button can be clicked to download and reset the program.


Ozone - The J-Link Debugger V3.28e - C:/Users/Leo/code/projects/ba/BA23_Sichere_Wireless_Kommunikation_fuer_Hoergeraete/build_gcc_rt685evk/mvp-app-hello/mvp-app-hello
File View Find Debug Tools Window Help
Functions
Name
ServiceHost.cpp
File Scope
main
Registers 1 (CPU)
Console
Memory 1 @ E000ED28
CPU halted.
Ln 114 Ch 1
Connected @ 2 MHz

```
___NVIC_ClearPendingIRQ_veneer
___NVIC_DisableIRQ_veneer
___NVIC_EnableIRQ_veneer
___NVIC_GetEnableIRQ_veneer
__aeabi_ldiv0
__aeabi_uldivmod
__assert_func(const char*, int, const char*, const cha
__BOARD_SetFlexspiClock_veneer
__disable_irq()
__do_global_dtors_aux
__DSB()
__enable_irq()
__get_PRIMASK()
__ISB()
__libc_init_array
__malloc_lock
__malloc_unlock
__NVIC_ClearPendingIRQ(enum IRQn)
__NVIC_DisableIRQ(enum IRQn)
__NVIC_EnableIRQ(enum IRQn)
__NVIC_GetEnableIRQ(enum IRQn)
__NVIC_SetPriority(enum IRQn, long unsigned int)
```

```
    BoardSupport::Printf("[[%s]] ", taskName);

    va_start(ap, fmt_s);
    result = BoardSupport::Vprintf(fmt_s, ap);
    va_end(ap);

    return result;
}

}

int main()
{
  return ServiceHost::main();
}

void lwip_debug(const char *m, ...) {
  va_list ap;
  va_start(ap, m);

  if (ServiceHost::LastEndingChar == '\n') {
    const char* taskName = OS_TASK_GetName(OS_TASK_GetID());
    BoardSupport::Printf("[[%s]] ", taskName);
  }

  BoardSupport::Vprintf(m, ap);
```

Name	Value	Descriptio
CPU	690 Registers	CPU Regis
Core (Now)	24 Registers	Current S
R0	0x0000 0000	General pu
R1	0x0000 0000	General pu
R2	0x2018 9158	General pu
R3	0x2018 1318	General pu
R4	0x0000 0000	General pu
R5	0x0000 0000	General pu
R6	0x55AA CC33	General pu
R7	0x0000 0000	General pu
R8	0x5AA5 5AA5	General pu
R9	0x5AA5 5AA5	General pu
R10	0x201F 0000	General pu
R11	0x0000 0000	General pu
R12	0x0000 0000	General pu
R13 (SP)	0x2020 0000	Stack poin
R14 (LR)	0x0008 0187	Link regist
R15 (PC)	0x0008 0BC8	Program c
APSR	0x6800 0000 (nZ	Applicatio
EPSR	0x0100 0000 (T)	Exception
IPSR	0x0000 0000	Interrupt
PriMask	0x0000 0000 (p)	Priority ma

```
[1][6]: E0042000 CID B105900D PID 000BBD21 DEVARCH 47701A14 DEVTYPE 14 CSS600-CTI
[0][1]: E0040000 CID B105900D PID 000BBD21 DEVARCH 00000000 DEVTYPE 11 TPIU
Connected to target device.
ResetTarget() start
Reset via SYSRESETREQ and reset pin + halt after bootloader
ResetTarget() end - Took 409ms
Elf.GetBaseAddr(); // returns 0x8000000
Target.ReadU32 (0x08000000); // returns 0x4, data is 0x0
Target.SetReg ("SP", 0x0);
Elf.GetEntryPointPC(); // returns 0x803B0
Target.SetReg ("PC", 0x803B0);
J-Link: Flash download: Bank 0 @ 0x08000000: Skipped. Contents already match
Elf.GetBaseAddr(); // returns 0x8000000
Target.ReadU32 (0x08000000); // returns 0x4, data is 0x0
Target.SetReg ("SP", 0x0);
Elf.GetEntryPointPC(); // returns 0x803B0
Target.SetReg ("PC", 0x803B0);
Startup complete (PC=0x00080BC8)
Executed J-Link command "SetRTTAddr 0x201885b8"
```

```
E000ED28  00 00 00 00 00 00 00 00  02 00 00 00 FC FF 1F 20  ............üÿ.
E000ED38  FC FF 1F 20 00 00 00 00  30 00 00 00 10 02 00 00  üÿ. ....0.......
E000ED48  00 00 20 00 00 00 00 00  40 1F 10 00 00 00 00 00  .. .....@.......
E000ED58  00 00 00 01 00 00 00 00  10 11 14 01 00 20 21 02  ............. !.
E000ED68  32 22 23 20 31 11 11 01  32 01 31 01 00 00 00 00  2"# 1...2.1.....
E000ED78  00 00 00 00 00 C0 00 80  00 00 00 00 00 00 00 00  .....À..........
E000ED88  0F 00 F0 00 03 0C 00 00  00 08 00 00 00 00 00 00  ..ð.............
E000ED98  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
E000EDA8  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
E000EDB8  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
E000EDC8  00 00 00 00 00 00 00 00  00 00 00 00 08 00 00 00  ................
E000EDD8  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
E000EDE8  FC FF 1F 20 00 00 00 00  03 00 13 00 00 00 00 00  üÿ. ............
E000EDF8  00 00 00 00 F0 07 10 01  00 00 00 00 00 00 00 00  ....ð...........
E000EE08  00 00 03 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
E000EE18  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
E000EE28  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
E000EE38  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
E000EE48  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
E000EE58  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
E000EE68  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

FIGURE 4.3: Ozone Debugger with our prototype with the programm haltet at the entry point `main` function.

Chapter 5

Results

After having gained a lot of knowledge in the theoretical part (see section 2) and plenty of experience in the practical part, we present the results of our work in this chapter. First, we will re-examine the goal of our thesis and the steps necessary to achieve it, followed by an overview of the project schedule aligning these tasks on a timeline. After the description of the targeted progress, we will present the actual outcome, i.e., what thereof has been achieved and which points are still open. Finally, we will describe the impediments we faced during the process of this thesis, which may explain why not all of the tasks mentioned in 5.1.1 could be completed within the given time frame.

5.1 Roadmap

5.1.1 Goal and Tasks

The overall goal of this thesis was to examine standardized security protocols for their utility in embedded systems, namely hearing aids. This included the following steps – some of them only crystallized out in the process and after consultation with the company representative:

- Research on available security protocols and their communication overhead.
- Agreement on one or more security protocols that seem promising for a practical evaluation.
- Research and choice of a library implementing this/these protocol(s).
- Implementation of a prototype demonstrating the behaviour of the security protocol on the communication level:
  - Implementation of the transport layer.
  - Implementation of the security protocol.
- Define metrics and measure the behaviour of the prototype, especially with regard to the communication overhead, but also to security and other parameters.
- Fine-tuning of the prototype in order to optimize the communication overhead.

– Documentation of insights and writing of the actual thesis.

5.1.2 Schedule

Figure D.1 in the appendix shows the tasks of our project on a time line. The kick-off for the project was in February, which was followed by a phase of intense research on the existing security protocols available for embedded devices. While sketchily recording the results of our research on Confluence, we would simultaneously start commissioning the MIMXRT685-EVK evaluation board that would serve as a prototype hardware. As soon as the research would yield fruit and point in a specific direction regarding the choice of the security protocol, we would begin researching appropriate implementation libraries. This should happen around half-time at the latest.

In parallel with the library research, we would lay the foundation for our practical part by implementing a transport protocol on which to run the TLS connection. This should take about two weeks. Afterwards, we could proceed with implementing the TLS protocol using wolfSSL. First, we would tackle TLS over TCP, and in a second step, TLS over QUIC, so that we could compare these two settings later on.

At the same time as the implementation takes place, we would begin writing the actual thesis and complement it as we gather more and more insights. A first draft had to be handed in three weeks before the final deadline. By that time, our gain of knowledge should ideally have come to an end such that the content of the thesis could be more or less finalized, and we would just have to fill in missing pieces and elaborate our notes taken before. The last three weeks would thus be dedicated to concluding the evaluation of the prototype implementation and completing the written thesis.

5.2 Achievements

5.2.1 Research

The results from the research phase have already been stated in Chapter 3 but will shortly be summarized here. The two most suitable (security) protocols for our purposes were TLS 1.3 over TCP and (TLS with) QUIC, both of them due to their relatively small communication overhead, and TLS over TCP possessing the additional advantage of fitting with our hearing instrument manufacturer’s present protocol stack. As a library implementing TLS, we chose wolfSSL because it seemed sufficiently documented and, what is more, was the only library in our selection that already supported TLS 1.3 as of May 2023. On the TCP layer, we opted for lwIP because of its up-to-date maintenance and documentation as well as the source code availability.

5.2.2 Prototype

Transport Protocol Layer. We implemented two tasks, a client and a server task, each of them attached to a customized driver that is able to send and receive packets over a TCP connection. A third task, called interceptor, serves as an intermediary that intercepts the raw IP traffic between the server and the client task. Each task is encapsulated in its own file and can easily be adapted or replaced. The program is functioning properly on the evaluation board. However, the transport layer for the TLS over QUIC scenario, which requires UDP, is still missing.

Security Protocol Layer. At the same time, we implemented a client-server pair on Ubuntu that establishes a TLS-secured connection and successfully transmits messages from one endpoint to the other. We integrated this setup into our prototype so that it compiles without errors. A part of the wolfSSL functions can be called on the evaluation board, but unfortunately, we did not manage to flash the whole client-server pair on the board without errors during the duration of the project and were therefore not able to assess its behaviour concerning the communication overhead and other metrics. Likewise, the fine-tuning of the TLS client-server pair with the goal of reducing the communication overhead could not be terminated. However, we discovered that omitting the protocol version on the record layer is not intended by wolfSSL and could not easily be accomplished, if possible at all.

The final project structure looks like the following (in excerpts, new files and folders highlighted in blue):

```
hd-sec-protocols
├── toolchain
├── mvp-lib-embos
├── mvp-servicehost
├── mvp-app-hello
│   └── Source
│       ├── ClientTask.cpp
│       ├── InterceptorTask.cpp
│       └── ServerTask.cpp
├── mvp-lib-lwip
├── mvp-lib-wolfssl
├── wolfssl-hosts-ubuntu
│   ├── Client.c
│   ├── Server.c
│   ├── ClientCallback.c
│   └── ServerCallback.c
└── assets
    └── putty.png
```

The client, server, and interceptor task files are included in the build and will execute when flashed onto the evaluation board. The wolfSSL functions that throw the HardFault are present but commented out.

The folders `wolfssl-hosts-ubuntu` and `assets` are not included in the project build. In the directory `wolfssl-hosts-ubuntu`, one can find the files that compile on a Ubuntu system when the corresponding wolfSSL library is built with the `configure` file, accompanied by the macro `--enable-aesgcm` (and `--enable-atomicuser`, in the case of the callback functions). The `Client.c` and `Server.c` form a ready-to-run client-server pair while the `ClientCallback.c` and `ServerCallback.c` contain the callback functions and their activation functions that are not fully working yet. The examples uploaded on GitHub contain the MAC-then-Encrypt version, as adaptations to the wolfSSL source code would have to be made in order to force the use of the callback functions when using Encrypt-then-MAC and TLS 1.3.

The `assets` folder comprises a screenshot of the putty setup that can be used to observe the execution of the flashed program on the board. The serial line may depend on the computer and can be seen under ports in the Device Manager.

As a bonus, we developed some metrics that could be used for a future assessment of a completely implemented prototype. They can be found in the Appendix B.

5.3 Impediments

In general, the research part went well and generated many insights that helped us get familiar with security protocols and the world of embedded systems. Nevertheless, it took a long time to collect a comprehensive understanding of the matter until we were able to agree on a security protocol and a library for the prototype implementation.

In the implementation phase, we had to cope with several issues. First, the setup of the TCP/IP framework was distinctly more complex and took more time than expected. This was owed to the fact that the documentation was sometimes not up-to-date which required us to understand the library with the source code. Furthermore we had to create an abstraction layer for embOS, which took more time than expected, especially due to the lack of experience with embOS. That led to a considerable delay and forced us to review our schedule, where some tasks had to be moved backwards or be omitted completely. Second, the only embedded library supporting TLS 1.3 was wolfSSL, which left little scope regarding the choice of a library. Third, the engagement in the topic of wolfSSL callback functions was cumbersome and yielded few results due to sparse documentation. This cost a considerable amount of time without a practical benefit for our prototype. Moreover, the integration of wolfSSL into our project worked out relatively smoothly at the beginning but resulted in a major fault on the prototype evaluation board when trying to use certificates. This could not be solved within the given time frame anymore. And finally, being unfamiliar with CMake posed some problems when integrating the external libraries into the project, especially at the beginning. However, we climbed up the learning curve relatively fast and eventually managed to integrate new libraries without external help.

Having stated the missing points in our prototype as well as the hindrances that were met in the process, it might be interesting to learn how our prototype solution can still contribute to the examination of the performance of standardized security protocols. This will be explained in the last chapter, together with some recommendations for sequel projects.

Chapter 6

Discussion and Outlook

The goal of this work was to examine the suitability of standardized security protocols for the use in hearing aids. On a theoretical level, we succeeded in identifying promising candidates, namely TLS over TCP and TLS over QUIC. On a practical level, though, some steps still have to be taken until a final judgement can be delivered. Our prototype could be used as a starting point in the future process of gaining a deeper understanding of the behaviour of the mentioned security protocols in an embedded setting. It already supplies a robust foundation for running a client-server pair with a secure connection on top by implementing the TCP protocol and providing custom drivers. Furthermore, the interceptor task at hand facilitates the analysis of packets on the TCP level. The modular setup allows for a quick exchange of components and even libraries.

However, we would not recommend working with wolfSSL as a TLS implementation library any longer. First, it is not clear if the error arising from the use of certificates can be solved in a reasonable time span as the evidence so far suggests that calling the buffer functions demands a considerable part of the memory space. If wolfSSL stayed in use, it would presumably have to be optimized vigorously according to the requirements of an embedded system, e.g. by reducing the stack size and excluding all unused files from the build, which should both be achievable by defining the corresponding preprocessor macros. Second, the documentation of wolfSSL is rather poor when it comes to cases stretching beyond the basic client-server setup, especially for callback functions on the record layer. Third, as far as we have seen, there is no option to omit the protocol version from the record layer packet. It would be the subject of further investigations if other libraries offer such an option, which may well not be the case as this is not a feature of standard TLS. Fourth, the wolfSSL commercial license poses some restrictions concerning the use of open source code that can be circumvented by using a library with a more permissive license. It would therefore be advisable to wait until emSSL offers TLS 1.3 support, since emSSL also naturally integrates with embOS.

When approaching the technology stack provided by SEGGER, it would sound reasonable to also implement emNet on the transport layer instead of lwIP. However, lwIP offers some non-negligible advantages over emNet despite its alienness to the SEGGER products. It has more features than emNet, especially worth mentioning here is the compression method 6loWPAN that is already integrated. Concerning the lwIP integration, the only thing we would retrospectively do differently is the usage of a git submodule rather than directly copying the code into the project. This would facilitate tracking the dependency and updating to newer versions. In this work, though, this would only have been possible if we had possessed the knowledge about CMake from the beginning. The same could have been done with wolfSSL, however, the library would first have to be adjusted to accept the user-defined settings file.

An alternative to our current setup would be the usage of a physical connection between the evaluation board and a computer that runs over a UART interface instead of faking two endpoints running on the same device. This would facilitate the simulation of the real use case but may introduce other issues like how to decompress the 6loWPAN packet on the host machine if the corresponding mechanism is used on the board.

Reconsidering the impediments that we encountered in the course of our thesis, we would recommend heralding the start of the practical part earlier, as there are often more unforeseen twists to be expected from the implementation phase than from the research phase, like in our case. It would also have been beneficial to make use of the Ozone debugger right from the beginning of the implementation process – this would have been a valuable support when debugging the TCP/IP stack.

What this all amounts to is that in a future work, our prototype could be used as a foundational setup providing a client-server-interceptor structure operating on the TCP level that then could be complemented with a library implementing the TLS protocol that enables secure communication between the client and the server task. In a second step, the same setup could be built for TLS over QUIC, and in a third step, the two setups could be compared regarding their communication overhead and further metrics, finally establishing whether standardized security protocols are suitable to secure the hearing aids of the next generation.

Bibliography

[1] L. Näf. "Formal verification of security protocols for wireless communication with hearing aids". In: (2021).

[2] NXP. *MIMXRT685-EVK*. URL: https://www.nxp.com/part/MIMXRT685-EVK#/. (accessed at: 24/04/2023).

[3] SEGGER. *embOS Real-Time Operating System User Guide & Reference Manual*. URL: https://www.segger.com/doc/UM01001_embOS.html. (accessed at: 19/05/2023).

[4] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. URL: https://www.rfc-editor.org/rfc/rfc8446. (accessed at: 22/04/2023).

[5] G. Restuccia; H. Tschofenig; E. Baccelli. "Low-Power IoT Communication Security: On the Performance of DTLS and TLS 1.3". In: *Proc. of 9th IFIP/IEEE PEMWN* (2020).

[6] A10 Networks. *Key differences Between TLS 1.2 and TLS 1.3*. URL: https://www.a10networks.com/glossary/key-differences-between-tls-1-2-and-tls-1-3/. (accessed at: 22/04/2023).

[7] H. Hooper. *CCNP Security VPN 642-648 Official Cert Guide*. Cisco Press, 2012.

[8] E. Rescorla; H. Tschofenig; N. Modadugu. *The Datagram Transport Layer Security (DTLS) Protocol Version 1.3*. URL: https://www.rfc-editor.org/rfc/rfc9147. (accessed at: 29/05/2023).

[9] E. Rescorla; N. Modadugu. *Datagram Transport Layer Security Version 1.2*. URL: https://www.rfc-editor.org/rfc/rfc6347. (accessed at: 09/06/2023).

[10] S. Kent; K. Seo. *Security Architecture for the Internet Protocol*. URL: https://www.rfc-editor.org/rfc/rfc4301. (accessed at: 07/05/2023).

[11] C. Kozierok. *IPSec Architectures and Implementation Methods*. URL: http://www.tcpipguide.com/free/t_IPSecArchitecturesandImplementationMethods-2.htm. (accessed at: 04/06/2023).

[12] C. Kaufman; P. Hoffman; Y. Nir; P. Eronen; T. Kivinen. *Internet Key Exchange Protocol Version 2 (IKEv2)*. URL: https://www.rfc-editor.org/rfc/rfc7296. (accessed at: 04/06/2023).

[13] J. Schwenk. *Sicherheit und Kryptographie im Internet: Theorie und Praxis*. ger. 5. Aufl. 2020. Wiesbaden: Springer Fachmedien Wiesbaden. ISBN: 9783658292591.

[14] A. Badach. *Technik der IP-Netze : : Grundlagen der IPv4- und IPv6-Kommunikation*. ger. 5th ed. München: Hanser, 2022. ISBN: 9783446474260.

[15] S. Kent. *IP Encapsulating Security Payload (ESP)*. URL: https://www.rfc-editor.org/rfc/rfc4303. (accessed at: 06/06/2023).

[16] J. Iyengar; M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. URL: https://www.rfc-editor.org/rfc/rfc9000. (accessed at: 04/06/2023).

[17] D. Stenberg. *Why QUIC*. URL: https://http3-explained.haxx.se/en/why-quic. (accessed at: 04/06/2023).

[18] M. Thomson. *Version-Independent Properties of QUIC*. URL: https://www.rfc-editor.org/rfc/rfc8999. (accessed at: 04/06/2023).

[19] J. Schaad. *CBOR Object Signing and Encryption (COSE): Structures and Process*. URL: https://www.rfc-editor.org/rfc/rfc9052. (accessed at: 24/04/2023).

[20] J. Schaad. *CBOR Object Signing and Encryption (COSE): Initial Algorithms*. URL: https://www.rfc-editor.org/rfc/rfc9053. (accessed at: 24/04/2023).

[21] J. Schaad. *CBOR Object Signing and Encryption (COSE)*. URL: https://www.rfc-editor.org/rfc/rfc8152. (accessed at: 24/04/2023).

[22] B. Mitchell. *What Is Wi-Fi Protected Access (WPA)?* URL: https://www.lifewire.com/definition-of-wifi-protected-access-816576. (accessed at: 01/05/2023).

[23] A. Irei. *Wireless security: WEP, WPA, WPA2 and WPA3 differences*. URL: https://www.techtarget.com/searchnetworking/feature/Wireless-encryption-basics-Understanding-WEP-WPA-and-WPA2. (accessed at: 01/05/2023).

[24] S. Deering; R. Hinden. *RFC 8200: Internet Protocol, Version 6 (IPv6) Specification*. URL: https://www.rfc-editor.org/rfc/rfc8200. (accessed at: 01/05/2023).

[25] W. Eddy (Ed.) *RFC 9293 Transmission Control Protocol (TCP)*. URL: https://www.ietf.org/rfc/rfc9293.html. (accessed at: 20/05/2023).

[26] G. Noviantika. *TCP Protocol: Understanding What Transmission Control Protocol Is and How It Works*. URL: https://www.hostinger.com/tutorials/tcp-protocol. (accessed at: 20/05/2023).

[27] C. Bormann (Ed.) *RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed*. URL: https://www.rfc-editor.org/rfc/rfc3095. (accessed at: 03/06/2023).

[28] R. Herrero. *Fundamentals of IoT Communication Technologies*. eng. Textbooks in Telecommunication Engineering. Cham: Springer International Publishing AG, 2021. ISBN: 9783030700799.

[29] LAN/MAN Standards Committee. *IEEE Standard for Low-Rate Wireless Networks*. URL: https://standards.ieee.org/ieee/802.15.4/7029. (accessed at: 21/05/2023).

[30] G. Montenegro; N. Kushalnagar; J. Hui; D. Culler. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. URL: https://www.rfc-editor.org/rfc/rfc4944. (accessed at: 21/05/2023).

[31] J. Hui (Ed.) *Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks*. URL: https://www.rfc-editor.org/rfc/rfc6282. (accessed at: 22/05/2023).

[32] S. Raza; T. Voigt; U. Roedig. *6LoWPAN Extension for IPsec. Proceedings of the IETF-IAB International Workshop on Interconnecting Smart Objects with the Internet*. 2011.

[33] S. Raza; S. Duquennoy; G. Selander. *Compression of IPsec AH and ESP Headers for Constrained Environments*. URL: https://datatracker.ietf.org/doc/html/draft-raza-6lowpan-ipsec-01. (accessed at: 08/06/2023).

[34] V. Jacobson. *Compressing TCP/IP Headers for Low-Speed Serial Links*. URL: https://www.rfc-editor.org/rfc/rfc1144#page-7. (accessed at: 20/05/2023).

[35] J. Ishac. “Survey of Header Compression Techniques”. In: *NASA/TM-2001-211154* (2001).

[36] J. Nieminen; T. Savolainen; M. Isomaki; B. Patil; Z. Shelby; C. Gomez. *IPv6 over BLUETOOTH(R) Low Energy*. URL: https://www.rfc-editor.org/rfc/rfc7668. (accessed at: 28/05/2023).

[37] Microsoft. *Cipher Suites in TLS/SSL (Schannel SSP)*. URL: https://learn.microsoft.com/en-au/windows/win32/secauthn/cipher-suites-in-schannel. (accessed at: 06/05/2023).

[38] Sectigo. *What Are the Differences Between RSA, DSA, and ECC Encryption Algorithms?* URL: https://sectigo.com/resource-library/rsa-vs-dsa-vs-ecc-encryption. (accessed at: 08/05/2023).

[39] A. Froehlich. *Elliptical Curve Cryptography (ECC)*. URL: https://www.techtarget.com/searchsecurity/definition/elliptical-curve-cryptography. (accessed at: 08/05/2023).

[40] D. Goodin. *RSA's demise from quantum attacks is very much exaggerated, expert says*. URL: https://arstechnica.com/information-technology/2023/01/fear-not-rsa-encryption-wont-fall-to-quantum-computing-anytime-soon/. (accessed at: 08/06/2023).

[41] Microsoft Azure Quantum Team. *The quantum computing effect on public-key encryption*. URL: https://cloudblogs.microsoft.com/quantum/2018/05/02/the-quantum-computing-effect-on-public-key-encryption/. (accessed at: 08/06/2023).

[42] B. Jena. *What Is AES Encryption and How Does It Work?* URL: https://www.simplilearn.com/tutorials/cryptography-tutorial/aes-encryption. (accessed at: 08/05/2023).

[43] M. Dworkin; E. Barker; J. Nechvatal; J. Foti; L. Bassham; J. Dray. *Advanced Encryption Standard (AES)*. 2001. DOI: https://doi.org/10.6028/NIST.FIPS.197.

[44] A. Dames; E. Richuso. *What Is Quantum-Safe Cryptography, and Why Do We Need It?* URL: https://www.ibm.com/cloud/blog/what-is-quantum-safe-cryptography-and-why-do-we-need-it. (accessed at: 05/06/2023).

[45] randomsapien. *Advanced Encryption Standard (AES)*. URL: https://www.geeksforgeeks.org/advanced-encryption-standard-aes/. (accessed at: 08/05/2023).

[46] J. Katz; Y. Lindell. *Introduction to Modern Cryptography*. CRC Press, 2020. ISBN: 978-1-351-13301-2.

[47] M. Bodewes. *What is the advantage of AEAD ciphers?* URL: https://crypto.stackexchange.com/questions/27243/what-is-the-advantage-of-aead-ciphers. (accessed at: 08/05/2023).

[48] F. Valsorda. *TLS nonce-nse*. URL: https://blog.cloudflare.com/tls-nonce-nse/. (accessed at: 23/05/2023).

[49] J. Salowey; A. Choudhury; D. McGrew. *AES Galois Counter Mode (GCM) Cipher Suites for TLS*. URL: https://www.rfc-editor.org/rfc/rfc5288. (accessed at: 23/05/2023).

[50] D. McGrew; D. Bailey. *AES-CCM Cipher Suites for Transport Layer Security (TLS)*. URL: https://www.rfc-editor.org/rfc/rfc6655. (accessed at: 23/05/2023).

[51] Soatok. *Comparison of Symmetric Encryption Methods*. URL: https://soatok.blog/2020/07/12/comparison-of-symmetric-encryption-methods/#aes-gcm-vs-aes-ccm. (accessed at: 15/05/2023).

[52] *Authenticated Encryption: CCM and GCM*. URL: https://www.brainkart.com/article/Authenticated-Encryption--CCM-and-GCM_8459/. (accessed at: 23/05/2023).

[53] T. Leek. *What's the hash for in ECDHE-RSA-AES-GCM-SHA?* URL: https://security.stackexchange.com/questions/39590/whats-the-hash-for-in-ecdhe-rsa-aes-gcm-sha. (accessed at: 08/05/2023).

[54] N. Landman; C. Williams; E. Ross. *Secure Hash Algorithms*. URL: https://brilliant.org/wiki/secure-hashing-algorithms/. (accessed at: 08/05/2023).

[55] J. Lake. *What is SHA-2 and how does it work?* URL: https://www.comparitech.com/blog/information-security/what-is-sha-2-how-does-it-work/. (accessed at: 16/05/2023).

[56] J. Lake. *What is a collision attack?* URL: https://www.comparitech.com/blog/information-security/what-is-a-collision-attack/. (accessed at: 16/05/2023).

[57] SEGGER. *emSSL — Transport Layer Security*. URL: https://www.segger.com/products/security-iot/emssl/. (accessed at: 08/06/2023).

[58] wolfSSL. *wolfSSL Embedded SSL/TLS Library*. URL: https://www.wolfssl.com/products/wolfssl/. (accessed at: 16/05/2023).

[59] ARM Mbed. *TLS*. URL: https://os.mbed.com/docs/mbed-os/v6.16/apis/tls.html. (accessed at: 08/06/2023).

[60] H2O. *picoTLS*. URL: https://github.com/h2o/picotls. (accessed at: 08/06/2023).

[61] Private Octopus. *picoquic*. URL: https://github.com/private-octopus/picoquic. (accessed at: 08/06/2023).

[62] mbedTLS. *Roadmap*. URL: https://mbed-tls.readthedocs.io/en/latest/project/roadmap. (accessed at: 09/06/2023).

[63] Altran. *PicoTCP*. URL: https://picotcp.altran.be/. (accessed at: 27/05/2023).

[64] A. Dunkels. *lwIP*. URL: https://www.nongnu.org/lwip/2_1_x/index.html. (accessed at: 27/05/2023).

[65] SEGGER. *emNet - The TCP/IP stack for embedded devices*. URL: https://www.segger.com/products/connectivity/emnet/. (accessed at: 27/05/2023).

[66] *lwIP - lightweight TCP/IP*. URL: http://lwip.wikia.com/wiki/LwIP_Wiki. (accessed at: 27/05/2023).

[67] SEGGER. *emNet User Guide & Reference Manual*. URL: https://www.segger.com/doc/UM07001_emNet.html. (accessed at: 27/05/2023).

[68] Altran EESY Belgium. *picoTCP on GitHub*. URL: https://github.com/tass-belgium/picotcp/wiki. (accessed at: 27/05/2023).

[69] P. Nohe. *TLS 1.3: A Complete Overview*. URL: https://www.thesslstore.com/blog/tls-1-3-everything-possibly-needed-know/. (accessed at: 17/05/2023).

[70] B. Kiprin. *What are TLS/SSL Cipher Suites and how to order them*. URL: https://crashtest-security.com/configure-ssl-cipher-order/#tls-cipher-suites. (accessed at: 18/05/2023).

[71] M. Dworkin. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. 2007. DOI: https://doi.org/10.6028/NIST.SP.800-38D.

[72] E. Tobias. *128 or 256 bit Encryption: Which Should I Use?* URL: https://www.ubiqsecurity.com/128bit-or-256bit-encryption-which-to-use/. (accessed at: 18/05/2023).

[73] Xander. *Why would I choose SHA-256 over SHA-512 for a SSL/TLS certificate?* URL: https://security.stackexchange.com/questions/165559/why-would-i-choose-sha-256-over-sha-512-for-a-ssl-tls-certificate. (accessed at: 18/05/2023).

[74] M. Bellare; C. Namprempre. "Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm". In: *ASIACRYPT 2000* (2002), pp. 531–545.

[75] Laplacian. *How to choose between AES-CCM and AES-GCM for storage volume encryption*. URL: https://crypto.stackexchange.com/questions/6842/how-to-choose-between-aes-ccm-and-aes-gcm-for-storage-volume-encryption. (accessed at: 15/05/2023).

[76] N. Ferguson. "Authentication weaknesses in GCM". In: (2005).

[77] M. Campagna; A. Maximov; J. Preuß Mattsson. *Galois Counter Mode with Secure Short Tags (GCM-SST)*. URL: https://www.ietf.org/archive/id/draft-mattsson-cfrg-aes-gcm-sst-00.html. (accessed at: 23/05/2023).

[78] D. McGrew. *An Interface and Algorithms for Authenticated Encryption*. URL: https://www.rfc-editor.org/rfc/rfc5116. (accessed at: 22/05/2023).

[79] C. Coleman. *A Practical Guide to BLE Throughput*. URL: https://interrupt.memfault.com/blog/ble-throughput-primer. (accessed at: 22/05/2023).

[80] M. Rosulek. *11.1: Security Properties for Hash Functions*. URL: https://eng.libretexts.org/Under_Construction/Book%5C%3A_The_Joy_of_Cryptography_(Rosulek)/12%5C%3A_Hash_Functions/12.01%5C%3A_Security_Properties_for_Hash_Functions. (accessed at: 22/05/2023).

[81] *lwIP GitHub repository*. URL: https://github.com/lwip-tcpip/lwip. (accessed at: 01/06/2023).

[82] *LwIP with or without an operating system*. URL: https://lwip.fandom.com/wiki/LwIP_with_or_without_an_operating_system. (accessed at: 01/06/2023).

[83] *Writing a device driver*. URL: https://lwip.fandom.com/wiki/Writing_a_device_driver. (accessed at: 01/06/2023).

[84] wolfSSL. *wolfCrypt Embedded Crypto Engine*. URL: https://www.wolfssl.com/products/wolfcrypt-2/. (accessed at: 16/05/2023).

[85] Microsoft. *FIPS 140-2 Validation*. URL: https://learn.microsoft.com/en-us/windows/security/threat-protection/fips-140-validation. (accessed at: 16/05/2023).

[86] wolfSSL. *WolfSSL Examples*. URL: https://github.com/wolfSSL/wolfssl-examples. (accessed at: 16/05/2023).

[87] wolfSSL. *client-tls13.c*. URL: https://github.com/wolfSSL/wolfssl-examples/blob/master/tls/client-tls13.c. (accessed at: 15/05/2023).

[88] wolfSSL. *server-tls13.c*. URL: https://github.com/wolfSSL/wolfssl-examples/blob/master/tls/server-tls13.c. (accessed at: 15/05/2023).

[89] wolfSSL. *6. Callbacks*. URL: https://www.wolfssl.com/documentation/manuals/wolfssl/chapter06.html#user-atomic-record-layer-processing. (accessed at: 02/06/2023).

List of Figures

Appendix A

Code Listings

LISTING A.1: ServiceHost.cpp

```
#include "BoardSupport.hpp"
#include "RTOS.h"

#include "ServiceHost.hpp"
#include "ServiceHost.h"

#include "netif/osqueuedriver.h"
#include "lwip/netif.h"
#include "lwip/ip4_addr.h"
#include "lwip/tcpip.h"
#include "lwip/ip4.h"
#include "lwip/def.h"

#include <stdlib.h>
#include <cstdarg>

extern "C" void OS_DeInitHW(void);

namespace ServiceHost
{

// buffer and context for embOS
#define BUFFER_SIZE    (256u)
static OS_U8           Buffer[BUFFER_SIZE]; // buffer for main stack copy
static OS_MAIN_CONTEXT MainContext;

// task control block and task stacks
static OS_STACKPTR int ServerTcbStack[1024];
static OS_TASK         ServerTcb;
static OS_STACKPTR int ClientTcbStack[1024];
static OS_TASK         ClientTcb;
static OS_STACKPTR int InterceptorTcbStack[1024];
static OS_TASK         InterceptorTcb;

// os queue used by network driver
#define MESSAGE_ALIGNMENT    (4u)
#define MESSAGE_SIZE_PAYLOAD (500u + OS_Q_SIZEOF_HEADER +
    MESSAGE_ALIGNMENT)
#define QUEUE_SIZE           (MESSAGE_SIZE_PAYLOAD)
static OS_QUEUE              DataQueue;
static char                  DataQueueBuffer[QUEUE_SIZE];

// lwip network driver
static struct netif queuenetif;
static ip4_addr_t   ipaddr, netmask, gw;
```

```

// used to save the ending char of the lwip debug message
static char LastEndingChar;

int main(void)
{
    BoardSupport::Init();
    BoardSupport::Printf("\r\n");
    BoardSupport::Printf("\r\n");
    BoardSupport::Printf("--- ServiceHost built for " BUILD_TARGET " with compiler " BUILD_COMPILER " initialized.\r\n");

    OS_Init();
    OS_InitHW();

    // setup lwIP
    IP_ADDR4(&ipaddr,  192, 168, 1, 100);
    IP_ADDR4(&netmask, 255, 255, 255, 0);
    IP_ADDR4(&gw,      192, 168, 1, 1);
    // currently using ip4_input which appears to restrict input to only ipv4 packets
    netif_add(&queuenetif, &ipaddr, &netmask, &gw, NULL, osqueueif_init, ip4_input);
    netif_set_up(&queuenetif);
    netif_set_default(&queuenetif);
    tcpip_init(NULL, NULL);

    // create queue
    OS_QUEUE_Create(&DataQueue, &DataQueueBuffer, sizeof(DataQueueBuffer));

    // create application tasks
    BoardSupport::Printf("Create Interceptor Task Control Block.\r\n");
    OS_TASK_CREATE(&InterceptorTcb, "Interceptor Task", 1, InterceptorTask, InterceptorTcbStack);

    BoardSupport::Printf("Create Client Task Control Block.\r\n");
    OS_TASK_CREATE(&ClientTcb, "Client Task", 2, ClientTask, ClientTcbStack);

    BoardSupport::Printf("Create Server Task Control Block.\r\n");
    OS_TASK_CREATE(&ServerTcb, "Server Task", 3, ServerTask, ServerTcbStack);

    // start embOS
    BoardSupport::Printf("Start embOS.\r\n");
    OS_ConfigStop(&MainContext, Buffer, BUFFER_SIZE);
    OS_Start();

    // terminate OS (only reached if no OS task is running anymore or OS_Stop() is called)
    OS_DeInitHW();
    OS_DeInit();
    BoardSupport::Printf("OS Terminated.\r\n");
    BoardSupport::Terminate();

    return 0;
}

int Printf(const char *fmt_s, ...)
{
    va_list ap;
    int result = 0;

    const char* taskName = OS_TASK_GetName(OS_TASK_GetID());
    BoardSupport::Printf("[[%s]] ", taskName);

    va_start(ap, fmt_s);
    result = BoardSupport::Vprintf(fmt_s, ap);
    va_end(ap);

    return result;
}

}

int main()
{
  return ServiceHost::main();
}

void lwip_debug(const char *m, ...) {
  va_list ap;
  va_start(ap, m);

  if (ServiceHost::LastEndingChar == '\n') {
    const char* taskName = OS_TASK_GetName(OS_TASK_GetID());
    BoardSupport::Printf("[[%s]] ", taskName);
  }

  BoardSupport::Vprintf(m, ap);

  va_end(ap);
}

OS_QUEUE* getQueuePtr() {
  return &ServiceHost::DataQueue;
}

struct netif* getNetifPtr() {
  return &ServiceHost::queuenetif;
}
```

LISTING A.2: ClientTask.cpp

```
#include "RTOS.h"
#include "ServiceHost.hpp"
#include "BoardSupport.hpp"
#include "lwip/sockets.h"

void ClientTask(void) {

    OS_TASK_Delay(1000);
    ServiceHost::Printf("Start client task.\r\n");

    char readBuffer[20] = { 0 };
    char* clientMsg = "client hello";

    struct sockaddr_in serv_addr;
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(8080);

    if (inet_pton(AF_INET, "192.168.1.100", &serv_addr.sin_addr) <= 0) {
        ServiceHost::Printf("ERROR: Invalid address. Address not supported
\r\n");
        abort();
    }

    while (1) {
        BoardSupport::Printf("\r\n");
        BoardSupport::Printf("\r\n");
        BoardSupport::Printf("\r\n");

        int client_fd;
        if ((client_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
            ServiceHost::Printf("ERROR: Socket creation error.\r\n");
            abort();
        }

        int connection_status;
        ServiceHost::Printf("Trying to connect to server.\r\n");
        if ((connection_status  = connect(client_fd, (struct sockaddr*) &
    serv_addr, sizeof(serv_addr))) < 0) {
            ServiceHost::Printf("ERROR: Connection failed.\r\n");
            abort();
        }
        ServiceHost::Printf("Connection to server successful!\r\n");

        send(client_fd, clientMsg, strlen(clientMsg), 0);
        ServiceHost::Printf("Client has sent message to server.\r\n");

        int read_status = read(client_fd, readBuffer, 20);
        ServiceHost::Printf("Received message from server: %s\r\n",
    readBuffer);

        ServiceHost::Printf("Closing client socket.\r\n");
        close(client_fd);

        OS_TASK_Delay(5000);
    }

}
```

LISTING A.3: InterceptorTask.cpp

```
#include "RTOS.h"
#include "ServiceHost.hpp"
#include "ServiceHost.h"
#include "netif/osqueuedriver.h"

void InterceptorTask(void) {

    ServiceHost::Printf("Start interceptor task.\r\n");

    while (1) {
        void* pData;
        int pDataLength = OS_QUEUE_GetPtrBlocked(getQueuePtr(), &pData);
        if (pDataLength == 0) {
            ServiceHost::Printf("ERROR: Message length is zero.\r\n");
            abort();
        }

        osqueueif_input(pData, pDataLength);
        OS_QUEUE_Purge(getQueuePtr());
    }

}
```

LISTING A.4: ServerTask.cpp

```cpp
#include "RTOS.h"
#include "ServiceHost.hpp"
#include "lwip/sockets.h"

void ServerTask(void) {

    ServiceHost::Printf("Start server task.\r\n");

    char readBuffer[20] = { 0 };
    char* serverMsg = "server hello";

    int server_fd;
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        ServiceHost::Printf("ERROR: Socket creation error.\r\n");
        abort();
    }

    struct sockaddr_in address;
    int addrlen = sizeof(address);
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(8080);

    if (bind(server_fd, (struct sockaddr*) &address, sizeof(address)) < 0) {
        ServiceHost::Printf("ERROR: Binding socket failed.\r\n");
        abort();
    }

    if (listen(server_fd, 3) < 0) {
        ServiceHost::Printf("ERROR: Failed to create listening socket.\r\n");
        abort();
    }

    while (1) {

        int new_socket;
        ServiceHost::Printf("Waiting for new connections.\r\n");
        if ((new_socket = accept(server_fd, (struct sockaddr*)&address, (socklen_t*)&addrlen)) < 0) {
            ServiceHost::Printf("ERROR: Failed to accept new connection.\r\n");
            abort();
        }
        ServiceHost::Printf("New client connected!\r\n");

        read(new_socket, readBuffer, 20);
        ServiceHost::Printf("Received message from client: %s\r\n", readBuffer);

        send(new_socket, serverMsg, strlen(serverMsg), 0);
        ServiceHost::Printf("Server has sent response message to client.\r\n");

        ServiceHost::Printf("Closing client socket.\r\n");
        close(new_socket);

    }

    shutdown(server_fd, SHUT_RDWR);

}
```

Appendix B

Metrics for Assessment

Characteristic	**Metric**
Performance	Communication overhead: • throughput (data/unit time) [bytes] • latency (time to send packet) [s]
Performance	Computational overhead: • CPU utilization (%) • latency (time to encrypt/decrypt) [s]
Storage	memory usage [bytes]
Security	• number of adaptations made to protocol • number of deviations from RFC standard
Security	confidentiality: • asymmetric crypto: – private key size [bytes] – security of cipher according to literature • symmetric crypto: – perfect forward secrecy (yes/no) – shared secret key size [bytes] – security of cipher and operation mode according to literature
Security	integrity: • MAC digest size [bytes] • secret key size used for MAC [bytes]
Security	authenticity: signature, key size [bytes]?

TABLE B.1: Security of prototype implementation.

Appendix C

Project Description

C.1 Übersicht

Moderne Hörgeräte werden immer vernetzter. Neben der kabellosen Kommunikation zwischen rechtem und linkem Hörgerät erfolgt zusätzlich eine Bluetooth-Kom -munikation mit Geräten wie Smartphones und PCs. Über diese Geräte können Hörgeräte zum Beispiel Firmware-Updates aus der Cloud empfangen. Meist werden von Patienten jedoch Mobiltelefone zur Konfiguration, Steuerung und Überwachung ihrer Hörgeräte verwendet. In ähnlicher Weise verbinden Hörgeräteakustiker ihre PCs mit den Hörgeräten, um diese an den Hörverlust des Patienten anzupassen.

Die Absicherung dieser Verbindungen gegen Cyberangriffe ist von großer Bedeutung. Ein Angreifer könnte in der Lage sein, Kontrolle über das Hörgerät zu erlangen und beispielsweise einen sehr lauten Ton abspielen, welcher das Gehör der Betroffenen (noch mehr) schädigt. Weiter könnten die Mikrofone des Hörgeräts dazu verwendet werden, Nutzer abzuhören. Bluetooth bietet zwar von Haus aus sichere Verbindungen auf der Basis von Pairing, jedoch ist das erreichbare Sicherheitsniveau begrenzt, da Hörgeräte aufgrund ihrer eingeschränkten Benutzeroberfläche keine authentifizierten Pairings ermöglichen.

Um dieses Problem zu lösen, hat ein Hörgerät-Produzent ein proprietäres Sicherheitsprotokoll auf Anwendungsebene entwickelt, welches einen authentifizierten, sicheren End-to-End-Kanal vom Hörgerät zu einer Applikation auf PC / Mobiltelefon oder in der Cloud bereitstellt. Ein proprietäres Protokoll war notwendig, weil bestehende Sicherheitsprotokolle aufgrund der extremen Hardware-Ressourcen-beschränkungen in Hörgeräten nicht verwendet werden konnten. Proprietäre Sicherheitsprotokolle haben jedoch den Nachteil, dass sie weniger von Fachleuten geprüft und getestet werden als standardisierte Protokolle und daher als weniger sicher gelten.

Glücklicherweise wird die nächste Generation der Hardware-Plattform dieses Produzenten über mehr Speicher und CPU-Leistung verfügen. Dies könnte die Umstellung auf ein standardisiertes Sicherheitsprotokoll ermöglichen. Es ist jedoch nicht zu erwarten, dass die Geschwindigkeit der Bluetooth-Verbindung in gleichem Mass zunehmen wird, so dass dies die grösste Hürde darstellt. Das Sicherheitsprotokoll muss so wenig Overhead wie möglich verursachen, um ressourcenschonend zu sein.

Ziel dieser Arbeit ist herauszufinden, inwiefern es möglich ist, für die Hörgeräte ein standardisiertes Sicherheitsprotokoll (z.B. IPsec, TLS, DTLS) zu verwenden und gleichzeitig einen minimalen Kommunikations-Overhead durch dieses Protokoll zu gewährleisten. Aufgaben

Die Bachelorarbeit umfasst die folgenden Teilaufgaben:

- Einarbeitung ins Thema:
  - standardisierte/verbreitete Security-Protokolle (z.B. IPsec, TLS, DTLS)
  - Protokoll-Varianten und -Konfigurationen (z.B. header compression)
  - Übersicht über verfügbare C/C++ Implementationen davon
- Inbetriebnahme der Prototypen-Hardware (MIMXRT685-EVK Boards)
- Implementation eines Prototyps in C++ mit folgenden Eigenschaften:
  - Test-Applikation, welche eine sichere Verbindung aufbaut und Test-Daten (z.B. mit verschiedenen Paketgrössen) verschickt/empfängt.
  - Modulare Implementation der sicheren Verbindung, sodass verschiedene Protokolle/Implementationen einfach und schnell ausgetauscht/getestet werden können.
  - Test-Infrastruktur, welche den Kommunikations-Overhead für die sichere Verbindung messen kann.
- Research:
  - Für jedes Protokoll: Vornehmen von Feinabstimmungen der Protokollkonfiguration und ev. Durchführung kleinerer Anpassungen an einem solchen Protokoll, mit dem Ziel, möglichst wenig Kommunikations-Overhead zu generieren.
- Evaluation für jeden Test:
  - Was ist der resultierende Overhead?
  - Welche Anpassungen an die Protokoll-Konfiguration oder die Implementation wurden dafür gemacht?
  - Einschätzung zur Sicherheit dieser Lösung, basierend auf:
    - Wie gut ist die Protokoll-Implementation/Konfiguration verbreitet/getestet weltweit?
    - Möglicher negativer Einfluss der Modifikation darauf?
- Wahrnehmen der Projektleitung insbesondere Erstellen von Projektplan und Protokollieren von Besprechungen.
- Dokumentation der Arbeit: Erstellen eines Berichtes über die Arbeit.

C.2 Organisatorisches

- In der Regel findet eine wöchentliche Besprechung mit den Betreuern (InES, Hörgerät-Produzent) statt.
- On-site Arbeit oder Workshops beim Hörgerät-Produzenten bei Bedarf
- Zusätzlich zu den generellen Vorgaben soll die Dokumentation folgende Punkte enthalten
  - Dokumentation der Konzepte und Lösungen: Aus welchen Komponenten besteht das System und wie funktioniert es?
  - Verbesserungsvorschläge bzgl. Schlüsseleigenschaften bzw. Dokumentation von Herausforderungen und Schwierigkeiten
  - Vollständige Informationen für den Nachbau des Systems - Begründungen zur Nachvollziehbarkeit von Design Entscheidungen
- Das Schreiben der Dokumentation soll parallel zur Umsetzung erfolgen. Die Planung soll Meilensteine für die Dokumentation enthalten. Drei Wochen vor dem Abgabetermin der Arbeit soll ein erster Entwurf abgegeben werden.

C.3 Allgemeine Rahmenbedingungen

- Ausgabe der Arbeit: Montag, 13.02.2023
- Abgabe der Arbeit: Freitag, 09.06.2023
- Umfang: 12 Credits. Dies entspricht einer Arbeitsbelastung von etwa 360h.
- Die Bewertung erfolgt anhand des vorgegebenen Rasters. Projektverlauf, Leistung, Arbeitsverhalten 1/3; Qualität der Ergebnisse 1/3; Form und Inhalt des Berichts und der Präsentation 1/3
- Beachten Sie die Anforderungen der Hochschule auf dem Intranet und in den Emails des Studiengangsekretariates.

Appendix D

Project Plan

See the project plan on the following page.

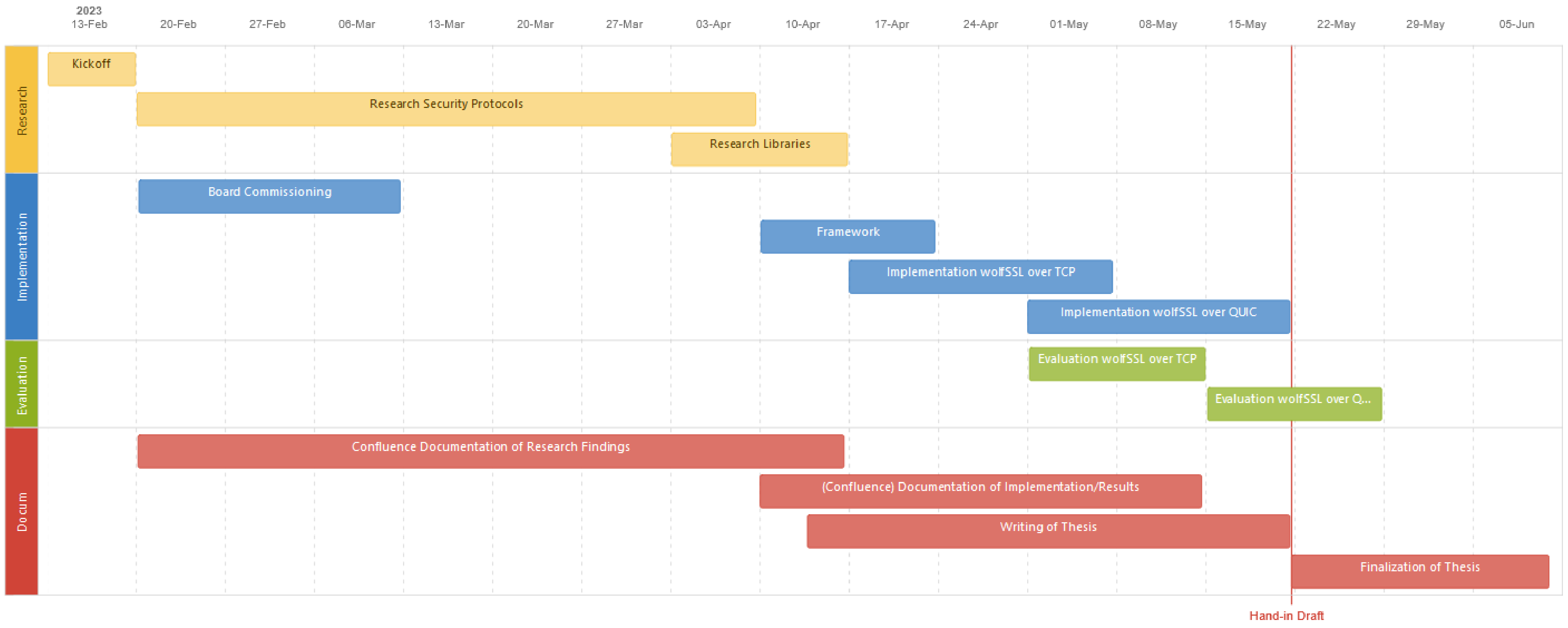


FIGURE D.1: Project time table.