Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Evgenia Kazakova

WireGuard for Securing Constrained Application Protocol for IoT Devices (CoAP)

Master's thesis in Communication Technology and Digital Security
Supervisor: Stig Frode Mjølsnes
Co-supervisor: Christian Tellefsen
June 2021

NTNU
Norwegian University of Science and Technology

Evgenia Kazakova

WireGuard for Securing Constrained Application Protocol for IoT Devices (CoAP)

Master's thesis in Communication Technology and Digital Security
Supervisor: Stig Frode Mjølsnes
Co-supervisor: Christian Tellefsen
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

Title: WireGuard for securing Constrained Application Protocol for IoT devices (CoAP)

Student: Evgenia Kazakova

Problem description:

WireGuard is a cryptographic encapsulation IP tunnel protocol implemented as a kernel virtual network interface on the Linux kernel. WireGuard has been proposed as a replacement for IPsec and OpenVPN due to its small code base, faster performance, configuration simplicity, and employment of state-of-the-art cryptography.

A major challenge when building Internet of Things (IoT) networks is securing communication between resource-constrained devices while still achieving adequate performance. For instance, Constrained Application Protocol (CoAP) is a lightweight IoT application protocol (Internet Engineering Task Force (IETF) standard) that adheres to the client/server Representational State Transfer (REST) communication model. CoAP is based on User Datagram Protocol (UDP) and is often secured using Datagram Transport Layer Security (DTLS). However, some researchers [GBP+21, RS16] have questioned whether DTLS is the best and most suitable channel security mechanism in this context, both in terms of performance, functionality, and security.

This master's thesis work will investigate the feasibility of using WireGuard to secure IoT communications. Moreover, it aims to study whether it can potentially be an advantageous replacement for DTLS when securing CoAP. This will be done by comparing WireGuard and DTLS and determining the necessary resources for such resource-constrained devices, in terms of computation time, message length, packet overhead, throughput, power consumption, etc. Analytical findings should be supported by measurements on an experimental setup.

Date approved: 2021-02-18

Responsible professor: Stig Frode Mjølsnes, NTNU

External supervisor: Christian Tellefsen, Thales

Abstract

With the ever-rising popularity of Internet of Things (IoT) devices, the issue of securing communication between the constrained nodes in the IoT network while still achieving adequate speed and performance is as significant as always. One of the protocols widely used for that purpose is the Constrained Application Protocol (CoAP). Designed as a lightweight and optimised version of Hypertext Transfer Protocol (HTTP), CoAP utilises the client/server REST architecture and runs on UDP. To secure communication, CoAP uses Datagram Transport Layer Security (DTLS) protocol. However, prior research and analysis of DTLS security mechanisms and performance overhead have shown that DTLS is yet to become the most efficient and secure protocol for IoT communication.

At the beginning of 2020, a new protocol has emerged, which could potentially solve some of the CoAP/DTLS issues. WireGuard is a cryptographic encapsulation IP tunnel protocol initially designed to implement a minimal Virtual Private Network (VPN) into the Linux kernel. Similarly to DTLS, WireGuard runs on UDP; however, unlike DTLS, it is cryptographically opinionated, meaning that it utilises a fixed set of encryption algorithms. Furthermore, it presents some modifications which make the protocol more energy-efficient than its alternatives.

This study explores the central security and performance requirements for IoT devices and summarises them in terms of Confidentiality, Integrity, Availability (CIA) triad. Furthermore, it presents CoAP in greater detail, followed by a presentation of DTLS and WireGuard. The fundamental part of the project is a comparative study between CoAP and CoAP over DTLS and WireGuard respectively. The reference implementations of the protocols are used to acquire the necessary data, and the experimental results are evaluated based on the criteria and attributes chosen beforehand. The results are further used to determine whether it could be beneficial to use WireGuard instead of DTLS.

Sammendrag

Populariteten til Internet of Things-enheter (IoT) har vært i stadig økning de siste årene, og spørsmålet om hvordan man kan sikre kommunikasjonen innad i IoT-nettverket og fremdeles beholde tilstrekkelig hastighet og ytelse er like relevant som alltid. En av protokollene som er mye brukt til det formålet er Constrained Application Protocol (CoAP). CoAP ble designet som en lettere og mer optimalisert versjon av Hypertext Transfer Protocol (HTTP) og bruker klient / server REST-arkitektur og kjører på UDP. For å sikre kommunikasjon, bruker CoAP Datagram Transport Layer Security (DTLS)-protokoll. Tidligere analyse av DTLS sine sikkerhetsmekanismer og ytelsesomkostninger har imidlertid vist at DTLS ikke nødvendigvis er den mest effektive og sikre protokollen for IoT-kommunikasjon.

En ny protokoll, som først ble presentert i begynnelsen av 2020, kan imidlertid løse noen av CoAP/DTLS sine problemer. WireGuard er en protokoll for kryptografisk innklapsling av IP-tuneller, og ble opprinnelig designet for å implementere en VPN tjeneste direkte i Linux-kjernen. Til tross for at både DTLS and WireGuard kjører på UDP, er WireGuard cryptographically opinionated, noe som betyr at den bruker et fast sett med krypteringsalgoritmer. Videre anvender den noen modifikasjoner som gjør protokollen mer energieffektiv enn de alternative protokollene.

Denne oppgaven utforsker de sentrale sikkerhetskravene til IoT-enheter og oppsummerer dem med tanke på CIA triad (Konfidensialitet, Integritet og Tilgjengelighet). Videre gir den en mer detaljert presentasjon av CoAP, etterfulgt av en grundigere presentasjon av både DTLS og WireGuard. Den grunnleggende delen av prosjektet er en komparativ studie mellom CoAP, CoAP over DTLS og CoAP over WireGuard. Referanseimplementeringene av protokollene er brukt til å skaffe de nødvendige dataene, og de eksperimentelle resultatene er evaluert basert på kriteriene og attributtene som er valgt på forhånd. Disse resultatene er videre brukt til å fastslå om det kunne være fordelaktig å bruke WireGuard i stedet for DTLS.

Preface

This master thesis was written in the spring of 2021 as the final part of the five-year Master of Science (MSc) in Communication Technology program at the Department of Information Security and Communication Technology (IIK). The research was carried out between January and June of 2021.

Firstly, I want to express my gratitude to my supervisors, Stig Frode Mjølsnes and Christian Tellefsen, for their time, support and guidance while writing this thesis. Their contributions to this thesis, both in the form of good advice and encouragement, were greatly appreciated. I have learned a lot during this project and am very grateful they could share this experience with me.

Furthermore, I would like to thank our advisor, Laurent Paquereau, for always answering my questions about courses, exchange year, and various other questions I have come up with during these last years. I would also say thank you to my friends from Class 2021 for being incredibly supportive and pleasant during these past five years. Most importantly, I want to thank them for helping me to keep up my courage when writing this thesis and reminding me to take a break when it was most necessary.

Lastly, I would like to thank our family friend, Natalia Andreassen, for proofreading my thesis and giving me valuable advice on making it even better. Last but not least, I want to give a special thank you to my parents for continuously checking up on me and lifting my spirits even when I did not think I needed it.

Evgenia Kazakova
Trondheim, June 2021

Contents

List of Figures

List of Tables

List of Acronyms

AMQP Advanced Message Queuing Protocol.

CBC Cipher Block Chaining.

CIA Confidentiality, Integrity, Availability.

CNN Constrained-Node Network.

CoAP Constrained Application Protocol.

CPU Central Processing Unit.

DDoS Distributed Denial-of-Service.

DoS Denial-of-Service.

DTLS Datagram Transport Layer Security.

ECDH Elliptic-Curve Diffie-Hellman.

HMAC Hash-based Message Authentication Code.

HTTP Hypertext Transfer Protocol.

IETF Internet Engineering Task Force.

IoT Internet of Things.

IP Internet Protocol.

MITM Man-In-The-Middle.

MQTT Message Queuing Telemetry Transport Protocol.

OSCORE Object Security for Constrained RESTful Environments.

pps packets per second.

REST Representational State Transfer.

RTT Round-trip Time.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

UDP User Datagram Protocol.

VPN Virtual Private Network.

WSN Wireless Sensor Network.

XMPP Extensible Messaging and Presence Protocol.

Chapter 1
Introduction

1.1 Motivation

Today, a wide range of various IoT devices is being used in all spheres of life. Smart homes, smartwatches and smart cars are well incorporated into our day-to-day life, whilst other types of IoT devices are being used in the fields of healthcare, transportation [WLP18, ZKKK19], manufacturing and logistics, among others. In other words, it is becoming more and more common to use such devices for safety and security-critical applications. According to Gartner [Hun17], the estimated number of IoT devices in the world in 2020 was around 20 billion. Furthermore, based on their research, roughly 25% of all the identified attacks against businesses are IoT based, making it evident that security plays a pivotal role for IoT devices.

With the fast growth of IoT devices usage, the amount of vulnerabilities to be exploited is inevitably increasing as well [Asi17]. However, different devices have particular requirements to be met to function securely and efficiently. These requirements have created a need for various communication protocols for securing end-to-end communication in the networks constructed of IoT devices. Among such protocols are Message Queuing Telemetry Transport Protocol (MQTT) and MQTT for Sensor Networks (MQTT-SN), Constrained Application Protocol (CoAP), Advanced Message Queuing Protocol (AMQP) and HTTP for IoT systems.

Two of the most popular messaging protocols for resource constrained devices such as IoT devices are MQTT-SN and CoAP. Both protocols are able to ensure bandwidth- and energy-efficient communication [TMV+14] on an acceptable level and are able to provide security using Datagram Transport Layer Security (DTLS). The utilisation of DTLS comes from both protocols running on UDP instead of Transmission Control Protocol (TCP) and thus not being able to rely on Transport Layer Security (TLS) as an encryption protocol. DTLS was designed to be a UDP implementation of TLS and consequently provide equivalent security guarantees [RM12].

As a consequence of the widespread popularity of IoT devices, a considerable amount of research discussing current security challenges they face has been published [MYAZ15, ZCW+14]. Using protocols tailored to IoT communication is thus important in order to mitigate these issues as much as possible. Since MQTT-SN and CoAP are two of the most used IoT protocols, several studies have been conducted to test the efficiency and security features of MQTT-SN and CoAP. The results published in various papers show that both protocols provide limited security features and could easily be subjected to different types of attacks. These attacks include Sniffing, Spoofing, Denial-of-Service (DoS) and Hijacking [AN19]. In addition, the protocols are vulnerable to attacks such as Cross-Protocol and Man-In-The-Middle (MITM) attacks, as well as Replay and Relay attacks [NC20, SHB14].

Since DTLS is often used to secure CoAP, the security of DTLS has also been broadly researched in the previous years. The research has helped to uncover several security issues with the protocol. According to [FBJM+20], the protocol could be subjected to attacks such as Heartbleed, CBC padding oracle and fuzzing[1]. With IoT security being a central topic of discussion in the last couple of years, several potential improvements for DTLS have been proposed [CCCP15, LS14].

Nevertheless, researchers are on the lookout for new possible solutions. One such solution has recently been introduced in the form of a new encapsulation Internet Protocol (IP) tunnel protocol called WireGuard. Released on the 30th of March 2020, WireGuard relies on UDP and operates with a fixed set of encryption algorithms in order to make it easier to manage. Moreover, WireGuard is not a "chatty"[2] protocol [Don17]. The creators of WireGuard claim it to be the simplest and fastest VPN protocol while simultaneously being extremely secure. Already a few months after its release, WireGuard was being discussed as a possible future replacement for well-known VPN protocols such as OpenVPN and IPSec, which are far more complex. Furthermore, the previously mentioned characteristics of the protocol show WireGuard to be an energy-efficient protocol, making it an attractive option for securing IoT communication. Thus, the protocol has a potential to provide security and privacy, alongside appropriate speed and performance, to small IoT sensors and other IoT devices operating with limited battery and storage capacity.

[1] *The Heartbleed bug* allows any malicious user to read the memory of the systems protected by the vulnerable versions of the OpenSSL software; *Cipher Block Chaining (CBC) padding oracle attack* is performed by getting the server to validate the padding of an encrypted message, which can provide enough data to encrypt and decrypt messages without knowing the encryption key; *fuzzing* is an automated software testing technique which involves feeding the system various permutations of data until some hackable software bugs are found.

[2] *Chatty* means that when there is no data to be exchanged, both peers will stay silent, positively affecting energy consumption

1.2 Research questions

The topic of security and privacy for IoT devices becomes progressively more relevant the more devices get connected. Moreover, attackers are becoming more innovative the more information they obtain about the existing IoT communication protocols. At the same time, many critical infrastructures relying on IoT devices, such as healthcare and transportation, require a high level of security and performance. However, the more developed and robust the protocols become, the more heavyweight they may come to be. Thus, there is a need to look for new solutions, which is one of the primary motivations for this thesis.

Unlike MQTT-SN, CoAP is specifically designed to run on DTLS. Since DTLS is the protocol a big part of this thesis will focus on, we will further concentrate our attention on CoAP, as well as CoAP and DTLS together. The goal of this thesis is to (i) present both protocols and their main issues, both with regards to security and performance; (ii) present WireGuard and discuss whether its usage could potentially be beneficial to CoAP and IoT devices using CoAP, and (iii) run experiments using CoAP with DTLS and CoAP with WireGuard, compare the results and conclude. Three research questions (RQs), introduced below, will be used to facilitate achieving these goals.

- **RQ1:** What are the main security concerns regarding confidentiality, integrity, and authentication (the CIA triad) when it comes to communication between IoT devices using CoAP?
- **RQ2:** What are the differences and similarities between WireGuard and DTLS with regards to features and how they operate?
- **RQ3:** Could WireGuard potentially be used instead of DTLS for CoAP as a protocol for securing communication?
  - **RQ3.1:** If yes, how?
  - **RQ3.2:** What would be the benefits of doing so and how would it affect the security concerns mentioned in RQ1, as well as CoAP's performance?

RQ1 and RQ2 will primarily be addressed in the first part of the thesis and later summarised in Chapter 6. To support the investigating of RQ3, a set of the following hypotheses was formed:

- **H1:** WireGuard is faster than DTLS when used to secure CoAP.
- **H2**: WireGuard adds less overhead than DTLS when used to secure CoAP.

- **H3:** It would be beneficial to choose WireGuard over CoAP rather than DTLS when securing CoAP.

1.3 Outline

This master thesis has the following structure:

Chapter 2 provides an introduction to the main protocols and their vulnerabilities. It also introduces constrained devices and networks, and the main security challenges faced by the said devices and networks.

Chapter 3 presents the research method used in this study.

Chapter 4 introduces the experimental setup used used to conduct the experiments during our research.

Chapter 5 presents and discusses the results obtained during the experimental phase of our study.

Chapter 6 discusses the results presented in chapter 4 in light of RQs and Hypotheses defined in Chapter 1.

Chapter 7 concludes this the master thesis and discusses with the future work proposed for this topic.

Appendix A includes the client and server configurations, as well as the screenshot of the outcome of running the HelloWorld server.

Appendix B presents the screenshots of WireShark captures for the three implementation of CoAP tested during our experiments.

Appendix C presents the final results in a table format.

Chapter 2
Background and related work

This chapter will first focus on explaining constrained devices and networks, along with some of the security and performance requirements such network and nodes commonly have. The three main protocols - CoAP, DTLS and WireGuard - will subsequently be introduced in greater detail, followed by a discussion of their main shortcomings, discovered and presented in earlier papers. Additionally, some of the proposed solutions for improving CoAP and DTLS will be mentioned.

2.1 Constrained devices and networks

The use of Internet of Things has been steadily increasing over the last two decades. With such devices being used for everything from facilitating remote monitoring of both patients and equipment in healthcare [WLP18] to improving parking systems in transportation [ZKKK19], the importance of these is evident. As a result of their popularity and widespread use, it is vital to keep the cost of IoT devices as low as possible. At the same time, Internet of Things devices are commonly small devices with limited Central Processing Unit (CPU), storage and power resources, which tend to operate with limited payload size and insufficient bandwidth [BEK14]. Due to those restrictions, devices with aforementioned characteristics are often called *constrained devices* or *constrained nodes*. The significance of constrained devices makes it essential to find an adequate balance between the cost of a device and its effectiveness.

Constrained devices are often in charge of gathering and storing information, which is later sent to another IoT device and one or more servers. Depending on the application, a device might also be in charge of pre-processing the data or perform some physical action on it, such as displaying it [BEK14, ZKKK19]. Constrained nodes that are connected and form a network are generally referred to as a *constrained network* or *CNN*. An example of a CNN can typically be a Wireless Sensor Network (WSN) consisting of nodes with limited resources (Figure 2.1). Nodes in a constrained

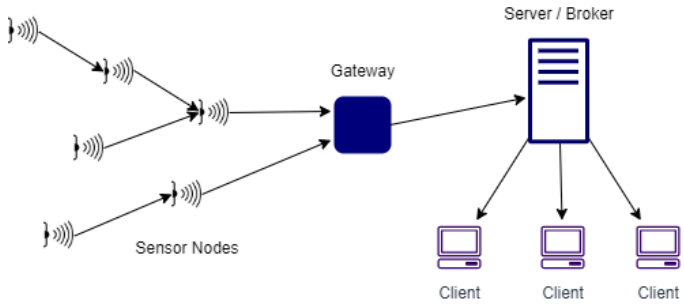


Figure 2.1: An example of a CNN: a Wireless Sensor Network (cf. [TMV+14])

network might be of different sizes and have different constraints and requirements. Hence, such networks can display unique constraints, e.g., low throughput, high packet loss, asymmetric link characteristics and a limit on reachability over time[1] [BEK14].

The modern-day connectivity and interactivity between various constrained devices create a need for communication protocols designed explicitly for such lightweight machine-to-machine (M2M) communication. CNNs require a protocol that is not only energy- and bandwidth-efficient but is also capable of securing connections between constrained nodes.

2.1.1 Main IoT security and performance issues

To understand why particular protocols are better suited for establishing and securing communication in CNNs, it is necessary to discuss the security and performance requirements for the IoT devices that compose such networks.

The main goal of information security is to govern the *confidentiality, integrity*, and *availability* of existing information assets, which include information resources such as data, information, hardware and software [WO20]. These three principles are commonly recognised as the *CIA triad*, where confidentiality refers to protecting data from unauthorised access, and integrity involves securing the data validity against undesired changes. Availability, in its turn, relates to ensuring the accessibility of the information to authorised parties and processes in the form and format needed [WO20].

The security goals of the CIA triad apply to the world of IoT as well. However, as previously mentioned, small IoT devices, or constrained devices, have a

[1]Due to limited battery capacity, IoT devices are usually programmed to go to "sleep mode" or "deep sleep" by default when not used for some time and wake up periodically to perform an action they are designed for.

manifold of constraints and limitations when it comes to computational and power resources. Hence, not all existing practices and protocols are suitable for securing IoT communication.

The main security challenges with regard to IoT devices can largely be divided into two main categories: *technological* challenges and *security* challenges [MYAZ15]. Alternatively, the two categories can be related to the two main factors IoT security is influenced by: the diversity of the devices (heterogeneity) and the communication medium (networking environment) of the IoT devices [ZCW+14].

Technological challenges are often the result of the heterogeneous[2] nature of the IoT devices, and are typically related to scalability, wireless technologies and power constraints [MYAZ15]. Furthermore, low computational power leads to low cognitive capability, which is becoming a progressively more significant issue as the volume of data generated by and circulated in the IoT-based network only increases [KOL19]. Another issue related to the heterogeneity of the devices and the vast scale of the IoT is created by the way the devices and programs are designed. Every device is different and vulnerable in its own way, which creates various possible attack surfaces. The heterogeneity and the vast number of existing devices make securing all of them in the same way difficult, creating a complex security problem [MYAZ15]. Adding standardised security mechanisms for security patching into the devices may be a possible solution to this issue. However, they too need to be carefully designed to suit all the platform they may be implemented on.

Other *security* challenges are related to the fundamentals and functionalities which should be enforced to achieve a secure network [MYAZ15]. Authentication and authorisation are among some of the most crucial principles necessary for attaining secure communication. However, due to the number of different entities involved in the process, as well as the fact that resource-constrained devices often have to interact with each other directly, these principles may be hard to obtain [MYAZ15, ZCW+14].

Many IoT devices collect personal data and information about users' behaviour and daily routine to enhance the user experience. Furthermore, the devices can interchange said information with each other to collectively provide better services. Thus, preserving users' privacy is another security challenge that cannot be ignored.

When it comes to securing the communication medium used by IoT devices, the networking environment is generally expected to be heterogeneous [MYAZ15]. However, various communication media may experience different security issues. As a consequence, the problem of compatibility between the devices from different networks

[2] *Heterogeneous* - composed of parts of different kinds; having widely dissimilar elements or constituents; varied; diverse (from: Dictionary.com)

arises. The usage of different protocols will make protecting the communication and the shared data more complex.

The central security and performance issues for resource-constrained devices are interconnected. To provide adequate security and satisfactory performance, the protocols have to be suitable for devices with limited computing power and battery capacity. Many cryptosystems and protocols which have been proposed for IoT and are considered secure and robust are not a good fit for resource-constrained devices such as sensor nodes and other smart devices. It imposes certain limitations for the design and implementation of protocols, especially when it comes to encryption and authentication of data and key distribution and management. Lightweight solutions that are compatible with the device capabilities are thus required. Constrained Application Protocol (CoAP) is one of the existing lightweight communication protocols that have been proposed to address the networking needs of resource-constrained devices.

2.2 An overview of Constrained Application Protocol

As defined in RFC 7252 [SHB14], Constrained Application Protocol (CoAP) is a specialised web transfer protocol based on a request/response interaction model between application endpoints. Designed as an optimised version of HTTP, CoAP is widely used for providing and securing communication between constrained nodes in CNNs. To understand CoAP, we must look at it's predecessor, HTTP, first.

HTTP is an application-level request/response protocol which has been used since 1990 [FGM+96] and is based on Representational State Transfer (REST). REST is a client/server architecture style which makes information available as *resources* identified by Uniform Resource Identifiers (URIs) [BCS12]. To access the resources, a client must request them using one of the four fixed methods - GET, PUT, POST or DELETE. HTTP adds several more methods, including HEAD, PUT, CONNECT, OPTIONS and TRACE [FR14]. Hence, HTTP methods may be used to not only retrieve, but also create, update and delete resources [MP+].

Using any of the aforementioned methods, a client can initiate a synchronous request-response communication with a server. A client can either be a user-agent (most commonly a Web browser), or a proxy sending requests on behalf of the client. The overall operation of HTTP is defined in RFC2616 [FGM+96] and is described as follows (Figure 2.2):

1. To send a request to the server, there must be an open TCP connection between the client and the server. The client can thus either open a new connection or utilise an already existing one.

2. The client sends a request to the server in the form of a URI and protocol version (e.g. GET / HTTP/1.1), followed by a MIME[3]-like message consisting of an optional set of HTTP headers, and possible body content. An empty line before the message body indicates that all the metainformation for the request has been sent.

3. The server responds with a status line containing the protocol version and a status code (e.g. 200, 301 or 404) together with a status message (e.g. OK, Moved Permanently or Not Found respectively). The status message is further followed by a MIME-like message consisting of an optional set of HTTP headers and, depending on the type of request, a body. Similar to the request, a blank line before the body would separate the metainformation from the body.

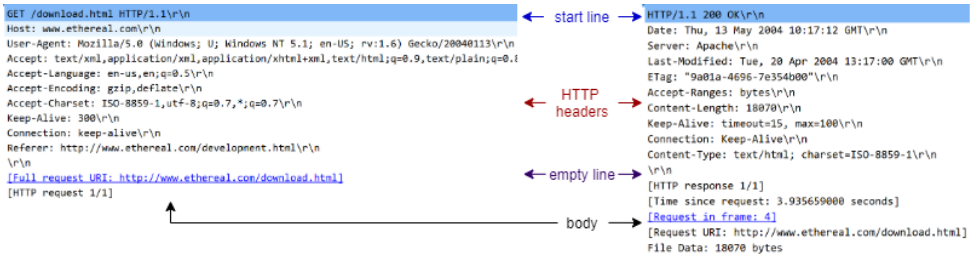


Figure 2.2: Typical HTTP request/response seen in Wireshark.

As already stated, HTTP is based on design principles of REST, one of which is statelessness [MP+]. HTTP is a stateless protocol, making each transaction an independent transaction that is not linked to any other transaction. After a request is sent and a response is delivered, the session is terminated. The server does not retain any client session data (state) between two requests. It is, however, possible to retain a session between a client and a server using HTTP Cookies, which can be added using the Set-Cookie header field [BB11]. Furthermore, although HTTP can be implemented of top of any protocol that guarantees reliable transport [FGM+96], it often takes place over TCP/IP, ensuring that the communication is reliable despite HTTP being stateless.

Having been in use for three decades, HTTP is undoubtedly a favourable and effective protocol. During that time, HTTP has been continually growing and progressively adding to the cost of both implementation code space and network resource usage [BCS12]. Hence, HTTP is a considerably expensive protocol to implement, making it unsuitable for constrained IoT devices. Thus, an optimised version of HTTP and REST for M2M communication was created.

[3]A *Multipurpose Internet Mail Extensions type*, also commonly referred to as a *media type* - a standard that indicates a type and a subtype of a document, file, or assortment of bytes [FKH13]

2.2.1 CoAP for M2M communication

Constrained Application Protocol is a specialised, lightweight web transfer protocol designed for use with constrained nodes and CNNs. The main goal of CoAP is to optimise HTTP for M2M applications, while still offering multicast support, low overhead and simplicity for CNNs [SHB14]. Similarly to HTTP, CoAP is based on REST architecture and supports the request/response model. Additionally, the protocol supports publish/subscribe architecture using an extended GET-method [TMV+14].

One of the central elements in CoAP's reduced complexity, compared to HTTP, is the usage of UDP instead of TCP. UDP, or User Datagram Protocol, is a communication protocol often used when transmission speed and efficiency is more crucial than security and reliability, such as when it comes to video playback or Domain Name Service (DNS) lookups. UDP uses a simple connectionless communication model, meaning that it does not establish a connection before the data is transferred. UDP provides a minimal amount of "regular" protocol mechanism, delivering datagrams[4] without establishing a connection first, indicating the order of the packets, or providing any guarantee of delivering or duplicate protection [P+80]. Avoiding the overhead of the said mechanisms makes UDP significantly faster than TCP, albeit much less reliable.

A CoAP request message is equivalent to a HTTP request message and is sent by a client to request action on a chosen resource identified by a URI, using one of the provided methods. The server gets the request and response with a status code, sometimes together with a source representation.

CoAP's messaging model is based on the exchange of messages over UDP, and the message format is shared both by request and response messages. Within UDP datagrams, COAP uses a four-binary header, followed by a sequence of binary options and a payload. As UDP does not provide reliability, CoAP gives every message a Message-ID, which is used to mark a message as Confirmable (CON). A default timeout and exponential back-off between retransmissions are used to retransmit a Confirmable message until the sender gets a message (ACK) with the correct Message-ID acknowledging that the message has been received (Figure 2.3, (b)). Messages that do not require as much reliability can be sent as Non-confirmable messages (NONs) (Figure 2.3, (a)). These messages will not get acknowledged, but they are still given a Message-ID to detect potential duplicates [SHB14]. In both cases (CON and NON), if a recipient cannot process the message and provide a suitable error response, it will reply with a Reset message (RST).

Another characteristic of CoAP is the use of an asynchronous message approach.

[4]UDP packets are referred to as datagrams.

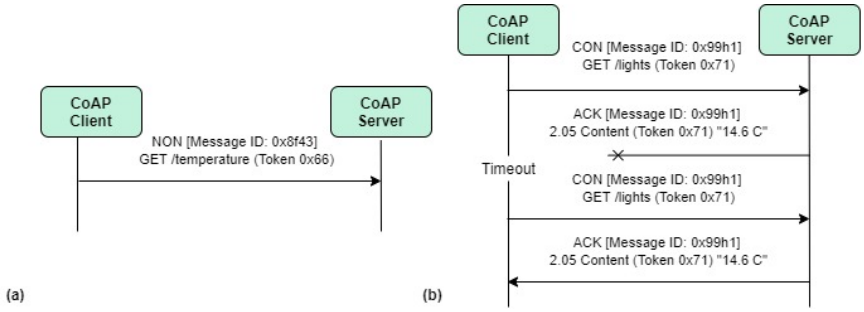


Figure 2.3: CoAP request/response examples. (a) CoAP Non-confirmable Message (NON) exchange; (b) CoAP Confirmable Message (CON) exchange, an example of a datagram loss being fixed by retransmission using Message ID.

When using HTTP, the transactions must always be initiated by the client, meaning that if a client wants to get an update on a certain resource, it has to send a GET request over and over again. This actions is called *polling*. In CoAP, however, a client can indicate its interest in being updated when the status of any resource changes. It is done by sending a GET request with a specified option called "Observe" [BCS12].

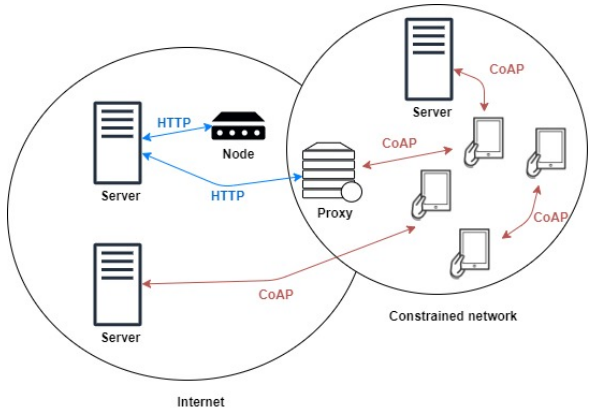


Figure 2.4: An example of a Web architecture with HTTP and CoAP (cf. [BCS12] and [Kaz20])

However, one of the main features of CoAP is providing a stateless HTTP mapping. This allows for building of proxies to provide access to CoAP resource via HTTP in a uniform way, and for HTTP simple interfaces to be realised alternatively over CoAP [SHB14]. In other words, enabled by the REST architecture, CoAP is capable of interworking with HTTP via proxies (Figure 2.4). As a consequence of having equivalent architecture, methods, response codes, and options, the mapping between the two protocols is straightforward [BCS12]. A proxy is usually able to translate

between CoAP and HTTP without putting any additional requirements on the client or the server, as it understands the semantics and syntax of both protocols. CoAP being capable of interworking with HTTP is crucial for the world of IoT, as it allows constrained devices to integrate into the Web.

2.2.2 Securing CoAP with DTLS

The most widely deployed protocol for securing network traffic is the Transport Layer Security protocol, or TLS [RM12], which is often inserted between an application layer and a transport layer to secure an application protocol. However, since TLS must run over a reliable transport channel, it cannot be used to secure UDP communication and is therefore not suitable for securing CoAP.

To solve this issue and to provide TLS-like security guarantees to datagram protocols, Datagram Transport Layer Security (DTLS) was created. DTLS is based directly on TLS and allows client/server applications to communicate without eavesdropping, tampering, or message forgery [RM12].

2.3 An overview of Datagram Transport Layer Security (DTLS)

The main reason for TLS requiring a reliable transport channel is that in a datagram environment (such as UDP), packets might get lost or delivered out of order. TLS has no mechanism to handle such situations, meaning that if a TLS packet is lost, the connection will be broken. The same applies to the TLS handshake: messages must be transmitted and received in a specific order; otherwise, an error will occur. Thus, DTLS needed to introduce new solutions in order to use TLS security mechanisms for datagram protocols.

Firstly, DTLS solves the issue of packet loss by using retransmission. When a client sends a *ClientHello* message, it expects to see a *HelloVerifyRequest* message from the server. If that does not happen before the timer expires, the *ClientHello* message will be retransmitted. The server also has a retransmission timer, retransmitting the lost packets when it expires. The problem of the handshake messages arriving in the wrong order is solved by assigning a specific sequence number to each of the messages. If the received message has the wrong number, it is queued until all the preceding messages have been received. In addition to that, DTLS supports replay detection and utilises fragmentation to deal with the issue of DTLS messages being too large for UDP datagrams [RM12].

Furthermore, DTLS employs a stateless cookie exchange mechanism to prevent DoS attacks. As explained in [RM12], a typical DoS attack is when an attacker

can transmit a large series of handshake initiation requests to the server, which causes the server to allocate all its resources to perform expensive cryptographic operations. An alternative DoS attack can be performed using a server as an amplifier by sending a *ClientHello* message with a spoofed IP address, making the server flood the victim with a large Certificate message (next message after *ClientHello*). In DTLS, when a client sends a *HelloClient* message, the server might respond with a *HelloVerifyRequest* message, containing a stateless cookie. The client must then return a *HelloClient* message with the given cookie, and the server proceeds with the handshake only if it can verify the cookie. The server is not required to perform this exchange, but the client has to be prepared to respond to it when required [TF16].

A fully authenticated DTLS handshake can be seen in Figure 2.5 and is detailed below:

- **Flight 1 and 2:** The client initiates the communication by sending a *ClientHello* message, which has information about the highest supported DTLS version. Moreover, it includes a random nonce, the cipher suite supported by the client, and optional extensions [FBJM+20]. The server responds with a *HelloVerifyRequest* message containing a stateless cookie. The cookie is then returned in a new *ClientHello* message.

- **Flight 3 and 4:** The client repeats the *ClientHello* message, and the server responds with a *ServerHello* message containing the server's DTLS version, the cipher suite chosen by the server, a second random nonce, and optional extensions. The server also sends its certificate with the server's public key and an optional *CertificateRequest* if the server is expecting the client to authenticate. In *ServerKeyExchange*, the server sends an ephemeral public key which is signed with the private key for the server's certificate.

- **Flight 5:** If requested and supported, the client will send its certificate back to the server; the same applies to the *CertificateVerify* message. They contain the certificate of the client and a signature computed over all previous messages using the client's long term private key. The *ClientKeyExchange* message contains a `pre-master secret`. The content of this message depends on the cipher suite selected earlier in the handshake. The `pre-master secret` is encrypted using the server's public key. In addition, the client's public key is sent to the server.

  The random nonces from *ClientHello* and *ServerHello* combined with the `pre-master key` are used to compute the `master_secret`. The *Finished* message (see Flight 6) will be the first to be encrypted using the `master_secret`. Both the client and the server then use this information to derive symmetric keys, which are then used for the rest of the session. The *ChangeCipherSpec*

message indicates that all subsequent messages will be encrypted with the negotiated cipher suite and keying material. Finally, the *Finished* message is sent, which contains an Hash-based Message Authentication Code (HMAC) over the previous handshake messages, and is encrypted with the new keys.

- **Flight 6:** The server answers with its own *ChangeCipherSpec* and *Finished* messages. After that, both client and server can exchange authenticated and encrypted application data, and the handshake is completed.

Since DTLS handshake is very expensive, there is generally no need for DTLS to establish a new session every time a client wants to request something from a server. RFC7925 [TF16] describes a helpful feature that has been implemented both for TLS and DTLS called Session Resumption. Session Resumption allows a client to continue an already established session, meaning that there will be no need to complete a full handshake once again. The feature improves the performance of the handshake and leads to fewer messages being exchanged, eliminating some of the computational overhead that comes with establishing a connection one more time.

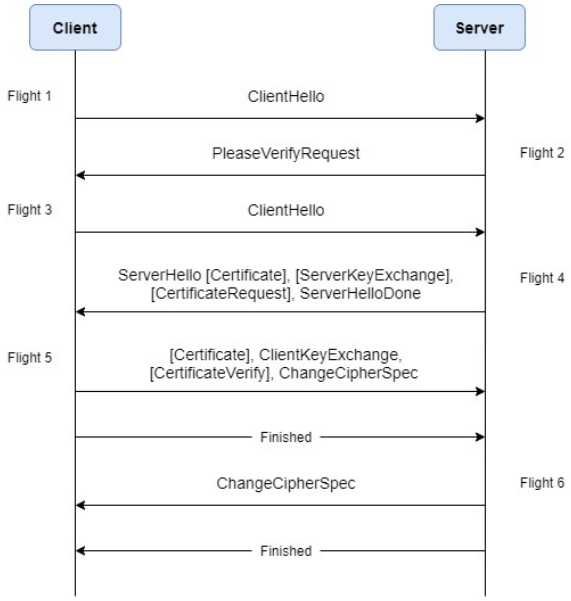


Figure 2.5: DTLS handshake (cf. [FBJM+20])

.

2.3.1 CoAP enhanced by DTLS

Similarly to HTTP using TLS, CoAP supports the usage of DTLS to protect and encrypt the communication between constrained devices as CoAP itself does not

provide protocol primitives for authentication or authorisation (Figure 2.6). By adding such primitives, DTLS helps ensure confidentiality when using CoAP [NC20]. The DTLS binding for CoAP is defined in terms of four security modes: NoSec, PreshardKey, RawPublicKey, and Certificates. These four modes differ in authentication and key negotiation mechanisms, and range from no security to security based on X.509 certificates [NC20]. These four modes are described in [SHB14]:

1. In **NoSec** mode, DTLS is disabled, meaning there is no protocol level security. The packets are simply sent over normal UDP and IP as usual.

2. **PreSharedKey (PSK)** mode enables DTLS and provides the node with a list of pre-shared keys, with each key including a list of trusted nodes it can be used to communicate with.

3. When using **RawPublicKey** mode, the node has a raw public key - an asymmetric key pair without a certificate - which is validated using an out-of-band mechanism, together with an identity calculated from that public key. The node is again provided with a list of the nodes it can communicate with. DTLS is enabled.

4. **Certificate** mode provides the node with an asymmetric key pair validated by an X.509 certificate that binds it to its subject. A common trust root signs the key pair. The node is also given a list of root trust anchors that it can use to validate a certificate. DTLS is enabled.

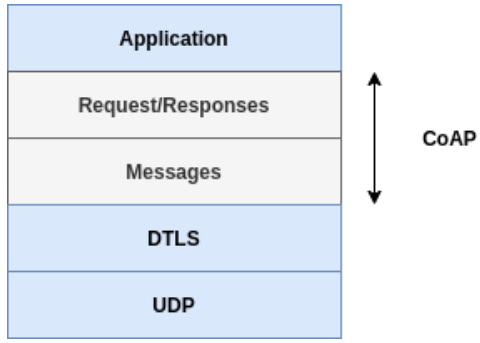


Figure 2.6: Abstract Layering of CoAP secured with DTLS

Which mode is the best for a certain device depends on the developers, who are responsible for figuring out the best trade-off between performance (energy constraints) and security requirements [NC20]. Some constrained nodes and networks are not able to support all provided modes due to limitations discussed in section 2.1. This

can also depend on which specific cipher suite is used; for example, the mandatory-to-implement cipher suite for PSK mode is TLS_PSK_WITH_AES_128_CCM_8 and TLS_Elliptic-Curve Diffie-Hellman (ECDH)E_ECDSA_WITH_AES_128_CCM_8 for the RawPublicKey mode [SHB14]. Some of the cipher suits add additional complexity to the existing protocol, which can be significant for small, constrained devices.

2.3.2 Known vulnerabilities / drawbacks in CoAP and DTLS

Due to the popularity and widespread use of both CoAP and DTLS, the protocols have been researched and repeatedly tested previously. Arvind et al. [AN19] and Nebbione et al. [NC20] identify the following potential threats and vulnerable processes of CoAP devices:

Message parsing: can be exploited if the processing logic of the peers does not properly handle incoming messages. Message parsing can have a negative effect on availability due to overload conditions and used for remote execution of arbitrary code by an adversary.

Proxying and caching: if proxies and caches do no properly implement the access control mechanisms needed, an attacker can gain access to the network and gain confidential information from the traffic.

Key generation: CoAP nodes can be compromised as cryptographic keys generation is not robust.

IP address spoofing: can potentially be exploited to launch a Distributed Denial-of-Service (DDoS) attack on UDP by forging IP addresses of legit CoAP nodes.

Cross-protocol exchanges (translation from TCP to UDP): an attacker can force a node to interpret a message to the rules of another protocol by sending a message with a spoofed IP address and a fake source port number [Kaz20].

These threats and more are discussed in more detail by Shelby et al. in RFC7252 [SHB14]. According to [NC20], improper message parsing is the most common security issue, and if exploited, it can lead to memory leaks, DoS attacks, and potentially remote code execution. To counteract some of the aforementioned issues, it CoAP might be secured using DTLS. However, although CoAP utilises DTLS to secure communication between nodes, DTLS only secures unicast (client-to-server) communication, and not multicast (one-to-many) communication, which is also supported by CoAP.

Moreover, as a result of DTLS being based on TLS, many attacks that can be

performed on TLS apply to DTLS as well. These attacks include the Heartbleed, CBC padding oracle and fuzzing attacks mentioned in Chapter 1. Furthermore, the cookie exchange mechanism presented in Section 2.3 of Chapter 2 does not provide any defence against DoS attacks from valid IP addresses [Kaz20]. It has also been determined that many TLS/DTLS cipher suits do not provide forward secrecy, which can lead to an attacker obtaining the long-term keys used to encrypt the session keys [SHSA15b]. This would allow the attacker to record encrypted conversations. Many more attacks applicable to both TLS andd DTLS have been summarised by Sheffer et al. [SHSA15a], who have also published recommendations for secure use of the protocols [SHSA15b].

When it comes to securing CoAP with DTLS, there is a widely discussed issue of DTLS possibly being too heavyweight for lightweight protocols such as CoAP [AN19, RS16]. Notably, a large message and handshake compression are among the main concerns for those debating using DTLS. However, several possible solution have been proposed lately. For example, Raza et al. [RSH+13] introduce a 6LoWPAN header compression scheme for DTLS, and Lakkudi et al. [LS14] discuss an integration of lightweight DTLS using Pre-Shared Key. Capossele et al. [CCCP15] propose to integrate DTLS over CoAP using Elliptic Curve Cryptography in order to minimise the computational overhead.

2.4 An overview of WireGuard

WireGuard is a cryptographic encapsulation IP tunnel protocol that was first presented at the beginning of 2020 by Jason A. Donenfeld [Don17]. The protocol was initially designed for the Linux kernel but has since become available for other OS such as Windows, macOS, Android, etc. WireGuard presents numerous benefits originating from a combination of modern cryptographic primitives and a simple design, together with a very small codebase and high software performance [Don17]. Indeed, the protocol is implemented in less than 4,000 lines of code, making it much easier to implement and maintain compared to other popular protocols such as IPsec and SSL/TLS-based OpenVPN (Figure 2.7). Moreover, fewer lines of code create a much smaller possible attack surface.

The fundamental principle of every secure VPN is the association between public keys of peers, and the IP addresses that those peers are allowed to use [Don17]. WireGuard uses *cryptokey routing* to secure communication and maintain a simple association mapping between public keys and the permitted IP addresses. The WireGuard interface - *a cryptokey routing table* - has a private key and a UDP port on which it listens, combined with a list of peers. The peers are identified by their public keys (a public key in WireGuard is a 32-byte Curve25519 point) and have a list of allowed source IPs. When a packet is transmitted on a WireGuard interface,

Figure 2.7: WireGuard codebase compared to other well-known VPN protocols (cf. [Don18])

the cryptokey routing table is used to decide which public key should be used for encryption.

When an interface is added, and its peers are configured using their public keys, they can begin communicating. From the user's perspective, WireGuard will appear to be completely *stateless*. The session states, connections, perfect forward secrecy and handshakes are managed "behind the scenes" and are invisible to both the regular user and the administrator [Don17].

Nonetheless, the creator of WireGuard is not only claiming the protocol to be extremely fast, but also highly secure [Don17]. One reason for WireGuard being presumably more secure than its predecessors comes from it being *cryptographically opinionated* [Don17]. Unlike IPsec, OpenVPN, or even DTLS, WireGuard does not offer an opportunity to choose between different cipher suites but instead provides a fixed set of primitives. The following primitives are provided by WireGuard, as indicated by Donenfeld [Don17]:

1. **Noise Protocol Framework**, or **NoiseIK** is used for key exchange and to provide forward secrecy (key exchange and key agreement are presented in more detail in Section 2.4.1).
2. **ChaCha20Poly1305** stream cipher is used for authenticated encryption.
3. **Curve25519 for Elliptic-Curve Diffie-Hellman (ECDH)** is used for key agreement - short pre-shared static keys are used for authentication.
4. **BLAKE2** is used for hashing.
5. **SipHash24** is used for hashable keys.

In addition to providing forward secrecy, WireGuard key exchange has built-in protection against key impersonation, replay attack, and identity hiding [Don17].

Furthermore, it utilises an improved version of DTLS IP-binding stateless cookie mechanism (discussed in Section 2.3 of Chapter 2) to add encryption and authentication, and to protect against possible DoS attacks. WireGuard is also a *stealthy* protocol, meaning that it does not respond to any unauthenticated packets, making it invisible to illicit peers and network scanners. Moreover, avoiding unnecessary "chatting" reduces the number of data packets being sent, cutting down on the amount of data available for potential sniffing and eavesdropping. Being a less "chatty" protocol also provides an opportunity for WireGuard to be more energy-efficient.

As discussed in Section 2.1.1, lightweight CNNs require lightweight solutions that provide acceptable security and privacy, together with adequate speed and performance. WireGuard ensures fast communication due to little overhead and high-speed cryptographic primitives [tea]. Combining it with the other presented features makes the protocol easy to set up and use. In addition, WireGuard can offer relatively low energy consumption, making it a suitable candidate for securing IoT communication.

2.4.1 WireGuard handshake and key exchange

The following four types of messages are present in WireGuard [Don17]:

1. *The handshake initiation message* which begins the handshake process (Figure 2.8). The message contains a cleartext ephemeral public key, the initiator's public key, which has been authenticated-encrypted using ECDH, and an authenticated-encrypted counter, used to prevent replay DoS.

2. *The handshake response to initiation message* which concludes the handshake process (Figure 2.8). Now the responder knows the initiator's static public key and ephemeral public key. The responder sends back a cleartext ephemeral public key, together with an empty buffer that is authenticated-encrypted using a key calculated using the previous message and a new round of ECDH.

3. *A reply to either the first or the second type of messages* which contains an encrypted cookie value used in resending either the rejected handshake initiation message or handshake response message.

4. *An encapsulated and encrypted IP packet* which uses the secure session established by the handshake.

As established in Section 2.4, Noise framework and ECDH are used to generate session keys. When both handshake messages have successfully been delivered, both parties can derive two symmetric authenticated-encrypted session keys from the

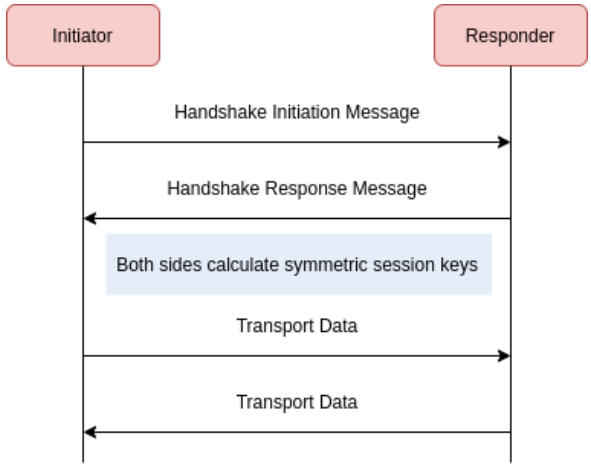


Figure 2.8: Wireguard Handshake and Key Exchange

static and ephemeral keys known to them. One of the keys is then used for sending data, while the other is used for receiving data. However, WireGuard provides a possibility for adding a 256-bit symmetric encryption key, which gets "mixed" into the initial handshake between the peers. Unlike with client/server architectures, both sides can reinitiate the handshake in order to derive new session keys, meaning that the initiator and the responder can swap roles [Don17].

The WireGuard key exchange only takes 1-RTT[5]. Moreover, it does not require any certificates, such as X.509 that are sometimes used for DTLS); similarly to SSH, both sides exchange short 32-byte base64-encoded public keys.

2.4.2 Known vulnerabilities / drawbacks

WireGuard is a protocol designed to balance high performance speed and good security, which is challenging without making some trade-offs. Due to how new the protocol is, it is yet to receive a rigorous security analysis, although some high-level analysis can already be found [ABD+15]. Donenfeld and Milner provide a computer-verified proof of the protocol in the symbolic model in [DM17], while Dowling and Peterson present a computational proof of the WireGuard handshake in [DP18]. Apart from that, the information about the security vulnerabilities in WireGuard is limited. However, the creator of WireGuard has publicly acknowledged the existing drawbacks of the protocol known to him on the WireGuard official website [Don], which include:

[5] *RTT* stands for Round-trip time and indicates the time needed for a signal to be sent, and for the sender to get an acknowledgement of that signal having been received

Deep Packet Inspection and **TCP Mode**: as WireGuard only runs over UDP, it does not support the TCP mode. In addition, the protocols do not focus on obfuscation. The task of transforming WireGuard's UDP packets into TCP packets thus falls onto a layer above WireGuard responsible for obfuscation.

Roaming Mischief: an active adversary can conduct a MITM attack and not only redirect packets but replace source IP addresses. The endpoint address can be updated, giving the attacker a chance to relay packets even after losing their MITM privileges. This can be prevented with the help of ordinary firewalling that can lock down the WireGuard socket to a particular IP address. Even if the packets do get intercepted, however, they will remain indecipherable for the adversary.

Identity Hiding Forward Secrecy: WireGuard encrypts the sender's public key using the static public key of the responder. Therefore, if the responder's private key and a traffic log of previous handshakes are compromised, it would make it possible for an adversary to figure out who has sent handshakes. The intercepted data will remain indecipherable.

Hardware: there is currently not plentiful dedicated hardware supporting the use of ChaCha20Poly1305. This could potentially be an issue, albeit it is changing as new and faster hardware gets introduced.

Post-Quantum Secrecy: WireGuard is not designed to be post-quantum secure. The secrecy of the data exchange via WireGuard depends on the security of the Curve25519 ECDH function. As mentioned above, there is an option to use an additional encryption key to provide a layer of post-quantum secrecy.

The best option for ensuring post-quantum security would be to run a post-quantum handshake on top of the protocol, inserting the resulting key into WireGuard's pre-shared key slot. Some possible post-quantum improvements to the WireGuard handshake are recommended in [KMR20]. Furthermore, a post-quantum version of WireGuard [HNS+21] was newly proposed.

2.5 Related work

Several RFC documents published by Internet Engineering Task Force (IETF) were consulted during the initial stage of our research. RFC7228 [BEK14] explains terminology for constrained-node networks. RFC2616 [FGM+96] and RFC7231 [FR14] provide information about HTTP. RFC7252 [SHB14] provides a detailed presentation of CoAP and all of its aspects. RFC6347 [RM12] describes DTLS 1.2 in detail, while RFC7925 [TF16] defines a TLS and DTLS 1.2 profile specifically with IoT in mind.

Due to the popularity of CoAP and DTLS for IoT networks, several studies evaluating the security aspects of the protocols are available to collect the information needed for creating a solid background for further research. In [BCS12], Bormann et al. give an overview of the REST architecture style, HTTP and CoAP. Arvind et al. [AN19] discusses common security issues found in CoAP and test the protocol by performing a sniffing attack on a test network with a CoAP client-server and a proxy model. Nebbione et al. [NC20] present and analyse potential security threat for protocols such as MQTT and CoAP and suggest countermeasures and good practices that could be applied to mitigate risks and enhance security. In [BMN19], Burange et al. also present vulnerabilities found in MQTT and CoAP. Moreover, they discuss the use of DTLS for securring CoAP and test CoAP by comparing the memory footprint of secured and non-secured version of CoAP.

Rahman et al. [RS16] outline security issues of 802.15.4, 6LoWPAN and RPL, protocols used for different layers of IoT communication. The paper also presents CoAP and DTLS and analyses the protocols' security status. Fiterau-Brostean et al. [FBJM+20] reveal four serious security vulnerabilities and functional bugs in DTLS by using a framework for applying protocol state fuzzing on DTLS servers, which is used it to learn state machine models for thirteen DTLS implementations. Sheffer et al. [SHSA15a] summarise known attacks on TLS/DTLS and propose security practices and countermeasures in [SHSA15b].

With regard to improving CoAP/DTLS security are a 6LoWPAN header compression scheme for DTLS proposed by Raza et al. [RSH+13] and an integration of lightweight DTLS using Pre-Shared Key discussed by Lakkudi et al. [LS14]. Another suggestion for enhancing the security and performance of CoAP/DTLS was published by Capossele et al. [CCCP15]. The paper proposes to to integrate DTLS over CoAP using Elliptic Curve Cryptography in order to minimise the computational overhead.

Being a relatively new protocol, WireGuard is yet to amass the same volume of research as CoAP and DTLS. However, there are a handful of studies discussing the security and performance of the protocol. In [ABD+15], Adrian et al. investigate the security of Diffie-Hellman key exchange, also used in WireGuard. The paper finds Diffie-Hellman to be less secure than initially believed and presents several ways to compromise it. Dowling et al. [DP18] discuss security features of WireGuard and present a computational proof of the WireGuard key exchange protocol. Donenfeld et al. [DM17] also discuss WireGuard key exchange and its properties before providing a formal, computer-verified proof of the protocol. In [KMR20], Kniep et al. discussed possible vulnerabilities of a WireGuard tunnel against quantum computers and introduced three incremental post-quantum improvements to WireGuard's handshake protocol. Moreover, a post-quantum version of WireGuard has been researched and proposed by Hülsing et al. [HNS+21].

There are several comparative studies evaluating the performance of CoAP and CoAP/DTLS against other IoT protocols. Kondoro et al. [KBTM21] present a real time performance analysis of both unsecured and secured versions of MQTT, CoAP and Extensible Messaging and Presence Protocol (XMPP). The paper evaluates a variety of metrics such as packet overhead, latency and scalability. Laaroussi et al. [LN21] provides an in-depth performance analysis of CoAP and MQTT against Object Security for Constrained RESTful Environments (OSCORE), testing both secured and unsecured versions of the three protocols. The scope of evaluation is defined in terms of latency and throughput, and the impact of extra data such as handshake introduces by security extensions is discussed. OSCORE is also evaluated against CoAP/DTLS in [GBP+21], where Gunnarsson et al. provide a comparative and experimental performance evaluation of the protocol on real resource constrained IoT devices. The performance is evaluated in terms of memory and CPU usage, energy consumption on the server side and RTT experienced on the client side.

When it comes to WireGuard, Lackorzynski et al. [LKS19] compare different VPN protocols for future industrial systems and evaluate them in terms of throughput and latency. The results indicate that WireGuard outperforms other solutions on different hardware platforms and that ChaCha/Poly1305 utilised by WireGuard performs best in resource-constrained networks. In [PWA+19], Plaga et al. presents the requirements for secure infrastructures and discusses WireGuard as a possible solution for securing IoT devices due to its simple setup and less consuming cryptographic primitives.

2.6 Contributions

The possibility of using WireGuard for IoT networks has already been voiced before. However, there is a lack of prior studies conducting a quantitative comparison of CoAP and CoAP secured by WireGuard and DTLS, respectively. Most of the research available on WireGuard focuses on comparing WierGuard to other VPN services such as IPSec and OpenVPN. Moreover, there are several comparative studies focusing on the widespread IoT protocols such as CoAP and MQTT, and their versions secured by DTLS and TLS consecutively. Nevertheless, experiments assessing the usability of WireGuard for securing CoAP and IoT networks and evaluating the results in terms of latency, handshake time, RTT and throughput have not previously been performed.

The absence of such experiments creates a gap in the research literature that this study aims to fill. The implementation of CoAP over WireGuard is particularly novel, as it, as far as we know, has not been suggested or done before. Our goals are to provide an insight on the impact WireGuard and DTLS have on the performance

of CoAP and attempt to determine which protocol is a better fit for securing an IoT environment not only in terms of performance but also security requirements.

This chapter introduced constrained devices and networks and presented an overview of the primary security challenges they face. Moreover, CoAP, DTLS and WireGuard, the protocols that are the primary focus of our research, and their drawbacks and vulnerabilities were discussed. At last, we presented relevant related studies consulted during the research and discussed how this thesis aims to contribute to the field of IoT protocols and networks.

Chapter 3
Methodology

This chapter presents the methodology used in this research. Subsequent sections define the methods chosen for conducting our study and explain why they were chosen.

As our research revolved around comparing two network protocols regarding their performance efficiency and security properties, relevant data needed to be acquired to have a solid base for comparison. RQ1 defined in Chapter 1 is a conceptual question. A *conceptual question* is a question that can be answered exclusively based on a knowledge of relevant concepts and not by performing comprehensive calculations. RQ2 is also a conceptual question, which includes a comparative element. Thus, it is natural to consider literature review as a methodology for answering these particular questions. Investigation of RQ3, on the other hand, requires us to work with numerical data obtained by experimenting. For this reason, we chose quantitative research, more precisely experimental design, as the method for answering RQ3.

3.1 Literature review

A literature review is a necessary part of research that helps to obtain relevant background information and build a solid foundation for later experiments. In their book, Gay et al. [GMA12] state that a literature review:

> *"...involves the systematic identification, location, and analysis of documents containing information related to the research problem, and demonstrates the underlying assumptions (i.e., propositions) behind the research questions that are central to the research proposal."*

In other words, a literature review assists us in acquiring information already known about the problem [Wie14]. Moreover, it explains to the reader why we choose specific hypotheses and research questions to research further. There are

a few different types of literature review, including traditional literature review and systematic literature review. In our project, we utilised the traditional, or narrative, literature review method. According to Coughlan et al. [CRC08], the primary purpose of a traditional literature review is to "provide the reader with a comprehensive background for understanding current knowledge and highlighting the significance of new research". As there is big volume of information that needs to be studied and summarised, we followed the five steps of literature review process defined in the article. The steps will be discussed in the following subsections.

3.1.1 Selecting a review topic

Identifying the subject of the research and literature review is an essential step in order to begin any project. Selection and further specification of the review topic were made in collaboration with the responsible professor and the supervisor during the first few weeks of the research. We established early on that there are many prior studies concerning the security aspect of IoT, together with some suggestions for how those confirmed security issues could be eliminated or made less severe. Moreover, plenty of paper about comparative research have been identified. Both CoAP alone and CoAP/DTLS have previously been tested, and the results published were a huge asset for our project. However, due to WireGuard being a relatively new protocol, there was a noticeable lack of academic articles and research available regarding the security and performance of the protocol. As far as we know, it has also never been tested for securing IoT communication. Therefore, one of the main questions of our research became the feasibility of WireGuard being a valuable protocol for securing IoT communication, as well as how it would be done and whether it would be beneficial. Before that, however, we decided to focus on the security of constrained IoT devices and how the known protocols used for securing them affect their performance.

3.1.2 Searching for literature

Once the review topic was defined, the next step was to find relevant literature that would help us gather information about IoT security, the protocols chosen for the research, and the possible implementation of WireGuard over another protocol. The web search engine Google Scholar[1] was used for accessing relevant academic research papers. Websites such as ResearchGates[2] and IEEE Xplore[3] were also frequently visited in order to find other studies related to our main topic, which Google Scholar might not have acquired yet. Moreover, YouTube was utilised for finding information regarding WireGuard, such as Jason Donenfeld's presentation of the protocol at the

[1] https://scholar.google.com/
[2] https://www.researchgate.net/
[3] https://ieeexplore.ieee.org

Black Hat conference in 2018[4]. During this phase, we have selected some search terms which gave the most accurate results related to our topics. These terms were often searched in combination with other relevant terms.

3.1.3 Gathering, reading and analysing the literature

This step included choosing the correct papers and dividing the ones chosen for our literature review into different categories, such as "IoT security", "CoAP/DTLS", "WireGuard", and "Performance analysis". The abstracts, conclusions and keywords indicated by the authors of the papers were helpful when deciding whether a paper was genuinely relevant and would contribute to our research. After the papers were grouped, we read through them with greater attention to detail. At this stage, we took notes on what we found most significant for both our background chapter research and the actual implementation and experiment. These notes included both information taken directly from the papers and our assessment on the relevance of the said information.

3.1.4 Writing the review

Chapter 2 presents and highlights the most relevant elements of the literature research described in the preceding subsections. These elements are categorised under appropriate sections and support the Research Questions and Hypotheses investigated in the later chapters.

3.1.5 References

During the literature review, all the references were gathered and stored, grouped by the same categories as mentioned in the *Gathering, reading and analysing the literature* subsection. The *References* section at the end of this thesis includes all the relevant references used during our research.

3.2 Quantitative research

Once the literature review phase was completed, and enough background information was obtained to build a solid foundation for answering RQ1 and RQ2, we could move on to the next phase of our study. Quantitative research is concerned with collecting and analysing numerical data. Inspired by papers such as [LN21] and [KBTM21], we wanted to collect performance metrics by running the protocols predetermined amount of times and in a fixed environment. Moreover, we wanted to check how changes in the network would reflect on the values we obtained prior to making the said changes.

[4]https://www.youtube.com/watch?v=88GyLoZbDNw&ab_channel=BlackHat

Similar to other types of research, quantitative research has both advantages and disadvantages. Since the methods for data collection in quantitative research are usually standardised and regulated, collecting the data and observing the outcomes is simple to replicate. Moreover, the same experiments can be conducted anywhere and with more or completely different objects of interest. The results can then be directly compared to the previous outcomes, and the effect the environment or new participants have on the results can easily be analysed. Quantitative research allows to analyse extensive data from the samples collected from the experiments, making it possible to use this analysis to confirm or deny hypotheses established beforehand. Hypothesis testing in quantitative research is formalised and follows specific procedures, and all the parts of the testing are carefully considered before any conclusion is drawn [Bha20].

However, when it is predetermined which values should be measured and what methods should be used, it is easier to ignore other potentially important and meaningful observations which could benefit the research. Furthermore, the values presented as the outcome of quantitative research do not explain anything behind them, making them superficial. In such cases, it is beneficial to utilise qualitative research to elaborate on the meaning behind the results. Additionally, quantitative research suffers from the lack of context since it regularly uses unnatural settings such as laboratories for conducting the experiments or fails to consider other contexts which might affect the results. Finally, quantitative research can be negatively impacted by structural biases such as inappropriate sampling methods, such as if some specific objects of interest are chosen more often than others.

Quantitative research uses three main methods based on different goals researchers might have: descriptive research, correlational research, and experimental research [Bha20]. For our study, we utilised the experimental research approach, also known as experimental design method.

3.2.1 Experimental design

Experimental design is a quantitative research method that is concerned with creating a standardised set of procedures to test hypotheses stated before the experimenting begins. These procedures assist us in collecting data which will either support or refute a hypothesis. The steps of experimental design [Bev19] that were followed during our research are:

1. Using literature review as our supporting research method, we defined the three Research Questions from Section 1.1, together with three testable Hypotheses related to RQ3;

2. The controlled environment used to conduct the experiments was carefully designed in order to measure the desired values. Our experimental treatment closely mimicked the treatment used in [LN21] with the intention to possibly validate our results by comparing them not only within themselves but also with external results obtained by other researchers. The environment is described in detail in Chapter 4;

3. After designing the controlled environment we determined the study size, which in our case included the number of requests sent by the client and the number of times the same scenario would be repeated for each protocol. A description of that can be found in Section 4.2;

4. The next step after designing the proper environment and determining the study size is experimenting. At this stage, a data set is created. It is vital that for the experiment itself to have been designed to simplify the entry process [RM17]. As shown in Section 4.1, our research focuses on a few specific metrics, which we manually entered into a table with a fixed number of rows and columns. The titles used closely resembled ones found in `WireShark` to facilitate the manual data transfer. The data was then imported to the analysis software;

5. The next step of quantitative research is analysing and interpreting the acquired data. In our research, we used descriptive statistics to represent the data, which is discussed in Section 3.3. The results for all the protocols are then compared to each other and to the results published in other papers such as [LN21].

6. The last step of the experimental design is discussing the viability of the hypotheses and answering the research questions. The discussion and conclusion can be found in Chapter 6.

During the experimental design phase, enough data should be collected to explore the hypotheses established at the beginning of the research. The end goal is to confirm or deny the hypotheses, which is ultimately valuable for answering the research questions.

3.3 Analysing quantitative data

Once the experiment has been conducted and the data has been collected and processed (step 5 of our experimental design explained in Section 3.2), it needs to be analysed and interpreted. According to Robson et al. [RM17], descriptive statistics - also called summary statistics - are used to represent important aspects of a data set by a single number. The results depicted in Section C.1 of Appendix are presented using descriptive statistics. The two aspects most commonly introduced

when discussing this method are the level of distribution and the spread of the distribution.

Measures of central tendency are used for summarising the level of distribution. Since the goal is to find a single number to represent the data, it is common to use the *average* value obtained by adding all the entries together and dividing the result by the number of entries. In academic research, the *average* is often referred to as the *arithmetic mean*. The network protocols investigated in our research we compared using the arithmetic mean measure. Hence, the results presented in Section 5 and Section C.1 are the arithmetic means calculated from the values obtained during our experiments.

The second aspect of descriptive research is the spread of the distribution. For this purpose, **measures of variability** were used. Measures of variability help distinguish how tightly clustered or spread the entries in a data set are. In our thesis, *standard deviation* was used to calculate the variance of the entries in our data set. The deviation is the difference between an individual entry and the arithmetic mean. A low standard deviation indicates that the entries in the set are close to the arithmetic mean, and a high standard deviation indicates that the entries are more spread out. All results in Chapter 5 and Section C.1 of Appendix are presented together with the standard deviation calculated for that particular set. Furthermore, all the results are graphically displayed using `Matplotlib`, a plotting library for Python. Presenting the data in a more clear and visual way helps when comparing the results based on the protocols.

3.3.1 Validity and reliability

Validity and reliability are concepts used to evaluate the quality of research. Validity is concerned about the accuracy of the research, focusing on how accurately a method measures what it is intended to measure. Reliability refers to the consistency of the research or evaluation and is concerned about the stability of the measure over time and with different observers [RM17]. When collecting various previous studies and analysing the evaluations presented there, a researcher interprets the meaning of these evaluations. This interpretation can be subjective and thus affect the validity and reliability of the research produces. Moreover, inadequate evaluation and interpretation can negatively impact the validity of inferences derived from the review, such as further experimental design and measurements. Thus, the researcher needs to focus on objectivity during the process of literature review in order to obtain validity and reliability [Del05].

The literature review in this study was conducted with the two concepts in mind. In order to answer our RQs, we collected various papers focusing on different aspects of our primary research topics (Section 3.1.3). However, there is a possibility that

some relevant research may not be included in our research, as relevant papers may have been overlooked in the initial stages of literature reviews. Moreover, the studies that have been deemed relevant may not be completely valid and reliable. In addition, more papers with newly conducted research may have been published without being noticed by us.

When conducting quantitative research, the validity and reliability of the methods and measurements also need to be taken into account. Four main types of validity that need to be considered are introduced by Middleton in [Mid20] and are closely echoed by Heale et al. in [HT15].

Similar to the definition mentioned above of validity, **construct validity** refers to how accurately a measurement tool represents what a study intends to measure. To ensure construct validity, we need to make sure that our indicators and measurements are developed based on relevant existing knowledge. Our measurements and attributes were closely based on the experiments performed by other researchers and described in the papers gather during the literature review phase.

Content validity is concerned with the measurement method covering all relevant aspects of the subject it aims to measure. As our goal was to measure the network performance of the CoAP implementations, we constructed our experiments in such a way that all relevant attributes related to performance speed and overhead would be included in the final results. Even though we concentrated on the four main attributes during the discussion part, all of the measured attributes can still be observed in Appendix C and server as supporting attributes for our eventual conclusion.

Face validity refers to how suitable the experiment content seems to be on the surface. It is similar to content validity but is considered a more informal type of validity and is often used in the initial phase of developing the research method.

Criterion validity accesses how closely the results of the experiments correspond to the results of a different experiment. After conducting our experiments, we could compare our results to prior research published in other research papers. Although no research has yet compared WireGuard to CoAP and CoAP/DTLS, various similar experiments have been conducted to measure the performance of the latter two protocols. When comparing our results to their results, a high degree of correlation has been observed, indicating that our experiments indeed measure what they intended to measure.

The reliability of our experiments can be judged by whether the same result can consistently be achieved by using the same methods under the same circumstances. The results obtained during our experiments were presented in the form of the

arithmetic mean and standard deviation. The entries in the tables presented in Appendix C indicate an overall low standard deviation, meaning that when running the same experiment multiple times, the results gathered were consistent and can thus be deemed reliable. One particular case of an uncharacteristically high entry is discussed in detail in Chapter 6.

This chapter presented the methodology used for our research, which consisted of Quantitative research using experimental design combined with a literature review. The data collected in the course of the research were analysed and interpreted using descriptive statistics. Validity and reliability of the research methods were also discussed. Chapter 5 introduces the results obtained during the experimental phase, and Chapter 6 focuses on step six of the experimental design, as it discusses the hypotheses and attempts to answer the research questions established in Chapter 1.

Chapter 4
Experimental setup

This chapter introduces the technical side of the study and includes the description of the testbed setup used during our research. The methodology for the experiments and data collection is described in Chapter 3. Furthermore, this chapter explains the changes made in the CoAP implementations and presents the final results, which will be analysed in Chapter 6.

4.1 Testbed setup

For conducting the study, an experimental testbed setup was designed. The main goal of the experiment was to be able to send CoAP packets through an encrypted tunnel secured by WireGuard. Similarly to the testbed presented in [LN21], our setup was limited to a single-hop scenario running in a network interface set up between a laptop and a virtual machine. To test CoAP secured by WireGuard, we set up an encrypted WireGuard interface between the peers and bound the server IP-address to the interface when needed. The setup can be seen depicted in Figure 4.1.

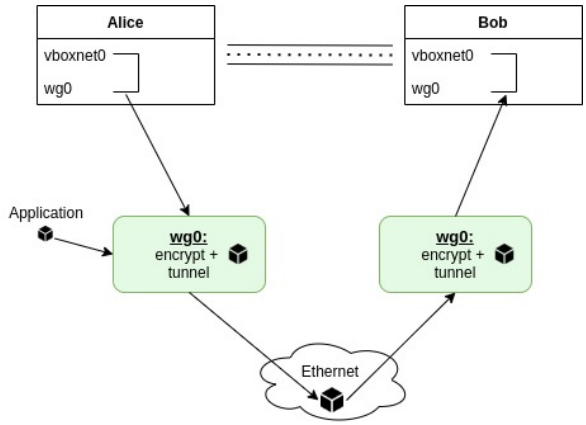


Figure 4.1: The experimental setup

The testbed consisted of two nodes, Alice and Bob. Depending on the experiments, the nodes were tested both as a client and a server. In our setup, Alice is an Ubuntu 20.04 virtual machine installed using the VirtualBox 6.1.16 application. The virtual machine was installed on the laptop, which in our setup is Bob. Bob is a Huawei Matebook X Pro 2020 laptop running Ubuntu 20.04 with Intel® Core i5-10210U processor and 16GB of RAM. Alice was connected to Bob via a host-only adapter, creating our own virtual interface *vboxnet0*. Both Alice and Bob were able to run the non-secured version of CoAP, CoAP with DTLS and CoAP with WireGuard, as well as plain WireGuard. The CoAP clients and servers ran Eclipse Californium[1] 3.0.0-M1 for the non-secured version and Eclipse Scandium 3.0.0-M1 for CoAP/DTLS[2]. Both implementations are developed in Java, and the changes made to the original code were minimal. The said changes will be further explained in Section 4.1.2.

4.1.1 WireGuard

To set up an encrypted WireGuard tunnel between two peers, both need to have a WireGuard interface and know each other's public keys. To install and configure WireGuard on Ubuntu, we used the guide provided by the WireGuard creators, which can be found on the official website[3]. Moreover, "Ubuntu 20.04 set up WireGuard VPN server"[4] guide published by Vivek Gite was used during the process of setting up the protocol. After installing the protocol, both Alice and Bob needed to generate base64-encoded public and private keys, as required by WireGuard. This wa done by running the following command:

```
1    wg genkey | tee privatekey | wg pubkey > publickey
```

Once the keys were generated, two configuration files were created, one for Alice and Bob, respectively. In the configuration files, we assigned a specific IP address to each node on the WireGuard's `wg0` interface. A port that would be used to listen for incoming WireGuard traffic was also added. In the same configuration file, we created a new peer using their public keys, `wg0` IP addresses and endpoint IP addresses. The configuration files for both nodes can be seen in Section A.1.1 of the Appendix.

Once the configuration files were created, the interfaces could be set up by running `$ wg-quick up wg0`. To get a detailed description of the interfaces that were up, the `$ wg` command was used (`$ sudo wg` when not running as a root). The examples of the output of these two commands can be seen in Figure 4.2. To check whether the nodes have successfully been connected, we pinged back and forth using both `wg0` and endpoint `vboxnet0` IP addresses. Pinging a `vboxnet0` IP address makes

[1] https://github.com/eclipse/californium
[2] https://github.com/eclipse/californium/tree/master/scandium-core
[3] https://www.wireguard.com/install/, https://www.wireguard.com/quickstart/
[4] https://www.cyberciti.biz/faq/ubuntu-20-04-set-up-wireguard-vpn-server/

```
root@osboxes:/etc/wireguard# wg-quick up wg0
[#] ip link add wg0 type wireguard
[#] wg setconf wg0 /dev/fd/63
[#] ip -4 address add 10.1.0.1/24 dev wg0
[#] ip link set mtu 1420 up dev wg0
root@osboxes:/etc/wireguard# wg
interface: wg0
  public key: vwjIhUrppqeBTsBepw347Fc7U4yu7ezTeZ5Vy3yMj1w=
  private key: (hidden)
  listening port: 41194

peer: e7HjfGJtGtpLKzJpCses1AfMPE9iYlOf2S64ztWxWVE=
  endpoint: 192.168.56.1:41194
  allowed ips: 10.1.0.0/24
```

Figure 4.2: Setting up *wg0* on Alice.

the traffic go through the standard network interface, which for this experiment is `vboxnet0`. Pinging a `wg0` IP address forces the traffic to go through the `wg0` interface instead. Once all the pings could reach the intended destination, the `wg0` setup was complete.

4.1.2 CoAP

As previously mentioned, Eclipse Californium and Scandium were used for running CoAP and CoAP/DTLS respectively. To achieve that, we cloned the repository and used Maven, a software project management and comprehension tool[5], to build the project. To edit the code and be able to run without using the command line, we also used Eclipse IDE[6]. Both Alice and Bob had Eclipse and Maven installed in order for the setups to be as similar as possible. Moreover, the folder containing the built Californium and Scandium projects was shared between Alice and Bob to ensure that the code run from both nodes would be identical.

To run the server and the client and connect them via the non-secured version of CoAP, we used the following commands (for the server and the client, respectively):

```
1    java -jar cf-helloworld-server-3.0.0-SNAPSHOT.jar
2        HelloWorldServer
3
4    java -jar cf-helloworld-client-3.0.0-SNAPSHOT.jar
5        GETClient coap://192.168.56.x
```

The IP address of the recipient depended on whether Alice or Bob was the server. Running the `HelloWorldServer` command created a server bound to all the available interfaces on the node, including `wg0` and `vboxnet0`. The port on which the server would listen for traffic or from which it would send traffic is a default CoAP port,

[5]https://maven.apache.org/
[6]https://www.eclipse.org/downloads/

`5683`. The partial output from running the server command can be seen in Figure A.1 in Section A.1.2 of the Appendix.

Since our goal was to imitate CoAP traffic between constrained devices, the code for both server and client was intentionally made to be as simple as possible. By making the nodes transmit a small payload, the packet size was deliberately minimised as well. Again, this was done so our traffic would resemble how the real-world IoT traffic would typically behave. The only resource implemented on the server was a "Hello World!" resource. Thus, the client could send a GET request to the server to access the resource, and if the request were successful, the server would respond with "Hello World!". The server code has not been changed and can be found in the Californium repository on github[7]. For the client side, a very basic `while`-loop was added in order for the client to repeat the request 150 times in a row. Otherwise, the code for the client can be found in the GitHub repository as well[8].

4.1.3 CoAP/DTLS

A similar setup was created for CoAP secured by DTLS. For this part of the experiment, however, some more modifications of the code had to be done. In addition to adding the same `while`-loop as for the non-secured CoAP client, modifying the server URI was possible to send a GET request directly to the needed IP address and port. It was done by binding the server to the desired interface (`vboxnet0`), thus binding it to the desired IP address. The code changed can be seen below.

```
1         DtlsConnectorConfig.Builder builder =
2             new DtlsConnectorConfig.Builder();
3         CredentialsUtil.setupCid(args, builder);
4
5         NetworkInterface nif = NetworkInterface.getByName("vboxnet0");
6         Enumeration<InetAddress> nifAddr = nif.getInetAddresses();
7         InetAddress a = null;
8
9         NetworkConfig config = NetworkConfig.getStandard();
10            while (nifAddr.hasMoreElements()) {
11                InetAddress b = nifAddr.nextElement();
12                if (a instanceof Inet4Address) {
13                    a = b;
14                }
15        InetSocketAddress bindToAddress =
16            new InetSocketAddress(a, \gls{dtls}_PORT);
17        builder.setAddress(bindToAddress);
```

[7] https://github.com/eclipse/californium/blob/master/demo-apps/cf-helloworld-server/src/main/java/org/eclipse/californium/examples/HelloWorldServer.java

[8] https://github.com/eclipse/californium/blob/master/demo-apps/cf-helloworld-client/src/main/java/org/eclipse/californium/examples/GETClient.java

The original code can also be seen on Scandium Github repository[9]. If Bob wanted to send a request to Alice, the SERVER_URI would be the following:

```
1    private static final String
2        SERVER_URI = "coaps://192.168.56.1:5684/Hello";
```

In our experiment, the default Scandium DTLS cipher suit was used, namely TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256. The full list of cipher suits provided by the Scandium implementation can be found on github[10].

One of the main attributes we wanted to measure for both DTLS and WireGuard was the handshake. In our experiment, the client and the server established a handshake each time the code was run. However, as we briefly discussed in Section 2.2.2, DTLS handshake is very expensive and is not performed every time a client wants to send something to the server. Moreover, our connection would technically not be broken if the client was started from the same communication endpoint (port) each time. For this reason, we tested the CoAP/DTLS implementation a few separate times, running it over a few hours, as well as overnight. This was done using the `Thread.sleep()` function in Java. The effect it had on the results will be discussed in Chapter 6.

The original code for Scandium CoAP/DTLS code can be found on GitHub[11]. For this part, all the codes were run in Eclipse IDE 4.19.

4.1.4 CoAP/WireGuard

For the purpose of our experiment, we used the non-secured CoAP server and client implementations to connect CoAP and WireGuard. Once the setup from 4.1.2 was complete, the server automatically started on the `wg0` interface as well, making it easy for the client to send GET requests to that particular IP address. We ran the following command to start the client:

```
1    java -jar cf-helloworld-client-3.0.0-SNAPSHOT.jar
2        GETClient coap://10.1.0.x
```

Again, the IP address the request was sent to was chosen depending on whether Alice or Bob acted as the server. As we indicated the endpoint IP addresses to be the `vboxnet0` addresses, the IP addresses from the two network interfaces - `wg0`

[9] https://github.com/eclipse/californium/blob/master/demo-apps/cf-secure/src/main/java/org/eclipse/californium/examples/SecureServer.java

[10] https://github.com/eclipse/californium/blob/master/scandium-core/src/main/java/org/eclipse/californium/scandium/dtls/cipher/CipherSuite.java

[11] https://github.com/eclipse/californium/blob/master/demo-apps/cf-secure/src/main/java/org/eclipse/californium/examples/SecureClient.java

and `vboxnet0` - are connected, which can be seen in Figure 4.1. When binding the CoAP server to the `wg0` IP address, it forced the traffic to go through the encrypted network tunnel secured by WireGuard. This partially answers our RQ3, which was the following:

> **RQ3:** Could WireGuard potentially be used instead of DTLS for CoAP as a protocol for securing communication? If yes, how?

Based on our experiments with setting up a working implementation of the CoAP/WireGuard protocol, we can conclude that it is, indeed, possible to use WireGuard for securing CoAP. We will further elaborate and discuss RQ3 in Chapter 6, together with the rest of the RQs and Hypotheses.

4.2 Data collection and analysis

As we presented in Section 4.1, our experiment consisted of running three different implementations of CoAP: non-secured CoAP, CoAP/DTLS and CoAP/WireGuard. These were run six times, three times with Alice as the server and three times with Bob as the server. The client sent 150 GET requests to the server each time the code was run. Taking inspiration from Laaroussi et al. [LN21], we ran the experiments once with no network delay and once with a fixed network delay of 60 ms, resulting in twelve experiments per implementation. We did it not only to assess the impact of the delay for our implementations but also to compare our results with the results published in [LN21]. We found it interesting to investigate whether our results could be validated by the results from the paper, given the similarities between the two setups. Thus, some of the attributes we chose are similar to the attributes seen in [LN21]. Another paper that has influenced our choice of attributes is the paper by Kondoro et al. [KBTM21], as it presents a similar comparative study of CoAP/DTLS, comparing it to MQTT/TLS 1.3 and XMPP/TLS 1.3. The attributes we chose to use for comparing the data we collected during our experiment are introduced and discussed in Chapter 5.

The tables with the detailed results can be found in Section C.1 of Appendix A. The results seen in the table are the arithmetic means of all obtained data sets for all the attributes. However, as established in Chapter 2, the main attributes and features we wish to investigate in order to be able to answer our RQs and Hypotheses are the latency, the RTT with and without the handshakes, the handshake time and the throughput. Thus, the tables and graphs used in the following chapters will only include those four primary attributes.

To conduct the performance analysis on the data obtained after running the experiments, several tools were utilised. `tshark`[12], a network protocol analyser, was used every time the client and the server communicated with each other to capture the traffic between them. The output of each captured conversation was saved in a `pcapng`. An example of a command for capturing the traffic for the CoAP/WireGuard implementation can be seen below:

```
tshark -f 'udp port 41194' -i any -w bob_client_wg_1.pcap
```

Once all the necessary `pcapng` files were obtained, we used the following `tshark` command to extract the round-trip times (RTTs) from all the files:

```
for i in *; do tshark -r "$i" -T fields
    -e frame.time_delta_displayed  > "$i".csv; done
```

This allowed us to quickly collect the RTTs for further investigation. In order to compare the RTTs for the different protocols gathered during the experiments, we wrote a short Python script utilising the `pandas`[13] library. The built-in functions were used to calculate the arithmetic mean of all collected RTTs and the standard deviation. The final code can be seen below:

```
import pandas as pd
import glob, os

os.chdir("/home/evgenia/Desktop/experiments_no delay")
    for file in glob.glob("*.csv"):
    data = pd.read_csv(file)
    average = data.mean()
    df = pd.DataFrame(data)
    dev = df.std()
    print('Name: ' + file, average, dev)
```

For evaluating other attributes such as latency, the number of packets per second, packet size and throughput, `Wireshark`[14] was used. The numbers used in further calculations were found by opening a `pcapng` file in `WireShark` and going to `Statistics -> Capture File Properties`. Afterwards, the built-in functions were used to calculate the mean values and the standard deviation in `Google Sheets`. To showcase some examples of the `wireShark` files captured and investigated, one `pcapng` file output per protocol can be found in Appendix B.

Lastly, to imitate the 60 ms fixed network delay a tool called `tc`[15] was utilised. `tc` is a Traffic Control tool for Linux kernel used to emulate network conditions, such

[12] https://www.wireshark.org/docs/man-pages/tshark.html
[13] https://pandas.pydata.org/
[14] https://www.wireshark.org/
[15] https://man7.org/linux/man-pages/man8/tc.8.html

as delay packet loss and duplication. The commands used to add and delete rules to manipulate the `vboxnet0` network interface were the following:

```
1    tc qdisc add dev vboxnet0 root netem delay 60ms
2
3    tc qdisc del dev vboxnet0 root netem delay 60ms
```

This chapter introduced the experimental setup used for conducting the comparative research and discussed the practical aspect of how the data was collected and analysed. The results of the experiments will further be presented in Chapter 5.

Chapter 5
Findings

This chapter introduces the attributes used to compare the CoAP/DTLS and CoAP/WireGuard implementations and presents the results based on the said attributes. The results are expressed by calculating the arithmetic means of the data sets consisting of the attributes and the standard deviation. The results presented in this chapter will serve as a foundation for discussing and answering Research Question and Hypotheses established in Chapter 1.

In order to further compare the two protocols and be able to answer our RQs, we chose several performance attributes, also called metrics, to measure during our experiments. In Section 4.2 of Chapter 4 we introduced the papers by Laarroussi et al. [LN21] and Kondoro et al. [KBTM21] that greatly inspired our choice of metric for this study. As we have observed in Chapter 2, there is a significant difference between DTLS and WireGuard handshake. For this reason, we chose *handshake time* - the overall time it takes to establish a connection - to be one of the metrics. An attribute related to the handshake that we will include in our evaluation is *packet overhead*. *Packet overhead*, both in the form of the number of packets and their message length, is introduced by extra packet data such as security handshakes, as well as extra headers [KBTM21]. To be able to see how the handshake affects the overall performance, we also want to measure *RTT* and *latency*.

RTT is the time it takes for the client to send one request plus the time it takes to receive the acknowledgement back from the server. IoT devices are typically small, constrained devices often responsible for communicating vital information needed to be conveyed fast. Devices used in the medical field are one example of that, and a short time span and RTT are seen as a significant advantage, making them appropriate attributes for analysis. *Latency* means the total amount of time between the initial request is sent and the sender receives the final response [KBTM21]. For this experiment, it was measured as a total RTT, which is the same as the total time span. For these metrics, we will present the result with and without the handshakes.

The last attribute we will examine in our research is *throughput*, which is the amount of traffic between a source and a destination measured over a period of time. This particular attribute was measured in two ways: in Mbits/s and the number of packets per second (pps). The sections below present both results with and without the handshake, which will undoubtedly affect the throughput.

Although *power consumption* was initially among the attributes we planned to examine, we eventually defined the scope of evaluation in terms of network performance metrics. The main reason for that was the discovery of the paper by Laaroussi et al. [LN21]. In their study, OSCORE, a new method to provide end-to-end security for CoAP, is compared with CoAP and CoAP/DTLS. The main objective of the papers was to test the efficiency of the new protocol compared to well-known versions of CoAP. Moreover, our plan from the beginning was to use Californium and Scandium for implementing CoAP and CoAP/DTLS. Altogether, this caused a change of the direction of the research, as we decided to focus on the network behaviour of the protocols rather than the physical impact running the protocols has on IoT devices.

Thus, the main focus of our experiments will be on four of the attributes mentioned above: handshake time, and latency, RTT and throughput with and without the handshakes. The rest of the chapter covers the results of the experiments based on these four attributes and gives a summary comparing the CoAP/DTLS and CoAP/WireGuard against each other and the non-secured version of CoAP.

5.1 Handshake time

As addressed previously, both DTLS and WireGuard use a handshake to establish the communication between the client and the server. The DTLS handshake is six flights (Figure 2.5), while WireGuard handshake is only two flights (Figure 2.8). Both protocols can essentially be considered security extensions for CoAP. These security extensions add *packet overhead* to the non-secured CoAP protocol. In Section 2.1.1, we established the importance of securing IoT communication. However, it is still crucial to attempt to minimise the metrics such as *packet overhead* with regards to constrained devices, and it increases the volume of the traffic.

For this experiment, we measured the handshakes for traffic with no network delay and with a delay of 60 ms. Table 5.1 shows the results and provides a comparison of the latency separated into protocol families, which will be the case throughout the rest of the analysis. The non-secured version of CoAP does not include establishing a connection with the help of a handshake; thus, the table entry for CoAP is empty.

In terms of the total number of additional packets, the packet overhead DTLS

	CoAP	CoAP/DTLS	CoAP/WireGuard
No network delay			
Handshake, ms	–	218 ± 212	1 ± 0.412
60 ms network delay			
Handshake, ms	–	407 ± 112	31 ± 30

Table 5.1: Measured handshake time expressed by the mean of the results obtained when running the experiment.

introduces to CoAP is six, while WireGuard only adds two additional packets. Furthermore, as seen in the Figure 5.1, the total amount of bytes exchanged during a CoAP/DTLS handshake equals 1573 bytes (12,584 bits). On the other hand, the size of the WireGuard handshake is 328 bytes (2,624 bits). Figure 5.1 compares the two fastest handshakes achieved during our experiment for CoAP/DTLS and CoAP/WireGuard, respectively.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.56.1	192.168.56.103	DTLSv1.2	191	Client Hello
2	0.001123645	192.168.56.103	192.168.56.1	DTLSv1.2	104	Hello Verify Request
3	0.003358592	192.168.56.1	192.168.56.103	DTLSv1.2	223	Client Hello
4	0.007106562	192.168.56.103	192.168.56.1	DTLSv1.2	507	Server Hello, Certificate, Server Key Exchange, Certificate Request, Server Hello Done
5	0.062745498	192.168.56.1	192.168.56.103	DTLSv1.2	429	Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Handshake Message
6	0.081405683	192.168.56.103	192.168.56.1	DTLSv1.2	119	Change Cipher Spec, Encrypted Handshake Message
7	0.097547282	192.168.56.1	192.168.56.103	DTLSv1.2	99	Application Data
8	0.108523404	192.168.56.103	192.168.56.1	DTLSv1.2	107	Application Data

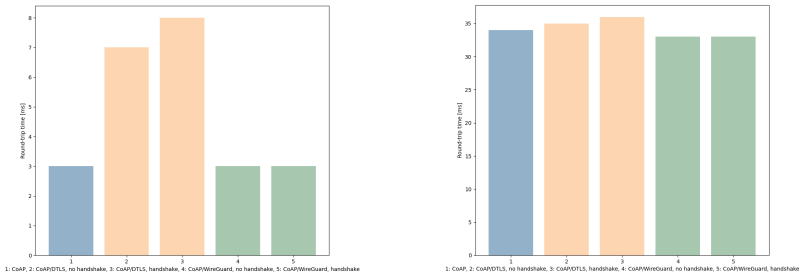
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.56.1	192.168.56.103	WireGuard	192	Handshake Initiation, sender=0x478635E2
2	0.000527015	192.168.56.103	192.168.56.1	WireGuard	136	Handshake Response, sender=0x5CB8CB1A, receiver=0x478635E2
3	0.033799665	192.168.56.103	192.168.56.1	WireGuard	124	Transport Data, receiver=0x478635E2, counter=0, datalen=48
4	0.036695091	192.168.56.1	192.168.56.103	WireGuard	588	Transport Data, receiver=0x5CB8CB1A, counter=1, datalen=512

Figure 5.1: Handshake + one RTT for GET request and one response.

It takes almost three times longer for CoAP/DTLS to establish a handshake compared to CoAP/WireGuard. Moreover, for this exchange, the DTLS handshake takes almost 75% of the processing time (similar to the results depicted in [KBTM21]). On the contrary, the WireGuard security handshake takes up only 1.436% of the whole processing time. To summarise, during our experiment, the following outcomes could be observed:

- Average handshake for CoAP/WireGuard is 218 faster than an average handshake for CoAP/DTLS without delay in the network, and approximately 13 (13.129) times faster for CoAP/WireGuard when a fixed delay of 60 ms is introduced.

- CoAP/WireGuard processing time is almost three times (2.957) less than CoAP/DTLS.



(a) No network delay

(b) 60 ms network delay

Figure 5.2: Measured RTT for CoAP, CoAP/DTLS (no handshake), CoAP/DTLS (handshake), CoAP/WireGuard (no handshake) and CoAP/WireGuard (handshake) respectively.

- DTLS handshake takes almost 75% (75.012%) of the whole processing time (handshake + one RTT for GET request and response), while WireGuard handshake only takes 1.436% of the processing time (for a network with no delay).

5.2 Round-trip time (RTT)

Table 5.2 and Figure 5.2 depict average RTTs for all the tested protocols and includes results with and without the DTLS and WireGuard handshake.

	CoAP	CoAP/DTLS	CoAP/WireGuard
No network delay			
RTT w/ handshake, ms	3 ± 0.210	8 ± 2.700	3 ± 0.419
RTT no handshake, ms	3 ± 0.210	7 ± 2.100	3 ± 0.157
60 ms network delay			
RTT w/ handshake, ms	34 ± 1	36 ± 0	33.680 ± 1
RTT no handshake, ms	34 ± 1	35 ± 0	33.870 ± 1

Table 5.2: Measured RTT expressed by the mean of the results obtained when running the experiment.

We can observe that the results are consistent with the results from Section 5.1. The main conclusions possible to draw from the outcome are summarised below:

- In our experiment, the average RTT was not particularly affected by adding a handshake. It is true for both CoAP/DTLS and CoAP/WireGuard and applies to the results obtained with and without a network delay.
- Again, the difference between the protocol becomes less the more delay is added, which is consistent with the findings reported by Laaaroussi et al. [LN21].
- Using WireGuard to secure CoAP adds no overhead looking from the perspective of average RTT. Furthermore, the average RTT for CoAP/WireGuard is measured more than half as much for CoAP/DTLS.

5.3 Latency

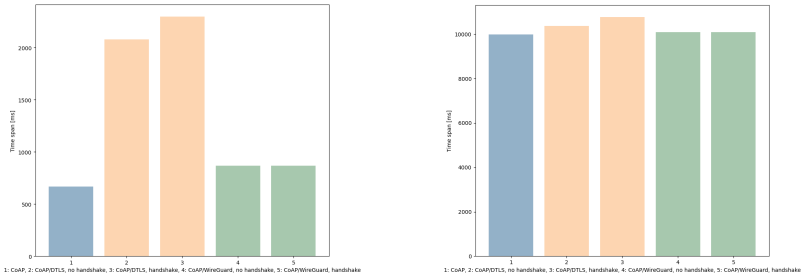
Table 5.3 demonstrates the experimental results for all three protocols. The results for DTLS and WireGuard are presented both with and without the handshakes. A more graphical comparison can be seen in Figure 5.3.

	CoAP	CoAP/DTLS	CoAP/WireGuard
No network delay			
Time span, handshake, ms	666 ± 63	$2,295 \pm 822$	865 ± 126
Time span, no handshake, ms	666 ± 63	$2,078 \pm 617$	865 ± 126
60 ms network delay			
Time span, handshake, ms	$9,983 \pm 205$	$10,773 \pm 96$	$10,103 \pm 250$
Time span, no handshake, ms	$9,983 \pm 205$	$10,367 \pm 106$	$10,092 \pm 242$

Table 5.3: Latency expressed by the mean of the results obtained when running the experiment.

A summary of the most important points from Table 5.3 and Figure 5.3 would be the following:

- As we could observe in the previous section, the footprint of the WireGuard handshake is so small that it does not have any significant influence on the overall time span.
- There is a slight decrease in the average time span when removing the handshake from the CoAP/DTLS implementation: the average time span without the handshake is 10.442% less than with the handshake in a network with no delay and 3.768% less for a network with a fixed 60 ms delay.
- Similar results can be observed when a 60 ms delay is introduced: there is only a very slight decrease in the average time span for CoAP/DTLS (3.916%) and CoAP/WireGuard (0.109%) when removing the handshake.



(a) No network delay

(b) 60 ms network delay

Figure 5.3: Measured latency for CoAP, CoAP/DTLS (no handshake), CoAP/DTLS (handshake), CoAP/WireGuard (no handshake) and CoAP/WireGuard (handshake) respectively.

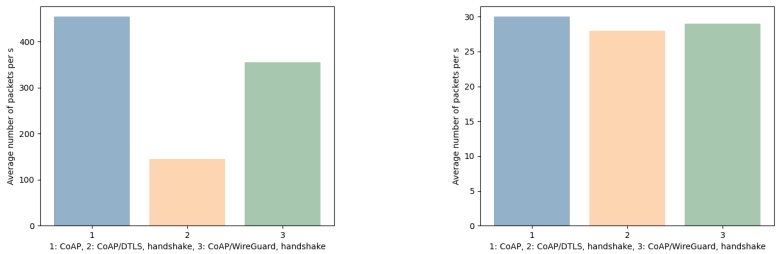
- Compared to non-secure CoAP, adding WireGuard increases the overall time span by 29.880% while securing CoAP with DTLS will increase the overall time span by 244.595% with the handshake and 212.012% without (for a network with no delay).
- Similarly to Laaroussi et al. [LN21], we observed that an introduction of a delay in the network yielded a smaller difference between the protocols.

5.4 Throughput

Finally, we measured average throughput for the three protocols, testing it while having no network delay and adding a delay afterwards to compare the results. Figure 5.4, Figure 5.5 and Table 5.4 present an extensive comparison of the throughput evaluation.

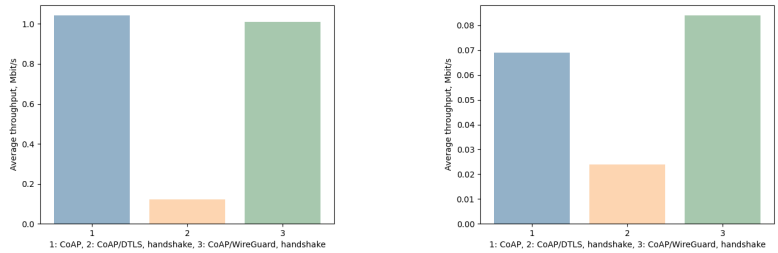
	CoAP	CoAP/DTLS	CoAP/WireGuard
No network delay			
Packets per second	454 ± 41.617	145.584 ± 35.247	355 ± 47.527
Throughput, Mbit/s	1.042 ± 0.092	0.123 ± 0.030	1.009 ± 0.136
60 ms network delay			
Packets per second	30.050 ± 0.608	28.417 ± 0.267	29.800 ± 0.792
Throughput, Mbit/s	0.069 ± 0.001	0.024 ± 0	0.084 ± 0.002

Table 5.4: Measured throughput expressed by the mean of the results obtained when running the experiment.



(a) No network delay (b) 60 ms network delay

Figure 5.4: Measured number of packets per second for CoAP, CoAP/DTLS (handshake) and CoAP/WireGuard (handshake) respectively.



(a) No network delay (b) 60 ms network delay

Figure 5.5: Measured throughput in Mbit/s for CoAP, CoAP/DTLS (handshake) and CoAP/WireGuard (handshake) respectively.

To summarise the results presented in this section, we can highlight some of the more significant results observed in our experiment:

- As expected, both CoAP/DTLS and CoAP/WireGuard have a lower throughput than non-secured CoAP in almost all cases. However, the pps ratio for CoAP secured by WireGuard is almost 2.5 times higher than the pps for CoAP secured using CoAP (for a network with no delay).
- The measured results for packets per second in a network with a fixed 60 ms delay exhibit little to no difference;
- Looking at the throughput in terms of Mbit/s, CoAP/WireGuard has only around a 3% decrease from the throughput recorded for non-secured CoAP. The decrease for CoAP/DTLS compared to CoAP is around 88% (for a network with no delay).

- Despite the results for the number of pps being similar for all the three protocols in a network with no delay, some significant differences can be observed for the throughput compared in Mbit/s. When securing CoAP with DTLS, the throughput went down by 65.220%. However, when using WireGuard instead of CoAP, the throughput increased by 21.740%. This will be discussed more thoroughly in Chapter 6.

This chapter introduced the main attributes used for comparing CoAP and its two secure implementations. Each subsection presented the results related to one of the attributes and summarised the main findings. The results presented in this chapter will serve as a foundation for further discussion in Chapter 6, where we will also address the Research Question and Hypotheses established in Chapter 1.

Chapter 6
Discussion

This chapter will conclude the problem investigation phase of our thesis and discuss the results presented in Chapter 5 more closely. Based on the results, we will answer our RQs defined in Chapter 1. A set of hypotheses established to investigate RQ3 in further detail will also be explored in this chapter.

6.1 Main security concern for IoT devices using CoAP

When aiming to find a protocol best-suited for securing any communication, it is important to examine both the security and performance requirements of the nodes responsible for carrying out the communication. A CNN consisting of constrained devices such as IoT devices has its own set of requirements, and our research focused on summarising these requirements in connection to the CIA triad. Our RQ1 as defined in Section 1.2 of Chapter 1 is as follows:

> **RQ1:** What are the main security concerns regarding confidentiality, integrity, and authentication (the CIA triad) when it comes to communication between IoT devices using CoAP?

Reflecting upon the security concerns explored in Section 2.1.1, we can conclude why the three principles are vital for achieving a secure communication framework [MYAZ15]:

Confidentiality: Given the sheer amount of data being collected by constrained devices, the data must be secure and only available to authorised users. In an IoT network, entities such as humans, different machines and services can all be users. Furthermore, users can include internal devices (devices that are part of the network) and external devices (which are not part of the network). Thus, it is vital to ensure that information is not shared with the nodes which are not authorised to obtain it.

There are multiple methods to potentially exploit confidentiality, e.g., performing attacks such as Replay Attack[1], eavesdropping or analysing the traffic.

Integrity: Due to the nature of IoT communication, where data is exchanged between a large number of different devices, it is essential to protect the information from being tampered with, either intentionally or unintentionally. The information shared should be accurate, complete, and come from the right sender. The usage of firewalls and protocols can help manage data traffic, but maintaining end-to-end security is vital for enforcing integrity in IoT communication. However, since constrained devices commonly have low computation power, the security at endpoints cannot be guaranteed. Integrity can be compromised by attacks such as jamming[2] and spoofing[3] attacks, as well as other types of unauthorised access.

Availability: The idea behind IoT is to connect numerous devices and provide data to the users whenever they need to access it. Hence, both IoT devices, services, and the data gathered must be reachable and available at any point in time. The most common type of attack to affect availability is DoS or DDoS.

Since CoAP is one of the most popular IoT protocols, these concerns are highly relevant for IoT devices utilising CoAP. CoAP itself does not have any authentication or authorisation mechanisms and does not provide confidentiality. Thus, there is a need for standard services such as TLS and DTLS.

It is possible to ensure confidentiality for CoAP by using one of the security modes and authentication methods provided by DTLS and explained in Section 2.3.1. However, as established in Section 2.3.2, CoAP is susceptible to different attacks, including message parsing and IP address spoofing. IP address spoofing can be used to launch a DDoS attack by forging valid IP addresses. The cookie exchange mechanism utilised by DTLS improves CoAP's resistance to various types of DDoS attacks. It is not, however, immune to attacks from valid IP addresses. Thus, DTLS does not eliminate the possibility for exploitation of IP address spoofing, which negatively affects the integrity and availability of the data being sent.

While DTLS is capable of providing some level of security and is widely used for securing CoAP communication, previous studies have uncovered vulnerabilities that negatively affect the protocol's security. With IoT devices being used for various purposes, the importance of implementing additional countermeasures to secure the protocol is evident. However, additional security features might increase the

[1] *Replay Attack* is a type of security attack where a third, unauthorised party captures the traffic and can send it to its original destination, acting as the original sender.

[2] *Jamming* is a kind of Denial of Service attack, where an attacker prevents other nodes from using a communication channel by completely occupying it.

[3] *Spoofing* is a situation when an attacker successfully masquerades as a legitimate entity to deceive the receiver.

complexity of the protocol even further, which is not desirable as some already consider DTLS too heavyweight for CoAP. There is, therefore, demand for more lightweight solutions that can match or improve the security level provided by DTLS.

6.2 Comparing WireGuard and DTLS

Studying DTLS and WireGuard during the literature review phase and presenting the central elements of the protocols in Chapter 2 gave us a foundation to answer RQ2. As defined in Section 1.2 of Chapter 1:

> **RQ2:** What are the differences and similarities between WireGuard and DTLS with regards to features and how they operate?

When considering replacing one protocol with another, it is important to highlight the main differences similarities between them. A concise summary, together with the evaluated experimental results, will assist us when deciding if replacing DTLS with WireGuard can be advantageous. A comparison of some of the core features introduced in Section 2.3 and Section 2.4 of Chapter 2 can be seen below:

- Both DTLS and WireGuard run on UDP, a feature that makes them convenient candidates for securing IoT communication. While DTLS was designed to secure the transport layer of the network stack, WireGuard exclusively secures the network layer.
- Both protocols utilise a cookie-based DoS mitigation technique and ensure anti-replay protection.
- Unlike WireGuard, DTLS offers cryptographic agility. As previously discussed, cryptographic agility allows users to tailor several different cipher suites to the network's and devices' demands. Nir et al. [NSS] provide an extensive list of available cipher suites for TLS and DTLS (up to version 1.3). WireGuard is not cryptographically agile, meaning that there is no possibility to choose other algorithms besides the ones provided by the protocol.
- As seen in Figure 2.5 and Figure 2.8, DTLS connection initiation requires the client and the server to agree on more features than WireGuard before any data can be transferred.
- The freedom to choose between different cipher suits and add more functionality increases the complexity of DTLS and creates a debate about whether it is too heavyweight for CoAP. There is no singular number for how many lines of code are needed to implement DTLS due to the amount of different implementations

available in different programming languages, but we can assume that the DTLS codebase is substantial and will impact CoAP performance negatively. WireGuard codebase, on the other hand, is under 4,000 lines of code, potentially making it the more favourable protocol for securing IoT communication.

- In contrast to DTLS, which uses certificates depending on the implementation, WireGuard does not require certificates. When using WireGuard, both sides exchange short 32-byte base64-encoded public keys.

Answering RQ2 provided us with an outline of the most notable differences between the two protocols. We believe that the outline shows several shortcomings of DTLS and provides solid incentives for researching more ways of securing CoAP communication such as WireGuard. The next sections will discuss this further based on the experiments performed as a part of our research.

6.3 Using WireGuard for securing CoAP communication

The question of whether or not WireGuard could be a suitable protocol for securing an IoT protocol such as CoAP was the central question of our research. RQ3 was formulated in Chapter 1.2 and is reintroduced below:

RQ3: Could WireGuard potentially be used instead of DTLS for CoAP as a protocol for securing communication?

- If yes, how?
- What would be the benefits of doing so? How would it affect the security concerns mentioned in RQ1, as well as CoAP's performance?

The two supporting questions were introduced to suggest a possible way of implementing CoAP over WireGuard and elaborate on the potential benefits of doing so. Furthermore, we formed three hypotheses that were tested during the experimental phase of our research. In this section, we will present and discuss each of the hypotheses. A general discussion of RQ3 will subsequently follow.

6.3.1 H1: WireGuard is faster than DTLS when used to secure CoAP

When it comes to IoT communication, better performance often means better speed. Different protocols have different features that affect how fast they operate. In our case, one such feature shared both by WireGuard and DTLS is the handshake. Therefore, measuring the time it takes to establish a handshake for both protocols

and its impact on the overall time span was essential. DTLS handshake has more flights than WireGuard handshake, making it reasonable to expect it to take a longer time. In our experimental setup described in Chapter 4, this was confirmed to be true.

As seen in Table 5.1, there is a significant difference between the two handshakes, with the WireGuard handshake being significantly faster. Moreover, when establishing a new connection between a client and a server in order to send one GET request, the processing time of this exchange is almost three times faster for CoAP/WireGuard than CoAP/DTLS. However, we should take into account that during real-life communication between IoT devices, there will be no need for DTLS to establish an entirely new connection between an already connected client and server when sending a new packet (Section 2.2.2 of Chapter 2). Not establishing a new handshake every time a connection is open would lower the handshake cost when it comes to the overall time span for real-life communication. However, Table 5.3 shows that even when removing the handshake completely, the overall time span for traffic sent with CoAP/DTLS is still substantially longer compared to both non-secured CoAP and CoAP/WireGuard.

When it comes to latency, we can observe that unprotected version of CoAP performs significantly better as it has no additional overhead (Table 5.3). Removing the handshake proves to make no difference for CoAP/WireGuard, while CoAP/DTLS performs slightly better without the handshake in terms of the overall time span. However, both with and without the handshake, WireGuard exhibits lower latency than DTLS. Similarly to the observation made by Laaroussi et al. [LN21], our results indicate less difference between the protocol the more delay is added to the network. Their results were most similar when adding 150ms fixed delay to their network, but since our client sends 150 GET requests as opposed to 19, the difference is minor already when the delay is 60ms.

Another metric we focused on during our experiments is RTT, presented in Section 5.2. Based on our experiments, the RTT for CoAP was only slightly affected when securing it with WireGuard. In comparison, when implementing CoAP over DTLS, RTT for the same amount of packets was almost tripled. This particular attribute was not affected by adding or removing the handshake both for DTLS and WireGuard. Based on the findings, we can conclude that CoAP/WireGuard exhibited better results compared to CoAP/DTLS.

All the results discussed so far show that sending CoAP packets through a WireGuard makes little to no difference in terms of both latency and RTT. It holds true for the last attributes we direct our attention at, namely throughput. Again, the non-secured version of CoAP exhibits a higher throughput as opposed to its two

secured versions, both in terms of the number of pps and throughput in Mbits/s (Table 5.4). Our results verify the results obtained by Laaroussi et al. [LN21] for a network with no delay. Similarly to their results, our findings indicate a significant decrease of the number of pps when adding implementing CoAP over DTLS compared to unsecured DTLS. As the paper also mentions, this result is anticipated, since DTLS handshake requires extra round trips. For the same reason, WireGuard's number of pps is also less than for non-secured CoAP. Adding a handshake negatively impacts the performance of the protocol being secured. Still, the results show that the footprint of the WireGuard handshake does not affect CoAP results as much as DTLS does.

When it comes to comparing the throughput achieved by the protocols in Mbits/s, the performance exhibited by CoAP/DTLS is significantly worse than the results measured for unsecured CoAP and CoAP/WireGuard. Moreover, when adding a fixed 60ms delay, we observe that the throughput achieved by CoAP/WireGuard is higher than the throughput achieved by the unsecured version of the protocol. The number of pps is approximately the same, but the slightly larger packet size of WireGuard packets ensures the 21.740% increase of the throughput discovered in Section 5.4. Moreover, due to the packet size of CoAP/DTLS packets, throughput is the only metric that gives very different results for the three protocols even when adding the network delay.

Altogether, the findings presented in Chapter 5 confirm our assumption that CoAP secured by WireGuard is faster than CoAP implemented over DTLS. It is both true for a network with no delay and a network with a fixed 60ms delay. However, it is important to consider that we can only claim this to be the case for our experimental setup. The protocols were tested in a fixed, prepared environment, and the way they behave will only apply in such an environment until proven otherwise.

Furthermore, even if an experimental setup is the same for all the protocols, the results obtained may still vary. As introduced in Chapter 4, each implementation in our research was tested twelve times: six times with no network delay and six times with a fixed delay of 60 ms. All the results presented in Appendix C.1 are not only derived by the arithmetic mean of those six experiments but also the standard deviation for each metric. Taking a closer look at the tables, we can observe a high standard deviation for CoAP/DTLS without the network delay, which usually indicates that the entries are very spread out. However, this is not the case for our experiments, as only one entry exhibited uncommonly high values for latency and handshake duration.

The overall time span for that particular experiment was almost twice as big as the mean of the other five experiments performed for CoAP/DTLS. The handshake

	CoAP	CoAP/DTLS	CoAP/WireGuard
Packets per second	454.150 ± 41.617	159.740 ± 16.982	355 ± 47.527
Packet size, B	287.500 ± 1.500	106 ± 0	355.500 ± 0.764
Throughput, Mbit/s	1.042 ± 0.092	0.135 ± 0.014	1.009 ± 0.136
Handshake time, ms	–	125 ± 49	1 ± 0.412
Handshake			
Time span, ms	666 ± 63	$1,936 \pm 190$	865 ± 126
RTT, ms	3 ± 0.210	7 ± 0.620	3 ± 0.419
No handshake			
Time span, ms	666 ± 63	$1,811 \pm 171$	865 ± 126
RTT, ms	3 ± 0.210	6 ± 0.570	3 ± 0.157

Table 6.1: The results for the three implementations of the CoAP protocol based on our experiments, excluding the outlier. (150 GET requests with no delay).

duration for the exact exchange was measured to be around five times longer than the mean of the other five handshakes. Subsequently, other metrics dependent on time, such as the number of packets per second and RTT, were also affected, albeit not as much as expected. Removing this singular entry reduces the overall time span to 1,936 ms with the handshake and 1,811 without. Regarding the handshake duration, the mean value goes from 218 ms down to 125 ms, which is undoubtedly significant. The final results without the discussed CoAP/DTLS can be seen in Table 6.1, as well as in Table C.2. Nevertheless, even when considering this, we can see that WireGuard still performs significantly faster than its counterpart, DTLS.

This particular entry for is a so-called *outlier*[4]. Many statistical procedures, including calculation of the arithmetic mean, are sensitive to the presence of outliers [RM17]. Excluding the outliers can lead to the results becoming more statistically significant [Fro].

In order to treat the outliers properly, it is essential to understand how they occur. The reason for this particular outlier occurring during our experiments is hard to ascertain. The experiments were run one after another, with no breaks when testing each protocol. Both the client and the server only communicated via a virtual network interface, `vboxnet0`. Thus, the experiments should not have been affected by any changes in the public university network.

Nevertheless, there was a possibility of external factors such as the speed of

[4]An *outlier* is generally a value which is a lot higher or lower than the main body of the data [RM17].

VirtualBox influencing the outcome of the experiments. Since the actual reasons were difficult to define, the outlier value was left as part of our results. The value will be further mentioned in this chapter whenever it directly affects the results being discussed.

6.3.2 H2: WireGuard adds less overhead than DTLS when used to secure CoAP

Overhead can be represented in the form of the handshake time, number of extra packets or number of extra headers. Figure 6.1 depicts the overall processing time it takes for CoAP/DTLS and CoAP/WireGuard respectively based on the times from Figure 5.1. We use these figures to analyse how much impact the handshakes have on the protocols, and to discuss how much protocol and packet overhead DTLS and WireGuard add to CoAP.

As it was shown in Section 5.1, DTLS adds six additional packets to CoAP, which results in additional overhead of 1573 bytes (12,584 bits). WireGuard, on the other side, only adds two more packets for the handshake, which is three times less. The whole WireGuard handshake increases the total number of bytes transferred by 328 (2,624 bits). This type of overhead is called the *protocol overhead*.

Although it is unknown how many packets were exchanged between the client and the server during the experiments discussed by Kondoro et al. [KBTM21], we can assume that the results presented there are also the means calculated after running the experiments several times. This assumption is used when comparing our results with the results from the paper. During our experiments, we could see that the DTLS security handshake takes up approximately 75% of the total processing time. Section 7.1 of [KBTM21] states this number to be 70%. Our research determined that WireGuard only consumes 1.436% of the whole processing time. These results are true for a network with no delay.

Based on our findings, it is evident that our assumption about WireGuard adding less overhead to CoAP than DTLS was correct. However, this hypotheses is only accurate when looking at the overhead added by the handshake, i.e. the protocol overhead. Figure 6.1 depicts the first data transfer after the handshake is complete. In our experiment, all the subsequent data transfers are identical to the first one, since we only request one particular resource from the server. Figure 6.1 shows that the GET requests sent using DTLS and WireGuard are not the same size. The same can be said about the response sent from the server. Curiously, the packets sent using WireGuard are noticeably bigger than the DTLS packets. The same pattern can be observed in Figure C.1; the value of the mean packet size is around 3.5 times larger for CoAP/WireGuard than CoAP/DTLS.

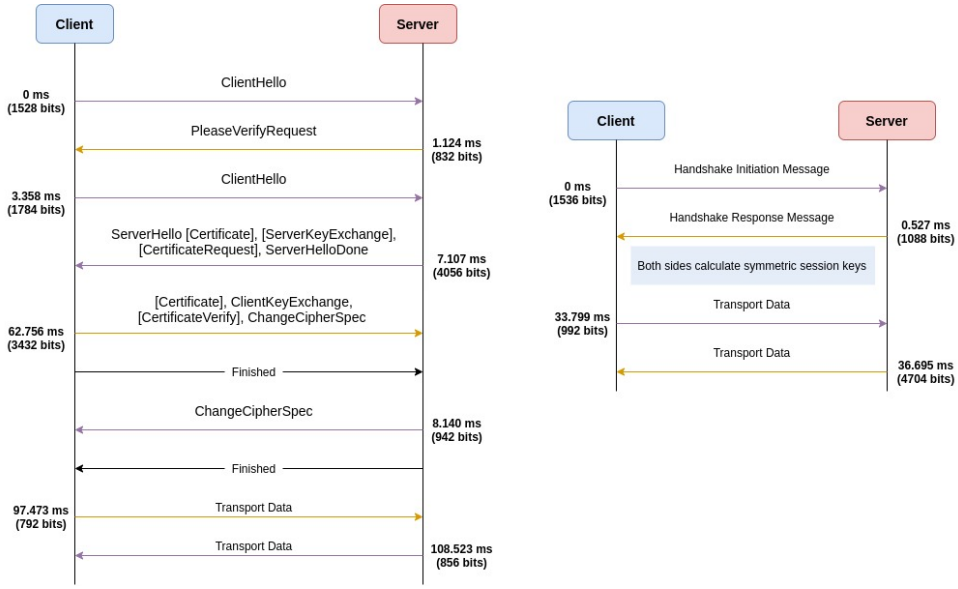


Figure 6.1: A visual depiction of the DTLS and WireGuard handshakes from Figure 5.1

Based on our findings, the size of a GET request using non-secured CoAP equals 62 bytes (448 bits), while the response is measured to be 515 bytes (4120 bits). When comparing the overhead added to these messages by DTLS and WireGuard, we can see that WireGuard overhead is larger. Moreover, the response message sent with DTLS is notably smaller than both non-secured CoAP and WireGuard responses. It is possible to use a TLS compression algorithm [DE08] when using DTLS to secure CoAP, which might be the reason for why DTLS response is smaller than even a non-secured CoAP response.

However, in the Scandium code the method is initially defined as *CompressionMethod.null*. Furthermore, [SHB14] advises that TLS compression should be disabled to mitigate compression-related attacks, and there is no indication that Scandium uses any sort of message compression. Another possible reason for this can be the differences in the implementation of Californium and Scandium, as Scandium does not seem to be directly built "on top" of Californium, but rather as its own implementation of CoAP and DTLS. In the end, the results presented in Table C.1 and Table C.3 show that WireGuard is more effective despite potentially adding more message overhead. It is clear that the DTLS handshake significantly affects the performance of CoAP/DTLS implementation in a negative way. Even if we do not consider the *outlier*-value discussed in Section 6.3.1, there is still a considerable difference between WireGuard and DTLS overhead.

Nevertheless, the findings presented in this thesis do not describe all the possible situations when using DTLS. As described in Chapter 4, CoAP/DTLS was tested by running the implementation both over a time span of several hours and overnight. When doing so, we confirmed that no handshakes apart from the initial handshake to initiate the connection were observed. The Californium implementation of CoAP allows client and server to resume a session either using session ID or master_secret, without using a full handshake. RFC5077 [SZET08] suggests that the upper limit for session ID lifetime should have an upper limit of 24 hours. As DTLS is based on TLS, it is logical to assume that DTLS follows the same guidelines. This upper limit is advised due to security concerns regarding a master_secret that is shared between a client and a server. Having obtained the master_secret, an adversary might be able to impersonate the compromised party up until the session ID is retired. Therefore, the more rare the handshake is the less secure the session might be. It is a classic security-performance trade-off situation for IoT communication, where a choice must be taken based on the requirements for the devices needed to be secure.

WireGuard handshake occurs every few minutes after the initial handshake that established symmetric keys for data transfer. The protocol implements a pulse mechanism to ensure that the latest keys and handshakes are not out of date and renegotiates them when needed. Furthermore, by initiating a new handshake every few minutes, WireGuard provides rotating keys for perfect forward secrecy [Don17]. Despite WireGuard initiating handshake more often than DTLS, the total overhead for a network with no delay will most likely still be more for DTLS than WireGuard. Using our results (Appendix C.1), we can estimate that WireGuard will need to handshake 218 times only to reach the time span of one DTLS handshake, or 125 when removing the outlier entry. Moreover, the RTT for WireGuard is less both with and without a handshake, meaning that we can discard the other packets when talking about the packet overhead.

6.3.3 H3: It would be beneficial to choose WireGuard over CoAP rather than DTLS when securing CoAP

H3, together with the supporting question RQ3.2, were defined at the beginning of our research to facilitate the discussion of RQ3. The supporting question was defined as follows:

> **RQ3.2:** What would be the benefits of doing so and how would it affect the security concerns mentioned in RQ1, as well as CoAP's performance?

The first question concerns both the performance and security side of the possible CoAP/WireGuard implementation. When it comes to performance, the results

obtained from our experiments are extensively examined in the previous sections of this chapter, as well as Chapter 5. Our results suggest that using WireGuard for securing CoAP can lead to a better overall performance of the protocol. Security-wise, both advantages and shortcoming of DTLS and WireGuard have been discussed in the preceding chapters. In Section 6.1, we established that DTLS can provide confidentiality for IoT devices but falls short when it comes to securing integrity and availability.

WireGuard's creators claim that WireGuard only provides perfect forward secrecy but also built-in protection against key impersonation, replay attack and identity hiding [Don17]. Furthermore, WireGuard's security primitives such as Noise Protocol Framework and ChaCha20Poly1305, together with the stateless cookie mechanism (Section 2.4), serve as a good foundation for implementing the CIA triad. In addition, these security primitives are less resource consuming, as shown by [LKS19, PWA+19], along with our research.

Nevertheless, there are some potential disadvantages and security concerns linked to using WireGuard for IoT devices worth mentioning. As the protocol has a fixed set of cryptographic algorithms, some of the algorithms may be broken sometime in the future, which will require an update of all the devices implementing WireGuard. With the number of tiny IoT devices potentially utilising WireGuard, updating all of them at the same time may be cumbersome. Implementing stable self-update features on the devices may help to reduce the scope of the issue.

When it comes to potentially breaking the cryptographic primitives used by WireGuard, it is essential to mention the concept of post-quantum computers. WireGuard is not yet post-quantum secure, meaning that when large quantum computers are created, they may break the underlying algorithms used by WireGuard. This will significantly weaken the security of WireGuard. In section Section 2.5, some papers presenting potential improvements regarding post-quantum cryptography are presented. Despite being more secure, a new, post-quantum version of WireGuard may become too heavyweight for CoAP and other similar protocols. IoT devices would thus have to evolve in such a way that the new post-quantum version does not significantly affect their performance.

Ultimately, however, when designing an IoT network and choosing a protocol, it is up to the network planner to choose the protocol that best suits the predetermined requirements for both the devices themselves and the traffic being exchanged in the network. Currently, DTLS is the most common protocol used for securing CoAP, and as seen in Section 2.5, there is considerable interest in improving the security and performance of the protocol. However, securing IoT devices is an ongoing endeavour, and new protocols such as WireGuard will likely become a focus of more thorough

research in the near future. In addition to generating more research, WireGuard needs to be developed to better suit IoT devices in order to become a full-fledged safe alternative to a well-establish protocol such as DTLS.

6.3.4 Implementing CoAP over WireGuard

During the course of this study, we wanted to establish whether it would be feasible to secure CoAP using WireGuard not only from the theoretical point of view, but also in terms of an actual implementation. The supporting question RQ3.1 was thus designed to facilitate the investigation of RQ3 and was defined as follows:

> **RQ3:** Could WireGuard potentially be used instead of DTLS for CoAP as a protocol for securing communication?
>
> – **RQ3.1:** If yes, how?

As this thesis shows, it is possible to secure CoAP communication using WireGuard. In our research, CoAP server and client sent packets to each other via an encrypted WireGuard tunnel, as depicted in Figure 4.1. However, the setup described in Chapter 4 is experimental and is not entirely applicable for real-life IoT communication. The limitations enforced on the devices by the CPU and reduced storage will likely require a change in implementation methods. Still, it is entirely possible to build and run a CoAP Californium server on a Raspberry Pi and other IoT devices[5].

Moreover, many modern devices are running on Linux distributions, with AndroidThings, Raspbian and Ubuntu Core being some of them. 2020 IoT Developer Survey [Fou20] reveals that 43% of the developers choose Linux as the operating system for constrained devices and edge nodes. Since WireGuard is now integrated into the Linux kernel, these IoT devices have access to it and should have no trouble installing it. Furthermore, our method of running CoAP over WireGuard requires little to no modifications done to the Californium code, simplifying the task of configuring CoAP/WireGuard on small IoT devices.

This chapter further examined the experimental results presented in Chapter 5 and answered the RQs established in Chapter 1 with the help of the Hypotheses and supporting questions. Discussing the results in light of the RQs provided a base for suggesting that using Wireguard for securing CoAP will benefit the overall peformance of the IoT protocol.

[5]https://www.raspberrypi.org/forums/viewtopic.php?t=277111

Chapter 7
Conclusion and Future work

7.1 Conclusion

During the last decades, IoT devices have become an integrated part of various infrastructures, including critical infrastructures such as the healthcare and public health sector, transportation and manufacturing. Furthermore, smart homes composed of small smart devices are becoming increasingly popular, as such devices may simplify day-to-day tasks and thus improve the quality of life. These devices usually have limited processing, storage and power capacity and can be referred to as constrained devices. Often, constrained devices form constrained networks and regularly gather and exchange data. Their growing popularity is accompanied by growing interest from adversaries whose goal is to compromise both the devices' and their owners' safety and security.

This thesis has explored the main security challenges faced by the constrained nodes and networks by reviewing the previous works related to the topic. The review established that security and performance issues of resource-constrained devices are closely connected since securing the communication creates an overhead that reduces the performance of many IoT devices and networks. In addition to the literature review, this study has presented a comparative networks performance analysis of CoAP and its two secured implementations, CoAP/DTLS and CoAP/WireGuard. The primary goal of the thesis was to find out whether it would be beneficial to secure CoAP with WireGuard instead of DTLS both when it comes to security and performance.

RQ1 explored the main security concerns regarding IoT communication using CoAP and summarised them in connection to the CIA triad. Information gathered and stored by the constrained nodes must not be shared with the nodes not authorised to access it. When the information is shared with other nodes, it must be accurate and complete; receivers expect information not to have been tampered with and come from the right senders. Furthermore, the nodes must be reachable and available

at any point in time to provide access to the data to users and other nodes whenever they require it. Therefore, a protocol securing IoT communication must ensure confidentiality, integrity and availability for the devices and networks it is securing. In addition to establishing that, RQ1 examined whether DTLS is able to provide this level of security for CoAP. Taking into consideration the drawbacks and limitation of DTLS, we concluded that the protocol could be used to ensure confidentiality by using security and authentication modes provided for CoAP. However, DTLS does not give full protection against attacks such as IP address spoofing and DDoS attacks, and thus cannot ensure full integrity and availability.

RQ2 focused on comparing DTLS and WireGuard in order to find features shared by the two protocols, as well as differences they exhibit. The results showed that the two protocols are predominantly different and emphasise different features. For instance, network layer protocols WireGuard tries to minimise potential overhead by offering a fixed set of cipher suites and a short handshake. DTLS, on the other hand, is a transport layer protocol that provides a possibility for customisation of cipher suites based on the network's and devices' demand and requires a more extended handshake in order for client and server to agree on how they want to communicate further. The flexibility and overhead added by the handshake adversely affect the performance of the protocol compared to WireGuard. Overall, RQ2 outlined some of the shortcomings of DTLS and supported further research of other ways to secure CoAP.

RQ3 summarised the experimental results and confirmed that adding a security overhead has a significant impact on the network performance of CoAP. The overhead was most noticeable when using DTLS for securing CoAP, with the handshake having a substantial negative impact on CoAPs performance. The most effective protocol in terms of performance was deemed to be the non-secured version of CoAP. The overhead measured for WireGuard was much smaller, and even when comparing the implementations without considering the handshake, the outcome remained the same. The results demonstrated that CoAP implemented over WireGuard exhibited better performance in terms of both latency, handshake time, RTT and throughput.

The experimental results obtained during this study suggest that WireGuard could be a better fit than DTLS for securing CoAP as it reduces the security overhead, which leads to better overall network performance. Furthermore, WireGuard's security primitives provide a good foundation for ensuring confidentiality, integrity and availability, making it not only preferable in terms of performance but also security requirements of IoT devices.

7.2 Future work

As a part of further research, we propose evaluating the performance of the implementation on real resource-constrained IoT devices running Linux OS. Moreover, we believe it would be valuable to evaluate the implementation in terms of both network performance and other relevant performance metrics to provide new insight into the effectiveness and efficiency of the protocols. These metrics may include but not be limited to power consumption and memory space. When it comes to experimenting with real-life IoT devices, it could also be relevant to test the protocols using devices utilising microcontrollers with an integrated hardware encryption engine. With some of them supporting the AES, DES and 3DES standards, it may have a significant impact on overhead and how resource-intensive DTLS and WireGuard encryption would be.

Among other possibilities for examining the implementation compared during our research is completely disabling the encryption for DTLS to reduce the overhead from cryptographic operations in the software utilising DTLS. In this case, disabling the encryption means using the NULL cipher suite provided by DTLS. Moreover, a new and improved version of DTLS, DTLS 1.3, has recently been introduced. Since there was no available implementation of CoAP over DTLS 1.3 at the time of conducting this study, it would be interesting to repeat the experiments and add a CoAP/DTLS 1.3 implementation for further comparison.

References

[ABD+15] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, et al. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 5–17, 2015.

[AN19] S. Arvind and V. Narayanan. An Overview of Security in CoAP: Attack and Analysis. *2019 5th International Conference on Advanced Computing & Communication Systems (ICACCS)*, pages 655–660, 2019.

[Asi17] Makkad Asim. Security in Application Layer Protocols for IOT: A Focus on COAP. *International Journal of Advanced Research in Computer Science*, 8(5), 2017.

[BB11] A. Barth and U. C. Berkley. RFC 6265 - HTTP State Management Mechanism. https://www.barretlee.com/ST/RFC-HTTP/, April 2011. (Accessed on 04/13/2021).

[BCS12] C. Bormann, A. P. Castellani, and Z. Shelby. CoAP: An Application Protocol for Billions of Tiny Internet Nodes. *IEEE Internet Computing*, 16(2):62–67, 2012.

[BEK14] C. Bormann, M. Ersue, and A. Keranen. RFC 7228: Terminology for Constrained-Node Networks. https://www.hjp.at/doc/rfc/rfc7228.html, May 2014. (Accessed on 03/31/2021).

[Bev19] R. Bevans. A Quick Guide to Experimental Design: 4 Steps & Examples. https://www.scribbr.com/methodology/experimental-design/, December 2019. (Accessed on 06/12/2021).

[Bha20] P. Bhandari. What Is Quantitative Research? Definition, Uses and Methods. https://www.scribbr.com/methodology/quantitative-research/, June 2020. (Accessed on 06/12/2021).

[BMN19] Anup Burange, Harshal Misalkar, and Umesh Nikam. Security in MQTT and CoAP Protocols of IOT's Application Layer. In Shekhar Verma, Ranjeet Singh

Tomar, Brijesh Kumar Chaurasia, Vrijendra Singh, and Jemal Abawajy, editors, *Communication, Networks and Computing*, pages 273–285, Singapore, 2019. Springer Singapore.

[CCCP15] Angelo Capossele, Valerio Cervo, Gianluca Cicco, and Chiara Petrioli. Security as a CoAP resource: An optimized DTLS implementation for the IoT. 06 2015.

[CRC08] Patricia Cronin, Frances Ryan, and Michael Coughlan. Undertaking a literature review: A step-by-step approach. *British journal of nursing (Mark Allen Publishing)*, 17:38–43, 01 2008.

[DE08] T. Dierks and Rescorla. E. RFC5246 - The Transport Layer Security (TLS) Protocol Version 1.2. https://datatracker.ietf.org/doc/html/rfc5246, August 2008. (Accessed on 06/17/2021).

[Del05] Amy B Dellinger. Validity and the review of literature. *Research in the Schools*, 12(2):41–54, 2005.

[DM17] Jason A Donenfeld and Kevin Milner. Formal verification of the WireGuard protocol. Technical report, Technical report, July, 2017.

[Don] Jason A Donenfeld. Known Limitations - WireGuard. https://www.wireguard.com/known-limitations/. (Accessed on 05/10/2021).

[Don17] Jason A Donenfeld. WireGuard: Next Generation Kernel Network Tunnel. In *NDSS*, 2017.

[Don18] Jason A Donenfeld. WireGuard: Fast, modern, secure VPN tunnel. *Black Hat USA*, 2018.

[DP18] Benjamin Dowling and Kenneth G. Paterson. A Cryptographic Analysis of the WireGuard Protocol. Cryptology ePrint Archive, Report 2018/080, 2018. https://eprint.iacr.org/2018/080.

[FBJM+20] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS Implementations Using Protocol State Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[FGM+96] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, R. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1. https://www.hjp.at/doc/rfc/rfc2616.html, June 1996. (Accessed on 04/01/2021).

[FKH13] N. Freed, J. Klensin, and T. Hansen. RFC 6838 - Media Type Specifications and Registration Procedures. https://tools.ietf.org/html/rfc6838, January 2013. (Accessed on 04/07/2021).

[Fou20] The Eclipse Foundation. 2020 IoT Developer Survey: Key Findings. The Eclipse Foundation, 2020.

[FR14] R. Fielding and J. Reschke. RFC 7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. https://www.hjp.at/doc/rfc/rfc7231.html, June 2014. (Accessed on 04/07/2021).

[Fro] J. Frost. Guidelines for Removing and Handling Outliers in Data - Statistics By Jim. https://statisticsbyjim.com/basics/remove-outliers/. (Accessed on 06/21/2021).

[GBP+21] Martin Gunnarsson, Joakim Brorsson, Francesca Palombini, Ludwig Seitz, and Marco Tiloca. Evaluating the performance of the OSCORE security protocol in constrained IoT environments. *Internet of Things*, 13:100333, 2021.

[GMA12] L.R. Gay, G.E. Mills, and P.W. Airasian. *Educational Research: Competencies for Analysis and Applications*. Pearson College Division, 2012.

[HNS+21] A. Hülsing, K. Ning, P. Schwabe, F. Weber, and P. R. Zimmermann. Post-Quantum WireGuard. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 511–528, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[HT15] Roberta Heale and Alison Twycross. Validity and reliability in quantitative studies. *Evidence-Based Nursing*, 18(3):66–67, 2015.

[Hun17] Mark Hung. Leading the IoT: Gartner Insights on How to Lead in a connected World. https://www.gartner.com/imagesrv/books/iot/iotEbook_digital.pdf, 2017. (Accessed on 11/13/2020).

[Kaz20] E. Kazakova. WireGuard for securing Constrained Application Protocol for IoT devices. Project report in TTM4502, Department of Information Security and Communication Technology, NTNU – Norwegian University of Science and Technology, Dec. 2020.

[KBTM21] Aron Kondoro, Imed Ben Dhaou, Hannu Tenhunen, and Nerey Mvungi. Real time performance analysis of secure IoT protocols for microgrid communication. *Future Generation Computer Systems*, 116:1–12, 2021.

[KMR20] Quentin M. Kniep, Wolf Müller, and Jens-Peter Redlich. Post-Quantum Cryptography in WireGuard VPN. In Noseong Park, Kun Sun, Sara Foresti, Kevin Butler, and Nitesh Saxena, editors, *Security and Privacy in Communication Networks*, pages 261–267, Cham, 2020. Springer International Publishing.

[KOL19] Kenneth Kimani, Vitalice Oduol, and Kibet Langat. Cyber security challenges for IoT-based smart grid networks. *International Journal of Critical Infrastructure Protection*, 25:36–49, 2019.

[LKS19] Tim Lackorzynski, Stefan Köpsell, and Thorsten Strufe. A comparative study on virtual private networks for future industrial communication systems. In *2019 15th IEEE International Workshop on Factory Communication Systems (WFCS)*, pages 1–8. IEEE, 2019.

[LN21] Zakaria Laaroussi and Oscar Novo. A Performance Analysis of the Security Communication in CoAP and MQTT. In *2021 IEEE 18th Annual Consumer Communications Networking Conference (CCNC)*, pages 1–6, 2021.

[LS14] Vishwas Lakkundi and Keval Singh. Lightweight DTLS implementation in CoAP-based Internet of Things. September 2014.

[Mid20] F. Middleton. The 4 types of validity | explained with easy examples. https://www.scribbr.com/methodology/types-of-validity/, June 2020. (Accessed on 06/23/2021).

[MP+] Snehal Mumbaikar, Puja Padiya, et al. Web services based on SOAP and REST principles. *International Journal of Scientific and Research Publications*, 3(5):1–4.

[MYAZ15] R. Mahmoud, T. Yousuf, F. Aloul, and I. Zualkernan. Internet of things (IoT) security: Current status, challenges and prospective measures. In *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 336–341, 2015.

[NC20] Giuseppe Nebbione and Maria Carla Calzarossa. Security of IoT Application Layer Protocols: Challenges and Findings. *Future Internet*, 12(3), 2020.

[NSS] Y. Nir, R. Salz, and N. Sullivan. Transport Layer Security (TLS) Parameters. https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml, Last updated: 06/04/2021. (Accessed on 06/18/2021).

[P+80] Jon Postel et al. RFC768 - User datagram protocol. 1980.

[PWA+19] Sven Plaga, Norbert Wiedermann, Simon Duque Anton, Stefan Tatschner, Hans Schotten, and Thomas Newe. Securing future decentralised industrial IoT infrastructures: Challenges and free open source solutions. *Future Generation Computer Systems*, 93:596–608, 2019.

[RM12] E. Rescorla and N. Modadugu. RFC 6347 - Datagram Transport Layer Security Version 1.2. https://tools.ietf.org/html/rfc6347, January 2012. (Accessed on 11/15/2020).

[RM17] Colin Robson and Kieran McCartan. *Real World Research, 4th Edition*. 12 2017.

[RS16] R. A. Rahman and B. Shah. Security analysis of IoT protocols: A focus in CoAP. In *2016 3rd MEC International Conference on Big Data and Smart City (ICBDSC)*, pages 1–7, March 2016.

[RSH+13] Shahid Raza, Hossein Shafagh, Kasun Hewage, René Hummen, and Thiemo Voigt. Lithe: Lightweight secure CoAP for the internet of things. *IEEE Sensors Journal*, 13(10):3711–3720, 2013.

[SHB14] Z. Shelby, K. Hartke, and C. Bormann. RFC 7252 - The Constrained Application Protocol (CoAP). https://tools.ietf.org/html/rfc7252#section-11, June 2014. (Accessed on 03/11/2021).

[SHSA15a] Y. Sheffer, R. Holz, and P. Saint-Andre. RFC 7457 - Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS). https://www.hjp.at/doc/rfc/rfc7457.html, February 2015. (Accessed on 06/18/2021).

[SHSA15b] Y. Sheffer, R. Holz, and P. Saint-Andre. RFC 7525 - Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). https://www.hjp.at/doc/rfc/rfc7525.html, May 2015. (Accessed on 06/18/2021).

[SZET08] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. RFC5077 - Transport Layer Security (TLS) Session Resumption without Server-Side State. https://datatracker.ietf.org/doc/html/rfc5077, January 2008. (Accessed on 06/16/2021).

[tea] IPVN team. PPTP vs IPSec IKEv2 vs OpenVPN vs WireGuard. (Accessed on 05/3/2021).

[TF16] Hannes Tschofenig and Thomas Fossati. Transport Layer Security (TLS)/Datagram Transport Layer Security (DTLS) profiles for the Internet of Things. In *RFC 7925*. Internet Engineering Task Force, 2016.

[TMV+14] D. Thangavel, X. Ma, A. Valera, H. Tan, and C. K. Tan. Performance evaluation of MQTT and CoAP via a common middleware. In *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, pages 1–6, 2014.

[Wie14] R.J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer Berlin Heidelberg, 2014.

[WLP18] Min Woo Woo, JongWhi Lee, and KeeHyun Park. A reliable IoT system for Personal Healthcare Devices. *Future Generation Computer Systems*, 78:626–640, 2018.

[WO20] Merrill Warkentin and Craig Orgeron. Using the security triad to assess blockchain technology in public sector applications. *International Journal of Information Management*, 52:102090, 2020.

[ZCW+14] Z. Zhang, M. C. Y. Cho, C. Wang, C. Hsu, C. Chen, and S. Shieh. IoT Security: Ongoing Challenges and Research Opportunities. In *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*, pages 230–234, 2014.

[ZKKK19] Fotios Zantalis, Grigorios Koulouras, Sotiris Karabetsos, and Dionisis Kandris. A Review of Machine Learning and IoT in Smart Transportation. *Future Internet*, 11(4), 2019.

Appendix

Appendix A

A.1 Implementations

A.1.1 WireGuard configuration files

```
[Interface]
Address = 10.1.0.1/24
ListenPort = 41194
PrivateKey = mCL1t5Qb03+cVlVqz5Gwqq6Da56j1pxk2DJYmel2qkM=

[Peer]
# Bob's public key
PublicKey = e7HjfGJtGtpLKzJpCses1AfMPE9iYlOf2S64ztWxWVE=
AllowedIPs = 10.1.0.0/24
Endpoint = 192.168.56.1:41194
```

Listing A.1: Configuration file (Alice)

```
[Interface]
Address = 10.1.0.2/24
ListenPort = 41194
PrivateKey = wEnLE0NpX8ysREhKt/H+7r6vMIggjdwMGnXYY1RUb1s=

[Peer]
# Alice's public key
PublicKey = vwjIhUrppqeBTsBepw347Fc7U4yu7ezTeZ5Vy3yMj1w=
AllowedIPs = 10.1.0.0/24
Endpoint = 192.168.56.103:41194
```

Listing A.2: Configuration file (Bob)

A.1.2 HelloWorld server CoAP

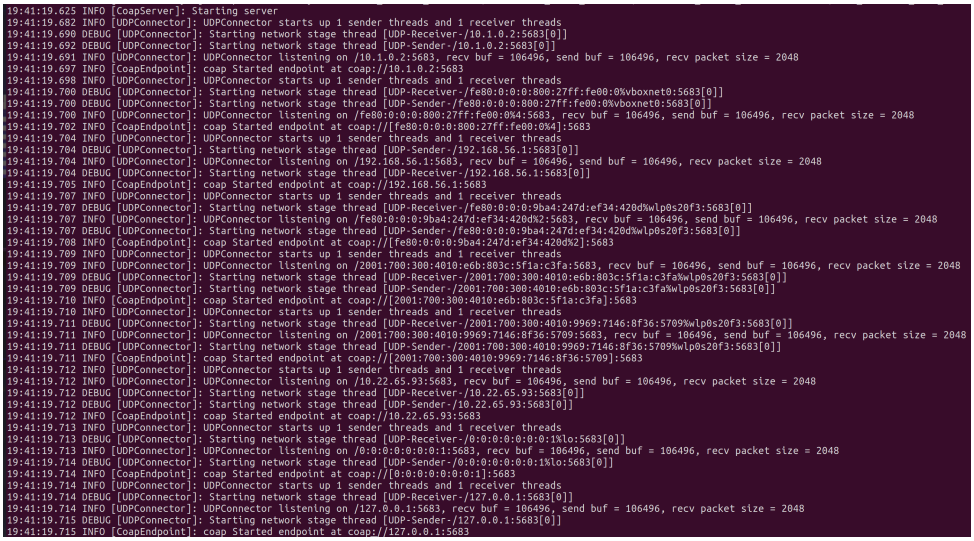


Figure A.1: Starting the HelloWorld server for CoAP

Appendix

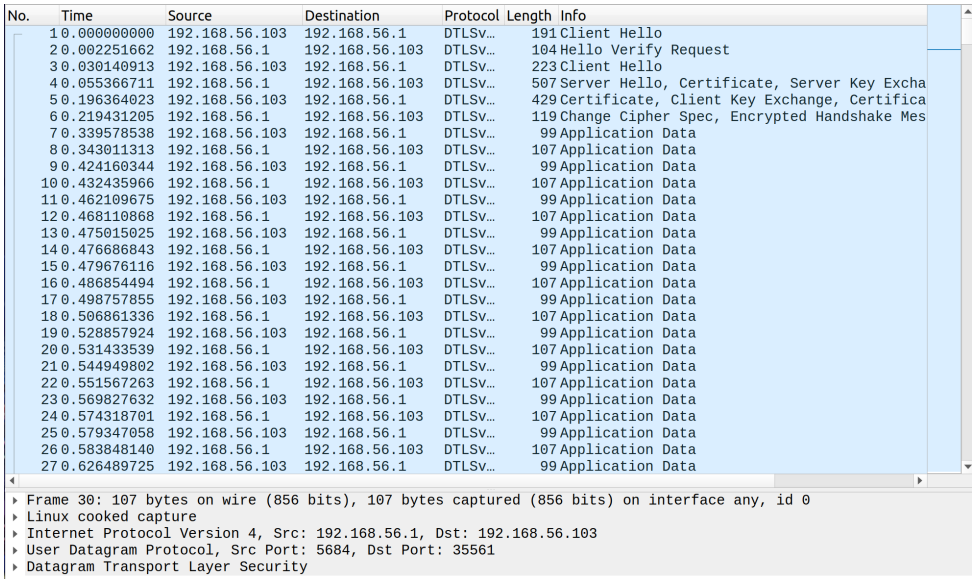
Appendix B

B.1 WireShark captures

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.56.103	192.168.56.1	CoAP	62	CON, MID:20364, GET, TKN:10 d3 6c 35 4b b2
2	0.001128766	192.168.56.1	192.168.56.103	CoAP	515	ACK, MID:20364, 2.05 Content, TKN:10 d3 6c
3	0.040751176	192.168.56.103	192.168.56.1	CoAP	62	CON, MID:20365, GET, TKN:6c fa 68 dc 3b 33
4	0.042307808	192.168.56.1	192.168.56.103	CoAP	515	ACK, MID:20365, 2.05 Content, TKN:6c fa 68
5	0.052255497	192.168.56.103	192.168.56.1	CoAP	62	CON, MID:20366, GET, TKN:fc d4 9e 59 12 d2
6	0.053486057	192.168.56.1	192.168.56.103	CoAP	515	ACK, MID:20366, 2.05 Content, TKN:fc d4 9e
7	0.058163698	192.168.56.103	192.168.56.1	CoAP	62	CON, MID:20367, GET, TKN:24 ae 8e 13 a2 24
8	0.058792068	192.168.56.1	192.168.56.103	CoAP	515	ACK, MID:20367, 2.05 Content, TKN:24 ae 8e
9	0.082066681	192.168.56.103	192.168.56.1	CoAP	62	CON, MID:20368, GET, TKN:90 18 bb 02 5b 60
10	0.084672823	192.168.56.1	192.168.56.103	CoAP	515	ACK, MID:20368, 2.05 Content, TKN:90 18 bb
11	0.093848542	192.168.56.103	192.168.56.1	CoAP	62	CON, MID:20369, GET, TKN:50 c4 25 28 db 61
12	0.095764543	192.168.56.1	192.168.56.103	CoAP	515	ACK, MID:20369, 2.05 Content, TKN:50 c4 25
13	0.102536348	192.168.56.103	192.168.56.1	CoAP	62	CON, MID:20370, GET, TKN:f0 4d 30 52 03 84
14	0.104544164	192.168.56.1	192.168.56.103	CoAP	515	ACK, MID:20370, 2.05 Content, TKN:f0 4d 30
15	0.115907836	192.168.56.103	192.168.56.1	CoAP	62	CON, MID:20371, GET, TKN:a0 69 29 4a d9 92
16	0.117551276	192.168.56.1	192.168.56.103	CoAP	515	ACK, MID:20371, 2.05 Content, TKN:a0 69 29
17	0.128590316	192.168.56.103	192.168.56.1	CoAP	62	CON, MID:20372, GET, TKN:9c 18 fc 88 e2 e0
18	0.131371181	192.168.56.1	192.168.56.103	CoAP	515	ACK, MID:20372, 2.05 Content, TKN:9c 18 fc
19	0.140712259	192.168.56.103	192.168.56.1	CoAP	62	CON, MID:20373, GET, TKN:f0 dd 88 92 1c 83
20	0.142007954	192.168.56.1	192.168.56.103	CoAP	515	ACK, MID:20373, 2.05 Content, TKN:f0 dd 88
21	0.149902306	192.168.56.103	192.168.56.1	CoAP	62	CON, MID:20374, GET, TKN:84 00 bd 2b 8b d3
22	0.152198425	192.168.56.1	192.168.56.103	CoAP	515	ACK, MID:20374, 2.05 Content, TKN:84 00 bd
23	0.160711770	192.168.56.103	192.168.56.1	CoAP	62	CON, MID:20375, GET, TKN:e0 75 49 d9 ff e9
24	0.162880568	192.168.56.1	192.168.56.103	CoAP	515	ACK, MID:20375, 2.05 Content, TKN:e0 75 49
25	0.168095904	192.168.56.103	192.168.56.1	CoAP	62	CON, MID:20376, GET, TKN:fc f8 fb b4 cc a6
26	0.169018510	192.168.56.1	192.168.56.103	CoAP	515	ACK, MID:20376, 2.05 Content, TKN:fc f8 fb
27	0.178552423	192.168.56.103	192.168.56.1	CoAP	62	CON, MID:20377, GET, TKN:60 4a 8a 7c 46 d0

```
▸ Frame 33: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on interface any, id 0
▸ Linux cooked capture
▸ Internet Protocol Version 4, Src: 192.168.56.103, Dst: 192.168.56.1
▸ User Datagram Protocol, Src Port: 40581, Dst Port: 5683
▸ Constrained Application Protocol, Confirmable, GET, MID:20380
▸ VSS Monitoring Ethernet trailer, Source Port: 0
```

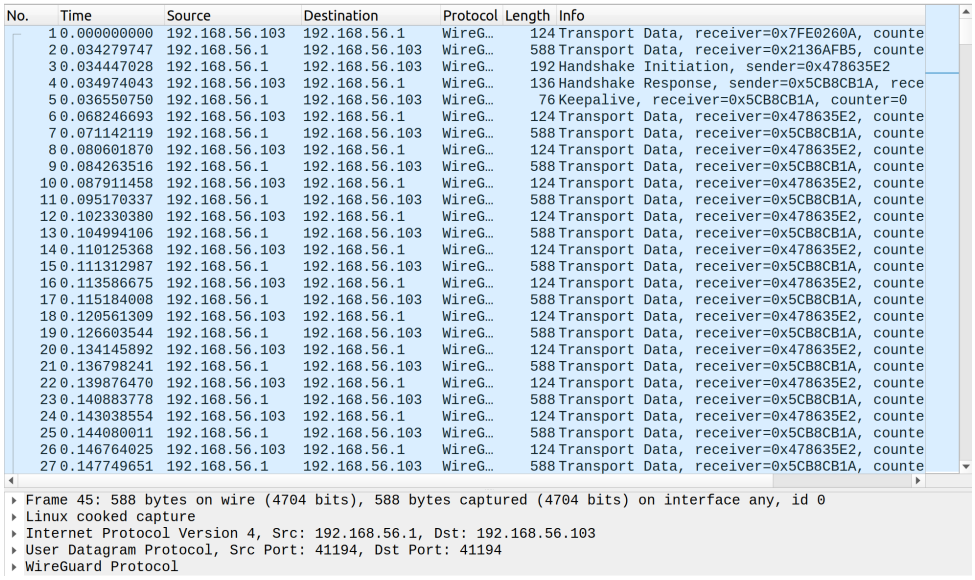
Figure B.1: CoAP for Alice as a client



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.56.103	192.168.56.1	DTLSv…	191	Client Hello
2	0.002251662	192.168.56.1	192.168.56.103	DTLSv…	104	Hello Verify Request
3	0.030140913	192.168.56.103	192.168.56.1	DTLSv…	223	Client Hello
4	0.055366711	192.168.56.1	192.168.56.103	DTLSv…	507	Server Hello, Certificate, Server Key Excha
5	0.196364023	192.168.56.103	192.168.56.1	DTLSv…	429	Certificate, Client Key Exchange, Certifica
6	0.219431205	192.168.56.1	192.168.56.103	DTLSv…	119	Change Cipher Spec, Encrypted Handshake Mes
7	0.339578538	192.168.56.103	192.168.56.1	DTLSv…	99	Application Data
8	0.343011313	192.168.56.1	192.168.56.103	DTLSv…	107	Application Data
9	0.424160344	192.168.56.103	192.168.56.1	DTLSv…	99	Application Data
10	0.432435966	192.168.56.1	192.168.56.103	DTLSv…	107	Application Data
11	0.462109675	192.168.56.103	192.168.56.1	DTLSv…	99	Application Data
12	0.468110868	192.168.56.1	192.168.56.103	DTLSv…	107	Application Data
13	0.475015025	192.168.56.103	192.168.56.1	DTLSv…	99	Application Data
14	0.476686843	192.168.56.1	192.168.56.103	DTLSv…	107	Application Data
15	0.479676116	192.168.56.103	192.168.56.1	DTLSv…	99	Application Data
16	0.486854494	192.168.56.1	192.168.56.103	DTLSv…	107	Application Data
17	0.498757855	192.168.56.103	192.168.56.1	DTLSv…	99	Application Data
18	0.506861336	192.168.56.1	192.168.56.103	DTLSv…	107	Application Data
19	0.528857924	192.168.56.103	192.168.56.1	DTLSv…	99	Application Data
20	0.531433539	192.168.56.1	192.168.56.103	DTLSv…	107	Application Data
21	0.544949802	192.168.56.103	192.168.56.1	DTLSv…	99	Application Data
22	0.551567263	192.168.56.1	192.168.56.103	DTLSv…	107	Application Data
23	0.569827632	192.168.56.103	192.168.56.1	DTLSv…	99	Application Data
24	0.574318701	192.168.56.1	192.168.56.103	DTLSv…	107	Application Data
25	0.579347058	192.168.56.103	192.168.56.1	DTLSv…	99	Application Data
26	0.583848140	192.168.56.1	192.168.56.103	DTLSv…	107	Application Data
27	0.626489725	192.168.56.103	192.168.56.1	DTLSv…	99	Application Data

```
▸ Frame 30: 107 bytes on wire (856 bits), 107 bytes captured (856 bits) on interface any, id 0
▸ Linux cooked capture
▸ Internet Protocol Version 4, Src: 192.168.56.1, Dst: 192.168.56.103
▸ User Datagram Protocol, Src Port: 5684, Dst Port: 35561
▸ Datagram Transport Layer Security
```

Figure B.2: CoAP/DTLS for Alice as a client



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.56.103	192.168.56.1	WireG…	124	Transport Data, receiver=0x7FE0260A, counte
2	0.034279747	192.168.56.1	192.168.56.103	WireG…	588	Transport Data, receiver=0x2136AFB5, counte
3	0.034447028	192.168.56.1	192.168.56.103	WireG…	192	Handshake Initiation, sender=0x478635E2
4	0.034974043	192.168.56.103	192.168.56.1	WireG…	136	Handshake Response, sender=0x5CB8CB1A, rece
5	0.036550750	192.168.56.1	192.168.56.103	WireG…	76	Keepalive, receiver=0x5CB8CB1A, counter=0
6	0.068246693	192.168.56.103	192.168.56.1	WireG…	124	Transport Data, receiver=0x478635E2, counte
7	0.071142119	192.168.56.1	192.168.56.103	WireG…	588	Transport Data, receiver=0x5CB8CB1A, counte
8	0.080601870	192.168.56.103	192.168.56.1	WireG…	124	Transport Data, receiver=0x478635E2, counte
9	0.084263516	192.168.56.1	192.168.56.103	WireG…	588	Transport Data, receiver=0x5CB8CB1A, counte
10	0.087911458	192.168.56.103	192.168.56.1	WireG…	124	Transport Data, receiver=0x478635E2, counte
11	0.095170337	192.168.56.1	192.168.56.103	WireG…	588	Transport Data, receiver=0x5CB8CB1A, counte
12	0.102330380	192.168.56.103	192.168.56.1	WireG…	124	Transport Data, receiver=0x478635E2, counte
13	0.104994106	192.168.56.1	192.168.56.103	WireG…	588	Transport Data, receiver=0x5CB8CB1A, counte
14	0.110125368	192.168.56.103	192.168.56.1	WireG…	124	Transport Data, receiver=0x478635E2, counte
15	0.111312987	192.168.56.1	192.168.56.103	WireG…	588	Transport Data, receiver=0x5CB8CB1A, counte
16	0.113586675	192.168.56.103	192.168.56.1	WireG…	124	Transport Data, receiver=0x478635E2, counte
17	0.115184008	192.168.56.1	192.168.56.103	WireG…	588	Transport Data, receiver=0x5CB8CB1A, counte
18	0.120561309	192.168.56.103	192.168.56.1	WireG…	124	Transport Data, receiver=0x478635E2, counte
19	0.126603544	192.168.56.1	192.168.56.103	WireG…	588	Transport Data, receiver=0x5CB8CB1A, counte
20	0.134145892	192.168.56.103	192.168.56.1	WireG…	124	Transport Data, receiver=0x478635E2, counte
21	0.136798241	192.168.56.1	192.168.56.103	WireG…	588	Transport Data, receiver=0x5CB8CB1A, counte
22	0.139876470	192.168.56.103	192.168.56.1	WireG…	124	Transport Data, receiver=0x478635E2, counte
23	0.140883778	192.168.56.1	192.168.56.103	WireG…	588	Transport Data, receiver=0x5CB8CB1A, counte
24	0.143038554	192.168.56.103	192.168.56.1	WireG…	124	Transport Data, receiver=0x478635E2, counte
25	0.144080011	192.168.56.1	192.168.56.103	WireG…	588	Transport Data, receiver=0x5CB8CB1A, counte
26	0.146764025	192.168.56.103	192.168.56.1	WireG…	124	Transport Data, receiver=0x478635E2, counte
27	0.147749651	192.168.56.1	192.168.56.103	WireG…	588	Transport Data, receiver=0x5CB8CB1A, counte

```
▸ Frame 45: 588 bytes on wire (4704 bits), 588 bytes captured (4704 bits) on interface any, id 0
▸ Linux cooked capture
▸ Internet Protocol Version 4, Src: 192.168.56.1, Dst: 192.168.56.103
▸ User Datagram Protocol, Src Port: 41194, Dst Port: 41194
▸ WireGuard Protocol
```

Figure B.3: CoAP/WireGuard for Alice as a client

Appendix

Appendix C

C.1 Results

	CoAP	CoAP/DTLS	CoAP/WireGuard
Packets per second	454 ± 41.617	145.584 ± 35.247	355 ± 47.527
Packet size, B	287.500 ± 1.500	106 ± 0	355.500 ± 0.764
Throughput, Mbit/s	1.042 ± 0.092	0.123 ± 0.030	1.009 ± 0.136
Handshake time, ms	–	218 ± 212	1 ± 0.412
Handshake			
Time span, ms	666 ± 63	$2,295 \pm 822$	865 ± 126
RTT, ms	3 ± 0.210	8 ± 2.700	3 ± 0.419
No handshake			
Time span, ms	666 ± 63	$2,078 \pm 617$	865 ± 126
RTT, ms	3 ± 0.210	7 ± 2.100	3 ± 0.157

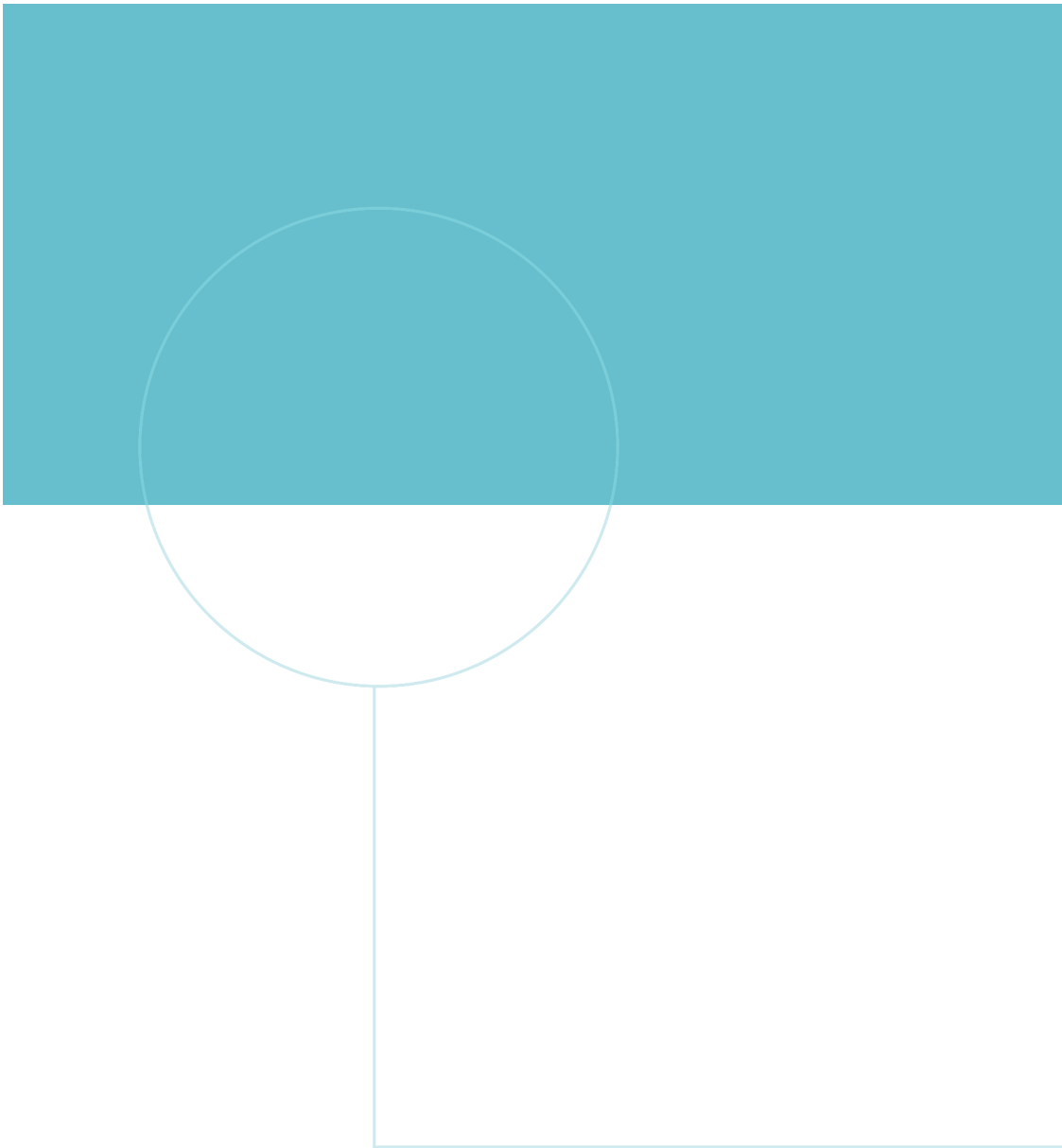
Table C.1: The results for the three implementations of the CoAP protocol obtained during our experiments. Expressed by calculating the arithmetic mean and the standard deviation. (150 GET requests with no delay).

	CoAP	CoAP/DTLS	CoAP/WireGuard
Packets per second	454 ± 41.617	159.740 ± 16.982	355 ± 47.527
Packet size, B	287.500 ± 1.500	106 ± 0	355.500 ± 0.764
Throughput, Mbit/s	1.042 ± 0.092	0.135 ± 0.014	1.009 ± 0.136
Handshake time, ms	–	125 ± 49	1 ± 0.412
Handshake			
Time span, ms	666 ± 63	$1,936 \pm 190$	865 ± 126
RTT, ms	3 ± 0.210	7 ± 0.620	3 ± 0.419
No handshake			
Time span, ms	666 ± 63	$1,811 \pm 171$	865 ± 126
RTT, ms	3 ± 0.210	6 ± 0.570	3 ± 0.157

Table C.2: The results for the three implementations of the CoAP protocol obtained during our experiments, excluding the outlier. Expressed by calculating the arithmetic mean and the standard deviation. (150 GET requests with no delay).

	CoAP	CoAP/DTLS	CoAP/WireGuard
Packets per second	30.050 ± 0.608	28.417 ± 0.267	29.800 ± 0.792
Packet size, B	287.500 ± 1.500	106 ± 0	355.334 ± 0.745
Throughput, Mbit/s	0.069 ± 0.001	0.024 ± 0	0.084 ± 0.002
Handshake time, ms	–	407 ± 112	31 ± 30
Handshake			
Time span, ms	$9,983 \pm 205$	$10,773 \pm 96$	$10,103 \pm 250$
RTT, ms	34 ± 1	36 ± 0	33.680 ± 1
No handshake			
Time span, ms	$9,983 \pm 205$	$10,367 \pm 106$	$10,092 \pm 242$
RTT, ms	34 ± 1	35 ± 0	33.870 ± 1

Table C.3: The results for the three implementations of the CoAP protocol obtained during our experiments. Expressed by calculating the arithmetic mean and the standard deviation. (150 GET requests with 60 ms delay).



NTNU

Norwegian University of
Science and Technology