Master’s Thesis

Efficient Security Protocol for RESTful IoT devices

By

Karnarjun Kantharajan and Sahar Shirafkan
ka7830ka-s@student.lu.se - sa2408sh-s@student.lu.se

Department of Electrical and Information Technology
Faculty of Engineering, LTH, Lund University
SE-221 00 Lund, Sweden

Abstract

In this thesis, we presented comparisons with respect to Energy Consumption, bandwidth, Constraint Application protocol (CoAP) transaction time and throughput for four different security protocols. We simulated and implemented the Datagram Transport Layer Protection (DTLS) version 1.2, Transport Layer Protocol (TLS) version 1.2 & 1.3, and Object Protection for Restricted RESTful Environments (OSCORE). All of the above security protocols allow client / server applications to communicate over the internet with message forgery, eavesdropping, and tampering protection. In particular, we compared the simulation and implementation results of the mentioned protocols to extrapolate the performance of the DTLS version 1.3.

Keywords: Internet of Things (IoT), TLS 1.2 & 1.3, DTLS 1.2, OSCORE, CoAP, Security Protocols.

Popular Science Summary

The Internet of Things (IoT) has become a concept that defines the billions of connected devices that are intelligent. IoT covers everything from connected devices, mobile home products, roadside cameras, production control equipment, medical equipment, vehicles, and more. To drive innovation and improve customer satisfaction, companies use IoT to transform their business and develop new revenue streams. There are three crucial reasons for illustrating the need for security in IoT devices, such as the sheer volume and diversity of applications and data sensitivity. By the early future, there will be an estimated 25 billion IoT devices worldwide, and about 25% of cyber-attacks will target IoT devices. Although many companies can recognize that IoT security is necessary to protect consumers and clients, the problem may rapidly become complicated. As the market continues to evolve, there is a lack of best practices and recommendations for securing the IoT device. Also, the other factor to be addressed when designing an IoT system is the power consumption of the system. With the rise of the Internet of Things, the development of battery-operated devices is a significant aspect that can make a huge difference in the device's efficiency. Device power consumption in IoT is challenging since the device should still be powered up and could be placed anywhere. Mostly, IoT devices are remotely placed and need to use a battery to operate.

Considering the current challenges facing IoT security, in this thesis, we picked four separate security protocols, including TLS 1.2, TLS 1.3, DTLS 1.2, and OSCORE for securing IoT device. To compare the security protocol's efficiency, we simulated and implemented the mentioned security protocol in a real environment. We compared the security protocols with the data transmission time and throughput and security overhead in the simulation by running the client and the server on PC. While in implementation using the Stand-alone multi-radio modules (NINA-W102) as a client, we compared the security protocols with the data transmission time and throughput, security overhead, and even calculated the energy consumption of the IoT unit, which is one of the issues in IoT system's design.

Acknowledgments

I want to express a major thank you to my supervisor at U-blox Malmo, Peter Karlsson, and Hari Vigneswaran for the opportunity and positive help during the study.

I would also like to thank the examiner Thomas Johansson and the LTH supervisor Christian Gehrmann for the advice, feedback, and written orders they have provided.

Sahar Shirafkan and Karnarjun Kantharajan

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Background

The Internet of Things (IoT) is the new global connectivity paradigm allowing billions of devices to communicate among themselves and with the rest of the Internet. Hence, IoT security is one of the top research topics. IoT has three layers consisting of layers of perception, network, and application. Security at the application layer offers an appealing alternative to secure applications on the Internet of Things (IoT), especially where protection of transport layers is not adequate or where safety needs to operate through a range of underlying protocols. A variety of safety standards can be used in each layer to achieve a reliable realization of IoT. Many new IoT protocols have been released, aimed at protecting critical data such as Datagram Transport Layer Security (DTLS) [1], Transport Layer Security (TLS) [2], and Object Security for Constrained RESTful Environments (OSCORE) [3] and Ephemeral Diffie-Hellman Over COSE (EDHOC) [4]. IoT devices can be restricted in various ways including memory, storage, processing capacity, and energy, so finding the most efficient security protocols for RESTful IoT units is an important issue.

The Constrained Application Protocol (CoAP) is a specialized Internet Application layer Protocol, as specified in [9] for constrained devices. The lightweight protocol CoAP is intended to be used and considered as a replacement of HTTP for being an IoT application layer protocol. This allows certain constrained devices called "nodes" to connect using common protocols to the broader Internet. Also, it is designed to be used between devices on the same constrained network (e.g., low-power, loss networks), between devices and general nodes on the

Internet, and between Devices on different constrained networks that are also linked to the Internet [10]. CoAP is also used by other channels, for example, SMS on mobile communication networks. CoAP is a service layer protocol intended for use in resource-constrained internet applications, such as network nodes with wireless sensors. It can run on most devices supporting User Datagram Protocol (UDP) or Transmission Control Protocol (TCP).

The TLS protocol's primary aim is to provide authentication, confidentially and integrity protection between two communicating peers. TLS runs over the transport layer protocol and generates security services for application layer protocols. TLS requires a connection-oriented transport channel-usually TCP. The protocol is released in different versions and has been upgraded throughout the years. The very first version was TLS 1.0 which was released in 1999 [21], TLS 1.1 was released in 2006 [22], and TLS 1.2 in 2008 [23]. The several weaknesses found in TLS 1.2 and below, as well as the growing demand to enhance protocol efficiency, motivated by introducing the next version of the protocol, TLS 1.3, in the spring of 2014. The Datagram Transport Layer Security (DTLS) protocol has been developed for applications that use UDP as a transport layer to provide secure communication between peers who communicate. DTLS is intentionally designed to be as similar as possible to TLS, both to eliminate innovation in protection and to increase the amount of reuse of code and infrastructure. The DTLS protocol has also been releases in different versions. DTLS 1.0 that was originally defined as a delta from TLS 1.0, DTLS 1.2 [5] was defined as a series of deltas to TLS 1.2 [6] and DTLS 1.3 protocol is based on the Transport Layer Security (TLS) 1.3 protocol and provides equivalent security guarantees.

OSCORE is a method for application-layer security of CoAP, using Concise Binary Object Representation (CBOR) a method for protecting individual messages at the application layer suitable for constrained devices is provided by CBOR Object Signing and

Encryption (COSE) [16]). OSCORE provides end-to-end protection between endpoints that communicate via CoAP or CoAP-mappable HTTP. This method is designed for constrained nodes and networks. OSCOE uses a small message size offering low complexity implementation as well as low memory requirements [15].

1.2 Problem

IoT security is characterized by a high-priority research interest as it is an evolution of the conventional, unsecured Internet paradigm where communications in the digital world reach the physical world. IoT systems often deal with personal information, valuable business data, and actuators interacting with the physical world. Not only do such systems need security and privacy, they often need end-to-end protection with source authentication and perfect-forward secrecy. In particular, IoT security frameworks must tackle conventional networking attacks and, at the same time, provide safe communications for all forms of interactions like human-to-machine and machine-to-machine. User data is protected by security protocols such as TLS, DTLS, OSCORE, and EDHOC. The selection of efficient security protocols for IoT devices is a critical issue as IoT devices can be restricted in various ways including memory, storage, processing capacity, and energy. Also, an important risk of IoT systems is cryptographic key exposure [7]. Network nodes can be physically open to attackers, so securing keys and collected data on the server end is also critical, as it is typical for IoT systems to collect a large amount of sensitive data.

There are lots of challenges that security protocols have to address in general like per-packet message size, overheads, transmission times, and power consumption. The message size of a key exchange protocol can have a major impact on the performance of an IoT device, particularly in noisy environments that show the need to have a security protocol with a small key exchange message size. In addition, the power usage of wireless devices is highly dependent on the transmitting, listening and receiving of messages, which indicates the need to use the appropriate security protocol depending on the transmitting of the data byte [4]. In this thesis, we will evaluate and test the per-packet message size overheads, transmission times, and power consumption for TLS 1.2 & 1.3, DTLS 1.2, and OSCORE that run above CoAP, to get a good view of which security protocol is the most efficient for IoT devices. The purpose for including security protocols above CoAP is that we are going to evaluate the performance of DTLS 1.3 from the results of the security protocols listed above.

1.3 Methodology

The thesis project will be based on the discovery of efficient security protocols in power and bandwidth for IoT devices. We are coding the software required for the embedded IoT device and perform power and overhead measurements on the IoT device. Software tools such as the Visual Studio and the Eclipse IDE are used to simulate security protocols, and Wireshark is used as a network analyzer. The open-source JAVA code of Californium is used for servers of CoAP, DTLS, and OSCORE. The client is coded in the Visual Studio for the CoAP, TLS 1.2 & 1.3, DTLS 1.2, OSCORE, and Wireshark for viewing the packet exchange. These security protocols are implemented into a NINA-W10 device to measure the power efficiency and overhead and compare the results to choose the appropriate one.

1.4 Outline

In this thesis chapter, 1 consists of the basic introduction of the thesis. It also contains Thesis Problems, Methodology, and Outline. Chapter 2, is an overview of IoT devices, considered application protocol, and security protocols. Chapter 3, is presenting the simulation of considered security protocols and the output results of simulation. Chapter 4, is representing the implementation of the mentioned security protocols over CoAP and the results of implementation.

Finally, Chapter 5 concludes the implementation and simulation of this thesis work.

CHAPTER 2

Overview of considered security protocols

2.1 The CoAP protocol

Constrained Application Protocol (CoAP) [9] is a light application layer protocol for constrained nodes and networks in IoT devices. CoAP with different request/response methods supports interaction between application endpoints with low overhead, multicast support, and simplicity for constraint environment.

Representational State Transfer (REST) is a software design style that specifies a series of constraints to be used when developing web services. Web services that fit with the REST architecture form, called RESTful Web Services. The aim of the Constrained RESTful Environments (CoRE) work is to implement the REST architecture in an acceptable form for the most constrained nodes and networks. One solution for the REST architect's deployment of constrained devices is the fragmentation of packets, which has the downside of reducing the throughput. CoAP comes with a new approach in REST architect deployment for restricted devices as it eliminates the need for fragmentation while keeping the overhead small. Important features of CoAP are:

- Fulfilling M2M requirements in constrained environments.
- Low header overhead and parsing complexity.
- URI and Content-type support.
- Simple proxy and caching capabilities.
- Functionality of mapping to HTTP and operating with protocols that are based on HTTP.
- Ability of binding to UDP and security protocols like DTLS.

In addition, CoAP must also be implemented through reliable transport, such as TCP or Transport Layer Security (TLS), in some situations, such as when networks do not forward UDP packets or are rate-limiting UDP traffic.

The CoAP protocol is dividing into two layers like Requests/Responses and Messages Figure 1. The next sections will introduce these layers.

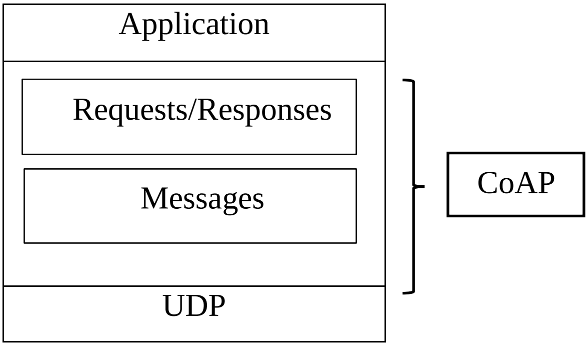
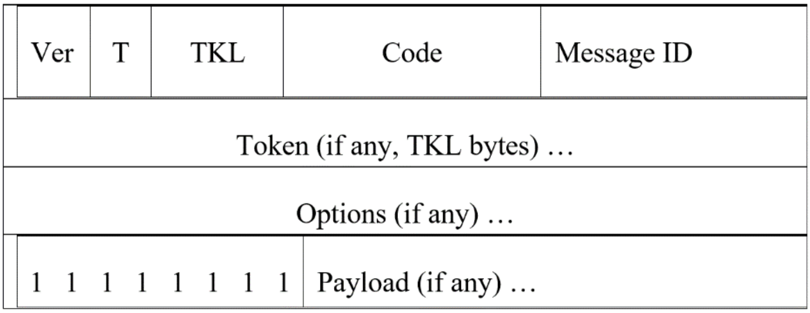


Figure 1: CoAP Layers

2.1.1 Message Layer

The CoAP messaging model is based on the transfer of messages between endpoints via UDP/TCP.

The CoAP message format includes fixed-size 4-byte header, variable-length Token value, options, and payload shown in Figure 2.



<table>
<tr><td>Ver</td><td>T</td><td>TKL</td><td>Code</td><td>Message ID</td></tr>
<tr><td colspan="5">Token (if any, TKL bytes) …</td></tr>
<tr><td colspan="5">Options (if any) …</td></tr>
<tr><td colspan="3">1 1 1 1 1 1 1 1</td><td colspan="2">Payload (if any) …</td></tr>
</table>

Figure 2: CoAP Message Format

CoAP message header includes:

- Version (Ver): This field is 2-bit unsigned integer shows the CoAP version.
- Type (T): This field is 2-bit unsigned integer shows message type. There are four different message types for CoAP like Confirmable message (CON), Non-Confirmable message (NON), Acknowledgement message (ACK), and Reset message (RST). An ACK and RST are CoAP server response type where ACK message recognizes the arrival of a particular Confirmable message and RST message shows the missing of some context in CON and NON. CON and NON messages are CoAP Request/Response methods. CON is for showing the reliability of a message and it needs ACK. NON is a message that does not require reliable transmission and ACK but it has a duplication identification Message-ID.
- Token Length (TKL): This field is 4-bit unsigned integer shows the length of the Token field (0-8 bytes).
- Code: This field is 8-bit unsigned integer, which is explained in the section 2.1.2.
- Message ID: This field is 16-bit unsigned integer which is used to match messages of types CON/NON with ACK/RST and detecting message duplication.

The second part of the message format is the Token value used to correlate requests and responses, which can be between 0 to 8 bytes long. The next field is filled when there are CoAP options otherwise, it is a sequence of zeros. Finally, the last part is filled with an optional payload. One-byte Payload Marker (0xFF) shows the payload's presence, and without this marker, the payload is zero.

The Message Layer is responsible for reliability and sequencing with different types of CoAP messages like CON, NON, ACK.

2.1.2 Request/Responses Layer

In the request/response layer, the CoAP client sends one or more CoAP requests to the server. The server that receives the Request will reply with the CoAP Response. The Request and the Response are exchanged asynchronously via CoAP messages. The CoAP message carries a method code or Response code. Also, the CoAP message carries some optional Request and Response information like URI, payload media type, and Token to match requests and responses. The CoAP Request methods are:

- GET: The GET method is used to obtain information that currently corresponds to the resource defined by the URI request.
- POST: The POST method recommends that the description used in the Request be processed.
- PUT: The PUT method recommends that the resource specified by the URI requirement be changed or generated with the enclosed representation.
- DELETE: The DELETE method recommends that the resource specified by the URI request be removed.

Based on the CoAP request methods, CoAP response codes are:

- 2.xx (Success): This code indicates that the client Request received, understood, and accepted successfully. Where the last two. xx denotes: .01 (Created), .02 (Deleted), .03 (Valid), .04 (Changed), and .05(Content).
- 4.xx (Client Error): This code shows that the server did not understand the request. Where the last two .xx denotes .00 (Bad Request), .01 (Unauthorized), .02 (Bad Option), .03 (Forbidden), .04 (Not Found), .05 (Method not Allowed), .06 (Not Acceptable), .12 (Precondition Failed), .13(Request entity too large), and .15 (Unsupported Content-Format).
- 5.xx (Server Error): This code shows server error where the last two .xx indicates: .00 (Internal Server Error), .01 (Not Implemented), .02 (Bad Gateway), .03 (Server Unavailable), .04 (Gateway Timeout), and .05 (Proxy Not Supported).

In this thesis, the CoAP Request and Response Carried in Confirmable Message (CON) was selected as seen in Figure 3:

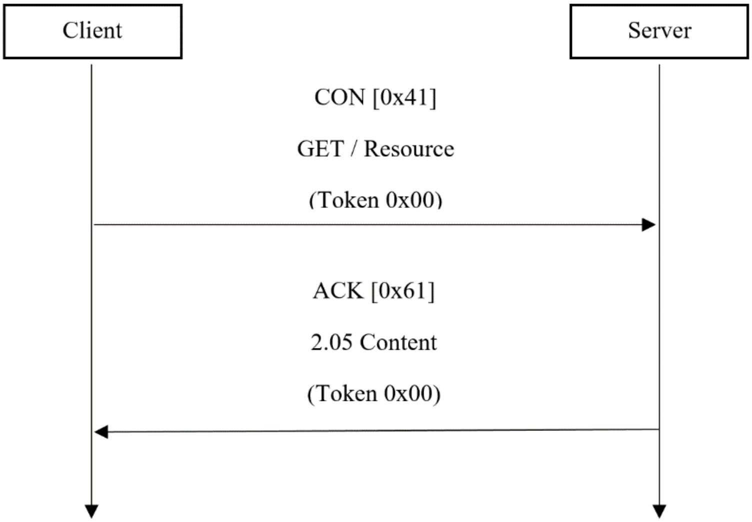


Figure 3: CoAP Message Transmission

2.2 The DTLS Security Protocol

Many techniques are used to secure network traffic. Transport Layer Security (TLS) [2] is one such technique that is the most widely used protocol for securing email and web traffic. It operates in a transparent connection-oriented channel and runs over reliable transport channels such as Transmission Control Protocol (TCP). Over the past few years, the usage of the User Datagram Protocol (UDP) has increased in many application protocols. The CoAP protocol is used for communication in IoT devices operating over UDP and TCP. There is also a need for a TLS compatible datagram variant. To mitigate innovation on security and to increase the amount of reuse of code and infrastructure IETF has proposed Datagram Transport Layer Security (DTLS) [1] [5].

Unreliability in TLS causes problems at two levels i.e.

1. Individual records are not independently decrypted by TLS, so if record N is not obtained the integrity check is on the sequence number, then the integrity check on record N+1 will be based on the incorrect sequence number and will thus fail.

2. If the messages are lost, the TLS handshake layer assumes that the handshake messages are delivered reliably and breaks.

For securing the communication and preventing eavesdropping, tampering, message forgery between the two different peers, the Datagram Transport Layer Security (DTLS) protocol can be used. Datagram transports applications include media streaming, Internet telephony, and online gaming for communication. All these applications are characterized by being delay-sensitive. Applications with such behaviors are unchanged when DTLS protocol is used for securing communication since the DTLS protocol does not compensate for lost or reordered data traffic. It is designed to run in application space and doesn’t need any kernel modifications. As discussed in Section 1.1, there are different versions of DTLS, this thesis focuses on DTLS 1.2. Below, we discuss how DTLS handles the different problems related to datagram transport.

2.2.1 Providing Handshake Reliability

In TLS, messages are mismatched and produce errors if the order is not defined correctly. So, messages must be defined in the order. This is incompatible with reordering and message loss. Also, TLS handshake messages create a problem of IP fragmentation for sending over datagram, as these messages are larger than the datagram's size. DTLS provides fixes for these two problems.

2.2.2 Loss of Packets

DTLS uses a retransmission timer to fix the issue of packet loss. The client sends the client hello message to the server during the initial process of the DTLS handshake and hopes to receive a hello verify request from the server. When the client does not receive the hello verify request within the specified period then the timer expires and the client knows that the request has been lost either to the client, hello, or from server hello. The client retransmits the message and retransmits it when the server receives the retransmission. The server also has the retransmission timer, and when the timer ends, it retransmits. For hello verify request, the timeout and retransmission do not apply. The hello verify request is designed to be small enough not to be broken by itself, thus eliminating the issues of multiple hello verifying requests.

2.2.3 Message Re-ordering

A specific sequence number has been assigned to handshake messages within that handshake in DTLS. The receiver, which receives this handshake, regulates the next upcoming message, which is as expected, or not. If the received message is not the same then it is put up in the queue for future handing; else if the message is as expected it proceeds with further processing.

2.2.4 Length of the Message

In DTLS and TLS, the length of the handshake messages is actually larger when it is compared with the length of the UDP datagram. DTLS handshake messages are fragmented into separate DTLS record layers. Each recording layer is intended to fit in a single IP datagram and this is the solution to solve the issue. Fragment offset and length consist of individual handshake messages. Hence, the receiver occupies all the bytes of handshake message and reassembles the original unfragmented message.

2.2.5 The DTLS Handshake Protocol

Seeing from Figure 4, DTLS uses almost the same handshake and flow communications as TLS, except for three critical modifications:

1. To avoid Denial of Service (DoS), a stateless cookie exchange has been added in DTLS.

2. Modifications have been made in the DTLS handshake header to handle message loss, reordering, and IP fragmentation.

3. To handle message loss, a retransmission timer has been added in DTLS.

In addition to the examples mentioned above, DTLS message formats flow, and logic is similar to TLS.

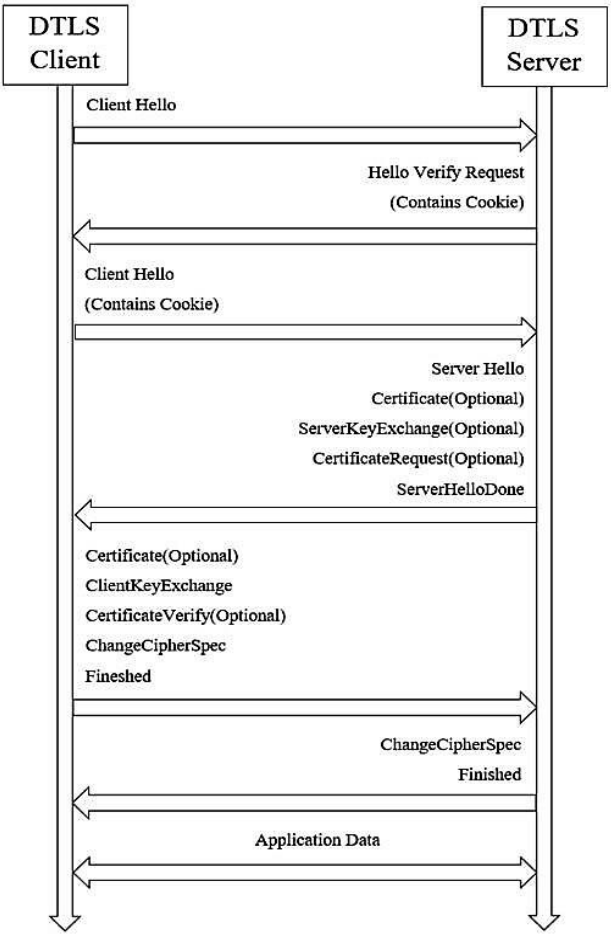


Figure 4: DTLS Handshake

2.2.6 Countermeasures on denial of service

A Denial-of - Service (DoS) attack is an attack designed to lock down a system or network, making it difficult for the intended users to reach it. DTLS contains two types of DOS attacks that are of major concern:

1. By transmitting a series of handshake initiation requests, an attacker can consume excessive resources on the server, this causes the server to perform expensive cryptographic operations and allocate TLS session state data.

2. By using the server as the amplifier, an attacker can send a connection initiation message with a forged source of the victim.

DTLS uses stateless cookie technique to protect the system against these two types of DoS attacks. When the client sends the client hello to the server, the server will respond with a hello verify request containing a stateless cookie generated by using the technique of PHOTURIS [28]. The client then responds back with client hello adding the cookie. Then the server verifies the cookie and proceeds with the handshake only if the cookie is valid. DoS attacks with spoofed IP addresses can be potentially stopped using this mechanism since it forces the attacker/ clients to be received with cookies; but still this method does not guarantee any defense against DoS attack with a valid IP address.

2.2.7 Cipher Suites

The cipher suite [41] is generated with a group of algorithms to secure the network connections, which uses TLS. As DTLS is based on TLS, the cipher suite used for DTLS in this thesis is TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256. The mentioned cipher suite uses authenticated encryption with additional data algorithm AEAD_AES_128_GCM, and it is the combination of authentication, encryption, and message authentication code (MAC) algorithms. AEAD is a form of encryption that provides confidentiality for the plaintext and a way to check its integrity and authenticity.

Three parts of this cipher suite include:

- ECDHE_RSA algorithm uses ephemeral elliptic curve Diffie-Hellman to exchange keys. During a handshake, the key exchange algorithm is used to decide whether and how the client and server can authenticate.

- AES_128_GCM with 128 bits is used to encrypt the message stream with a block cipher.
- SHA256, a message authentication algorithm, is used for ensuring message integrity.

2.2.8 Certificate

While analyzing the DTLS handshake, the certificate is divided into two parts, like Client and Server Certificate. The client certificate acts as a way for the end-user to claim their identity on the server, and the server certificate verifies and validates the certificate holder's identity before authenticating it. Implementations are responsible for verifying certificate integrity and should generally support messages for certificate revocation. Certificates will also be checked by a reputable Certificate Authority (CA) to guarantee a correct signature. Selecting and introducing trustworthy CAs will be handled with considerable care. Users should be able to view certificate information and root CA information.

2.3 The TLS 1.3 Security Protocol

TLS version 1.3 is modified version of TLS 1.2 with some important improvements. In this section, we will address the major difference between TLS 1.3 [43] and TLS 1.2. TLS 1.3 has increased security and speed. The list of main functional differences is as follows:

- All algorithms that are considered legacy have been removed from the list of approved symmetric algorithms, and the remaining algorithms are using Authenticated Encryption with Associated Data (AEAD).
- Since all public-key based key exchange mechanisms now provide forward secrecy, the Static Diffie-Hellman cipher suite has been removed.

- In the TLS 1.2 handshake, messages after Server Hello were not encrypted, however in TLS 1.3, all handshake messages after Server Hello were encrypted.
- The Extract-and-Expand Key Derivation Function (HKDF) based on HMAC is used as a key deviation function. This newly introduced function has improved key separation properties that make it easier for cryptographers to analyze.
- For more consistent and needless messages, such as Change Cipher Spec, The handshake state machine has been substantially restructured.
- Due to the reduction in the handshake the speed has been improved.
- TLS 1.3 just need only one round-trip time before the client sends the application data. Whereas the older version of TLS requires two round-trip time. Also, the server will send the application data in response to the client's first handshake reply. This means that network latency has less effect on the time taken to create a stable link.

Figure 5 shows the handshake of TLS 1.3.

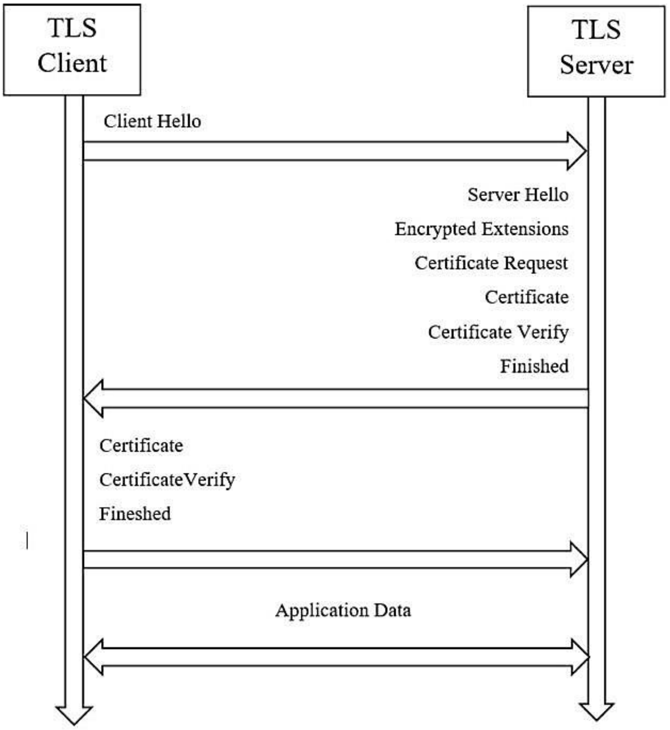


Figure 5: TLS 1.3 Handshake

2.4 The OSCORE Security Protocol

Object Security for Constraint RESTful Environments (OSCORE) protocol is an application-layer security method for the Constraint Application Protocol (CoAP) by means of CBOR Object Signing and Encryption (COSE). CBOR is a data format designed for small code size and small message size, which modified the JavaScript Object Notation (JSON) data model by allowing for binary data, among other changes. CBORE is used for compact encoding in OSCORE. The COSE structure arranges all of the security messages based on the CBOR array type, which is used for encryption and key derivation structures. In Figure 6 below, we give a schematic overview of the message exchanges of OSCORE when used by CoAP.

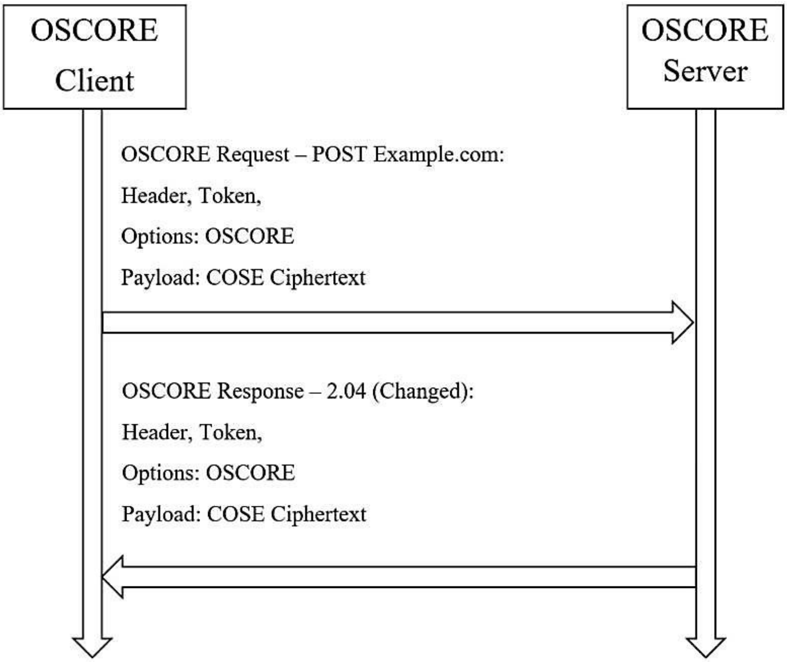


Figure 6: OSCORE Handshake

OSCORE can be used for both unreliable and reliable transport as these methods differ only in the CoAP messaging layer, which is not protected with OSCORE. In addition, OSCORE protects the RESTful interactions like the request method, the requested resource, and the payload of the message that is shown in Figure 7. As OSCORE protects only the relevant application layer information, the message overhead is minimal.

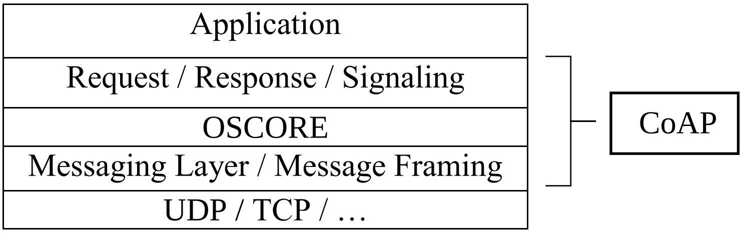

Application
Request / Response / Signaling
OSCORE
Messaging Layer / Message Framing
UDP / TCP / …
CoAP

Figure 7: CoAP + OSCORE Layers

OSCORE protects the plaintext of CoAP messages. Not all CoAP fields are equally protected but fields are separated into protected and unprotected fields. Table 1 below, gives an overview of how the CoAP header and payload fields are protected with OSCORE.

Field	Encrypt and Integrity Protect	Encrypt and Integrity Unprotect
Version		x
Type		x
Length		x
Token Length		x
Code	x	
Message ID		x
Token		x
Payload	x	

Table 1: Protection of CoAP Header Fields and Payload [18]

OSCORE can provide end-to-end protection between endpoints including CoAP-to-CoAP, HTTP-to-CoAP, and CoAP-to-HTTP proxies.

2.4.1 The OSCORE Option

The OSCORE header option indicates that the CoAP message is protected by the OSCORE security protocol, and it contains the compressed COSE object. The Object-Security option is critical, safe to forward, part of the cache key, not repeatable.

In Figure 8, the OSCORE header option includes OSCORE flag bits that occupy the first byte, Partial IV parameter that occupies n bytes, the kid context flag that occupies 1 byte to encode the length of the flag, s bytes to encode the kid context, and the remaining bytes to encode the kid's value. Where h is the kid context flag and k is the kid flag.

0 1 2 3 4 5 6 7 <---------- n bytes ----------> 0 1 2 3 4 5 6 7 < ------- s bytes ------->

0 0 0	h	k	n	Partial IV (if any)	S (if any)	Kid context (if any)	Kid (if any)

Figure 8: OSCORE Header Options

2.4.2 OSCORE Security Context

The security context is a set of parameters that link the security protocol to the environment and allow the server and the client to interact. OSCORE uses pre-shared keys that may have been generated out-of-band or with a key setup protocol that requires that the client and server establish a shared security context used to process the COSE objects. The security context is the set of elements of data necessary to perform cryptographic operations in OSCORE. There are three types of security contexts:

1. The Sender Context, which is used to secure the messages to be sent, includes:

a) Sender ID:

Sender ID is bytes of string used to identify the sender context, derive AEAD keys, Common IV, and to ensure unique AEAD nonces. AEAD Algorithm determines the maximum length of the sender ID, and it should be pre-established.

b) Sender Key:

Sender Key is bytes of a string containing the symmetric AEAD key to protect messages to send. It is derived from Common Context and Sender ID. AEAD Algorithm defines Sender key length.

c) Sender Sequence Number:

Sender Sequence Number is a non-negative integer used to list requests and certain responses where the AEAD Algorithm determines the maximum value of it.

2. The Receiver Context, which is used to confirm the messages received, includes:

a) Recipient ID:

Recipient ID is bytes of string used to identify the recipient context, derive AEAD keys, Common IV, and to ensure unique AEAD nonces. AEAD Algorithm determines the maximum length of it, and the value of the recipient ID should be pre-established.

b) Recipient Key:

Recipient Key is bytes of a string containing the symmetric AEAD key to verify messages received. It is derived from the Common Context and Recipient ID, and the AEAD Algorithm defines its length.

c) Replay Window:

Replay Window is on the server-side to evaluate requests received. DTLS-type replay protection and a window size of 32 will use as default values for Replay Window parameters.

3. The common context from which all contexts derive includes:

a) Authenticated Encryption with Associated Data (AEAD) Algorithm:

The COSE AEAD algorithm, which is used for encryption, has the default value of AES-CCM-16-64-128 (COSE encoding algorithm: 10).

b) HKDF Algorithm:

An HMAC-based key derivation function (HKDF), which is used to derive Sender Key, Recipient Key, and Common IV, has the Default value of HKDF SHA-256.

c) Master Secret:

Master Secret contains variable length and random byte string to derive AEAD keys and Common IV. The master secret must be pre-configured into the peers.

d) Master Salt:

Optional byte string with variable length containing the salt used to derive AEAD keys and Common IV with the default value of empty byte string.

e) ID Context:

Optional variable-length byte string providing additional information to identify the Common Context and to derive

AEAD keys and Common IV. The use of ID Context is described in.

f) Common IV:

Byte string derived from Master Secret, Master Salt, and ID Context. They are used to generate the AEAD Nonce.

Finally, some OSCORE differences with other security protocols include key negotiation and session management as it protects each payload with a pre-shared key, unlike (DTLS) which protects hop-by - hop messages, OSCORE protects the payload message with an end-to - end protection process. As a result, we can save power, bandwidth and computational resources. In addition, OSCORE enables translation of the HTTP-CoAP protocol at a gateway or a proxy that enables the use of OSCORE over TCP.

CHAPTER 3

Simulation Results

In this chapter we explain the simulation test we have performed for the different security protocols, such, we investigate, i.e., TLS 1.2, TLS 1.3, DTLS 1.2, and OSCORE over CoAP. We compared the security overhead, handshake time/throughput, and CoAP transaction (CoAP GET Request + Response) time/throughput for these selected protocols. The tests were performed using Visual Studio as a development environment on windows 32/64 bit processor platform, where we simulate both client and server on the same PC with the local IP address using the open sources like Contiki-NG and WolfSSL Libraries. All tests were simulated for 20 iterations, and the results reported are the average values.

3.1 Simulation Set-Up and Achieved Results

In this thesis, the transfer of CoAP packet between client and server is simulated with different security protocol selections. We utilized open sources libraries to build the simulations. WolfSSL [33], and Contiki-NG [34] was used as main security protocol implementation libraries. The WolfSSL library is a C-language-based SSL/TLS library designed for IoT, embedded, and RTOS environments. The advantages of this library, among other open sources, are the size, speed, feature set, and ability to support TLS 1.2, TLS 1.3, and DTLS 1.2. Since the WolfSSL library does not support OSCORE, a branch of Contiki-NG [34] open source was used for simulating OSCORE. Contiki-NG is an open-source library used for next-generation IoT applications, which focuses on low-power communication and standard protocols. Visual Studio Community Version 2019 was used to simulate the server and client on the same Computer with the Windows 32/64 bit processor and the specified libraries. Finally, Wireshark tool was used to capture the packets and analyze security performance. Figure 9 is the simulation set-up:

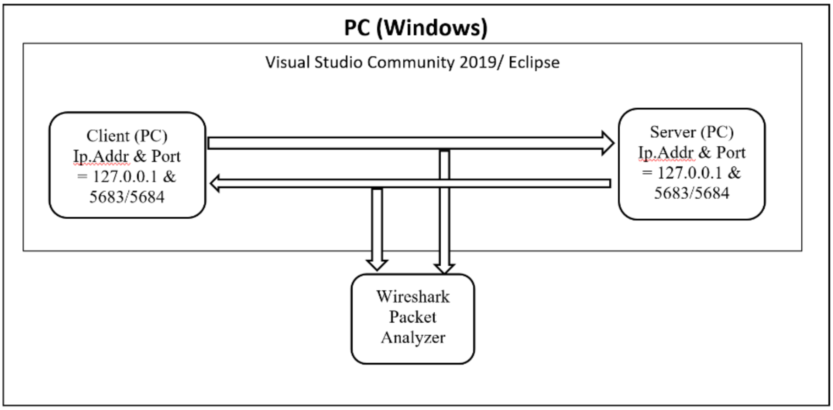

PC (Windows)
Visual Studio Community 2019/ Eclipse
Client (PC) Ip.Addr & Port = 127.0.0.1 & 5683/5684
Server (PC) Ip.Addr & Port = 127.0.0.1 & 5683/5684
Wireshark Packet Analyzer

Figure 9: Simulation Set-Up

3.2 TLS 1.2 Simulation Results

WolfSSL open source master version was used to simulate the TLS 1.2 client and server. A WolfSSL library was built using Visual Studio Community Version 2019. WolfSSL provides the user-settings.h header file to enable protocols and methods. This we utilized in the simulations. No changes to the user-setting.h file was needed for the TLS 1.2 simulation. In the simulations, we run the CoAP GET request with 7 bytes length and the responses from the server were set to the CoAP GET response with the size of 17 bytes and 94 bytes.

We run the simulations with the client and server default certificate (./certs/server-cert.pem) and the default key file (./certs/server-key.pem). It would have been possible for us to use another certificate, but as we are mainly interested in making a performance test, the default certificate fulfills our requirements. The cipher suite used during the SSL handshake for TLS 1.2 was TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256, a combination of authentication, encryption, and message authentication code (MAC) algorithms. This cipher suite was chosen from the list of Elliptic Curve Cryptography (ECC) cipher suites, which are supported by the WolfSSL library. This is a cipher suite recommended for many cloud servers as it has a good security level small overhead in with respect to TLS record layer header, encryption algorithm padding, and the MAC tag. We run TLS with the configurations and commands described in Appendix A.

For analyzing the performance of TLS 1.2, we calculated four different performance indicators:

- Handshake Time Analysis
- CoAP Transaction Time and Throughput Analysis
- Security Overhead

Handshake time is calculated using Wireshark tool analyzer from the first Client Hello until end of the handshake. For TLS 1.2 the average handshake time achieved for 20 iterations are 174.8 ms for total of 13 packets.

The following equation is used to calculate the CoAP Transaction throughput:

$$\text{Throughput} = \frac{\text{data}*8}{\text{Time}} \ \text{(bps)} \qquad (1)$$

In Equation (1), the data is the total length of input data achieved from the CoAP Request and Response size. To calculate the time and throughput for the CoAP transaction, we were simulating 20 iterations with 500 GET Request/Response and two different response size to check the efficiency. For the 7 bytes GET request with a response size of 17 bytes, the achieved average time and throughput for one CoAP transaction is 2.59 ms and 74.131 kbps. Similarly, for 94 bytes response size, the average time and throughput are 4.35 ms and 185.747 kbps.

The security overhead was measured from Wireshark by the difference between the length of the record layer of the application data and the CoAP packet length for the GET request/response. Therefore, the security overhead for TLS 1.2 is 29 bytes.

3.3 TLS 1.3 Simulation Results

The simulation set-up used for TLS 1.3 was the same as those for TLS 1.2. Furthermore, we used the same data and certificates are used as for TLS 1.2 simulations. The different configurations steps needed are listed in Appendix A.

We used a different cipher suite for TLS 1.3, as the handshake is different from the old TLS versions. In TLS 1.3, all encryption and authentication algorithms are combined in authenticated encryption with associated data (AEAD) encryption algorithm. Therefore, static RSA and Diffie-Helman cipher suites have been removed to make the protocol more secure. So, the TLS_AES_ 128_GCM _SHA256 cipher suite was chosen among the TLS 1.3 supported cipher suites.

When we analyzed the performance of TLS 1.3, we used the same measurements parameters as for the TLS 1.2 runs.

For handshake time analysis, the average time achieved for 20 iterations was 143.33 ms for total of 10 packets.

For CoAP transaction time and throughput analysis, the average time and throughput for one CoAP transaction was 2.19 ms, and 87.671 kbps, which is for 7 bytes GET request with a response size of 17 bytes. Similarly, the average time and throughput are 4.175 ms and 193.532 kbps for the response size of 94 bytes. Moreover, the security overhead for TLS 1.3 is 22 bytes.

3.4 DTLS 1.2 Simulation Results

The simulation set-up we used was the same as mentioned in Section 3.1. In addition, the same data, certificates, and cipher suite are used as TLS 1.2 as we want to compare security protocols. The different configurations steps needed are listed in Appendix A.

For analyzing the performance of DTLS 1.2, the same parameters are measured, as mentioned in section 3.2.

For handshake time analysis, the average time achieved for 20 iterations was 169.55 ms for 14 packets.

For CoAP transaction time and throughput analysis, the average time and throughput for one CoAP transaction was 2.24 ms, and 85.714 kbps, which is for 7 bytes GET request with a response size of 17 bytes. Similarly, the average time and throughput are 4.32 ms and 187.037 kbps for the response size of 94 bytes. Moreover, the security overhead for DTLS 1.2 is 37 bytes.

3.5 OSCORE Simulation Results

Contiki-NG [34] was used for the simulation of OSCORE with the same CoAP packet as mentioned in section 3.1. Eclipse/Californium was used as an OSCORE server that is running on port 5683 for the simulation part. The input parameters we used for establishment of security context between client and server are the following:

- AEAD Algorithm is AES-CCM-16-64-128

  As OSCORE uses an untagged COSE Encrypt0 structure with an Authenticated Additional Data Encryption (AEAD) Algorithm for encryption, the AES-CCM-16-64-128 algorithm suggested with the OSCORE IETF draft was used in our tests. The reason for selecting the AES-CCM method in OSCORE was that the message authentication is done on the plaintext, compared with the GCM method, which is for message authentication on the ciphertext. In the selected algorithm, the authentication field's size is 64 bits, the size of the length field is 16 bits, and the length of the Sender Key and Recipient Key is 128 bits.

- Master Salt and Master secret

  Master Salt and Master Secret for derivation of AEAD algorithm are set to the same values of the OSCORE server in our tests.

- HKDF Algorithm was HKDF SHA-256.

  HKDF Algorithm was used as a key derivation algorithm in OSCORE with the IETF draft's default method, which is SHA-256, and it is similar to other protocols in our tests.

- Replay Window size

  Replay Window size is selected as the default value that is DTLS-type replay protection with a window size of 32 in the IETF draft.

- Sender ID and Recipient ID

  To drive the communication between client and server, the client's SID value should be matched with the server's RID value, which is chosen as SID = {0x01} and RID = {0x02} in our tests. Also, to keep the packet size minimum, we selected one byte SID and RID.

When we analyzed the performance of OSCORE in simulation, three performance parameters were calculated:

- Time and Throughput for Packet Exchange
- Security Overhead

To calculate the time and throughput for packet exchange, we were simulating 20 iterations with 500 CoAP + OSCORE GET Request/Response and two different response size to check the efficiency. The 7 bytes CoAP GET request with a response size of 17 bytes for one packet exchange, the achieved average time, and throughput for one packet exchange was 0.593 ms and 323.777 kbps. Similarly, for 94 bytes response size, the average time and throughput are 0.661 ms and 1222.39 kbps. Moreover, the request overhead was 14 bytes, and the response overhead was 10 bytes.

3.6 CoAP Simulation Results

To evaluate and compare the performance parameters described in section 3.5, we simulated a CoAP transaction without the addition of a security protocol. The open-source Contiki-NG was used to simulate the CoAP client, and the UDP socket was used as the server.

For analyzing the performance of CoAP in simulation, two performance parameters were calculated:

- Time and Throughput of the CoAP transaction

To calculate the time and throughput for packet exchange, we were simulating 20 iterations with 500 CoAP GET Request/Response and two different response size to check the efficiency. For the 7 bytes CoAP GET request with a response size of 17 bytes for one packet exchange, the achieved average time and throughput are 0.342 ms and 397.66 kbps. Similarly, for 94 bytes response size, the average time and throughput are 0.633 ms and 1276.461 kbps.

3.7 Simulation Summary

Figure 10 shows the handshake analysis's simulation results for the three security protocols, TLS 1.2, TLS 1.3, and DTLS 1.2. It is clear that the average time for TLS 1.3 outperforms comparing with the other two protocols, and it will affect the performance when there is a need to send data with multiple handshakes.

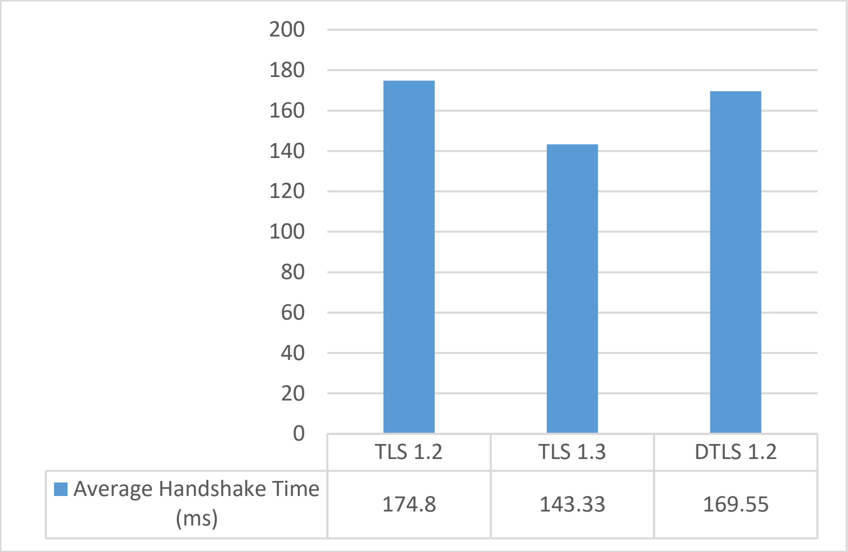
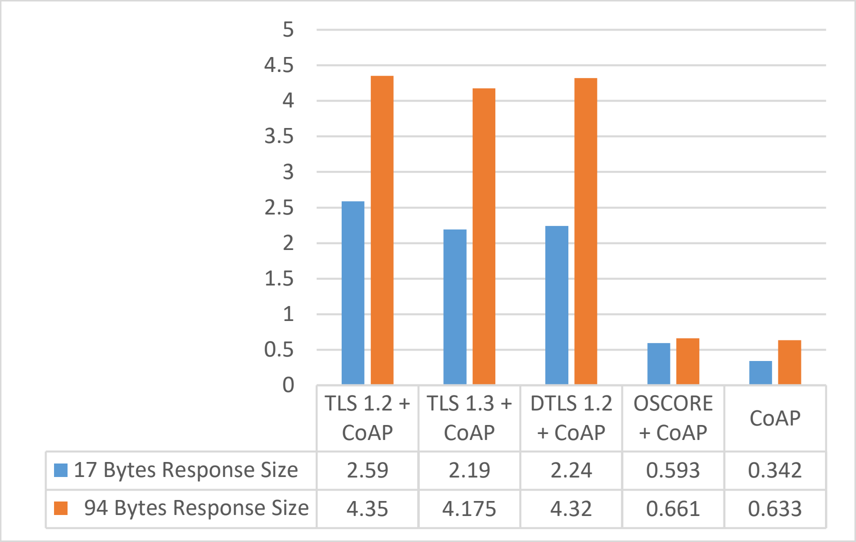


Figure 10: Simulation Handshake Time (ms)

Figure 11 shows how the CoAP Transaction Time will change for the different security protocols. While the response size increased, the average CoAP Transaction time will increase. CoAP Transaction time with OSCORE is shorter than other security protocols. Although there is not much difference between TLS 1.2, TLS 1.3, and DTLS 1.2, TLS 1.3 is faster than the others in transferring packets.



	TLS 1.2 + CoAP	TLS 1.3 + CoAP	DTLS 1.2 + CoAP	OSCORE + CoAP	CoAP
🟦 17 Bytes Response Size	2.59	2.19	2.24	0.593	0.342
🟧 94 Bytes Response Size	4.35	4.175	4.32	0.661	0.633

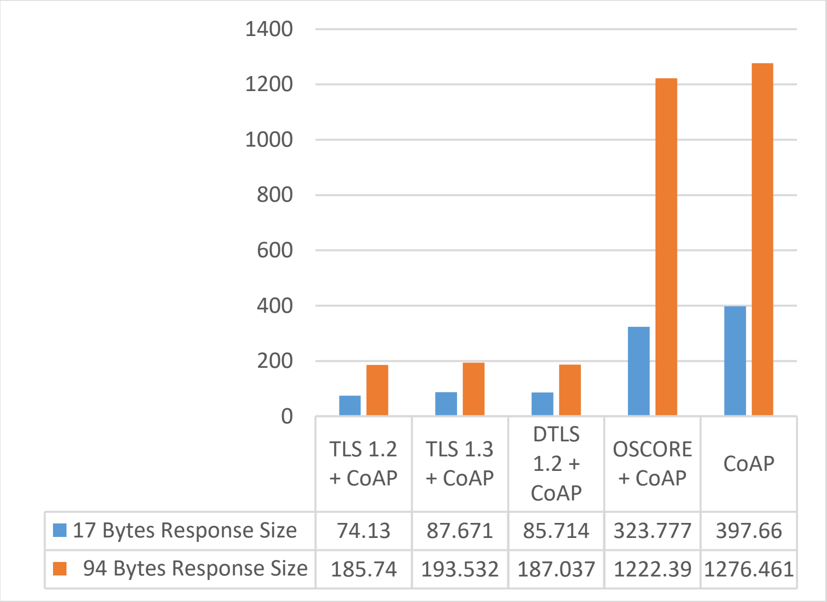
Figure 11: Simulation Average CoAP Transaction Time (ms)

From the specified table 2, we can note that the OSCORE security protocol contributes lower overhead to the CoAP GET request/response than all other protocols. With respect to the handshake security protocols, TLS 1.3 provides a significantly lower overhead than other protocols.

Security Protocol	Security Overhead (bytes) Request	Security Overhead (bytes) Response
TLS 1.2 + CoAP	29	29
TLS 1.3 + CoAP	22	22
DTLS 1.2 + CoAP	37	37
OSCORE + CoAP	14	10

Table 2: Security Overhead

Figure 12 shows the Average CoAP Transaction Throughput. By adding security over CoAP, the throughput is reduced, TLS 1.2 + CoAP has more security overheads than the others. OSCORE on the other hand, with lower security overhead, has also a lower impact on the CoAP transaction throughput.



	TLS 1.2 + CoAP	TLS 1.3 + CoAP	DTLS 1.2 + CoAP	OSCORE + CoAP	CoAP
🟦 17 Bytes Response Size	74.13	87.671	85.714	323.777	397.66
🟧 94 Bytes Response Size	185.74	193.532	187.037	1222.39	1276.461

Figure 12: Simulation Average CoAP Transaction Throughput (kbps)

CHAPTER 4

Implementation Results

This chapter presents performance results for implementation on IoT units of the previously presented security protocols TLS 1.2, TLS 1.3, DTLS 1.2, and OSCORE. For this purpose, we used a PC and NINA-W102 with ESP32 chip support as server/client; both are connected to the same Wi-Fi to communicate. We compared these protocols with their security overhead, handshake time and throughput, CoAP transaction time and throughput, and client current consumption. While the server was the same as used for the simulations. The client-side tests were performed using VisualGDB, which is a cross-platform for visual studio to build and debug the NINA-W102 unit, on windows 32/64 bit processor. The packet exchange between client and server is captured using Wireshark. All tests were simulated for 20 iterations, and reported results are the average values.

4.1 Implementation Set-Up and Results

The WolfSSL [33] and ESP-IDF [37] open sources were used to implement security protocols. WolfSSL was explained in chapter 3.

ESP-IDF is an IoT Development Framework, officially developed for ESP32, which provides necessary hardware, software libraries, source code, and scripts [39]. There are different versions of the ESP-IDF. Version 4.0 and Master are used for our implementation. Version 4.0 is one of the new updates available until October 2021 and supports the WolfSSL library. While for implementing DTLS 1.2, the latest version of ESP-IDF was used, the master version.

The NINA-W102 unit has opted for this work, which is mounted with an ESP32 chip. ESP32 is a series of low power and cost system-on-chip microcontrollers with Bluetooth and Wi-Fi dual-mode compatibility. It is optimized for smartphones, portable devices, and Internet-of-Things (IoT) devices. It features all state-of-the-art low-power chip characteristics, including fine-grained clock gating, various operation modes, and dynamic power scaling. The low-duty cycle is used to reduce the energy consumed by the chip. The power amplifier's performance is also flexible, leading to an optimum trade-off between contact frequency, data rate, and power usage [38]. These features motivate us to use the NINA-W102 unit for the implementation of the security protocols.

NINA-W102 [36] comes under the series of NINA-W10 series, and it is the product of U-blox. The NINA-W10 series are stand-alone multi-radio MCU modules that incorporate a powerful microcontroller (MCU) and a wireless radio. Customers will build specialized applications running on the 32-bit dual-core MCU utilizing the free CPU architectureNINA-W10 series modules that have a dual-core system with two Harvard Architecture Xtensa LX6 CPUs operating at a maximum of 240 MHz internal clock frequency. The internal NINA-W10 memory's main features include 448 Kbyte ROM for booting and core functions and 520 Kbyte SRAM for data and instruction. The radio supports 802.11b / g / n Wi-Fi in the 2.4 GHz ISM and Bluetooth v4.2 (Bluetooth BR / EDR and Bluetooth Low) band Communications Power (LE). NINA-W10 series modules are suitable for telematics, low power sensors, connected factories, connected buildings (appliances and surveillance), and point-of-sale, enabling advanced cryptographic hardware accelerators, health devices, and other design solutions which require the highest level of protection. The system's underlying architecture helps the developers to use an external antenna (NINA-W101) or the internal antenna (NINA-W102 and NINA-W106) in the application design. The NINA-W102 module contains a PIFA antenna. The RF signal is not attached to a pin on any board. The size of the panel is 10.0 x 14.0 mm, and the height is 3.8 mm. The maximum module supply voltage is 3.6 V, and the maximum current range is 500 mA. This module has 16/32 Mbit FLASH for code storage, including hardware encryption to protect programs. The product is shown in Figure 13:

Figure 13: NINA-W10

VisualGDB is an extension to Visual Studio that allows building embedded applications using GCC and debugging them using GDB. GDB or GNU Debugger is a GNU project that helps debug software applications and analyze what is happening during program execution. VisualGDB supports both local debugging (e.g., using an embedded simulator) and remote debugging. It also supports IoT modules; Barebone embedded systems, ESP32, and Arduino targets. VisualGDB will install and configure the required tools automatically [40].

Figure 14 shows the implementation set-up we has used for the experimental evaluations. The hardware used is the ESP32 NINA-W102 board connected to a windows operating computer using a USB cable for the client. Besides, we used software toolchains like GCC to compile the code, CMake, and ninja were used to build the tools with the help of Visual GDB application, and ESP-IDF to operate the toolchain. Also, Visual Studio Community 2019 was used as the text to write programs for projects in C. For the server, we have the same set-up, as mentioned in the simulation part chapter 3. To capture the packets, Wireshark was connected to the same Wi-Fi on the PC.

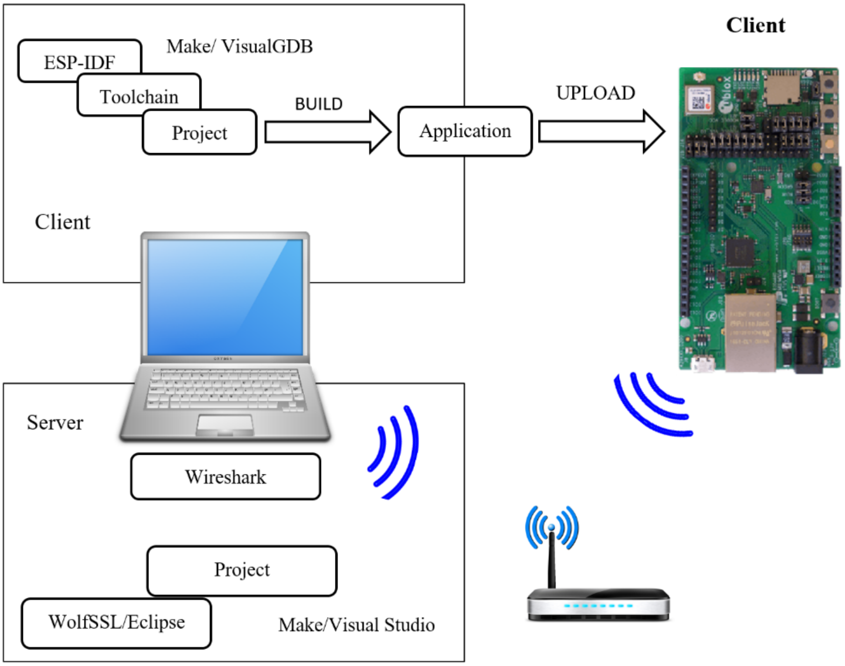


Figure 14: Implementation Set-Up

4.2 TLS 1.2 Implementation Results

ESP-IDF and WolfSSL open sources are used to implement the TLS 1.2 client, which described in Appendix B. NINA-W102 used as the client while the same server was used, as mentioned in section 3.2. To have the same status in all tests, certificate files, CoAP request/response, and cipher suite were the same for the simulation.

The first step of running the TLS 1.2 client is to add the WolfSSL library to the ESP-IDF and open a WolfSSL client project with the help of VisualGDB. The second step is to build and flash the TLS 1.2 client code to the NINA-W102. Finally, when we run the client, it was connected to the server with its IP address and port (5684) using Wi-Fi.

The performance parameters are the same as those used for the simulation part as we described in, Chapter 3, Section 3.2. In addition, the client current consumption during the implementation of TLS 1.2 is measured as a new performance parameter. To measure current consumption in each iteration during implementation time, the NINA-W102 ESP32 was connected to the DC Power Analyzer, and the current is monitored. The Wi-Fi connection is closed after the last flight in each iteration to reduce the noise and interferences during the current measurements. Figure 15, shown below, represents the client current measurement set-up:

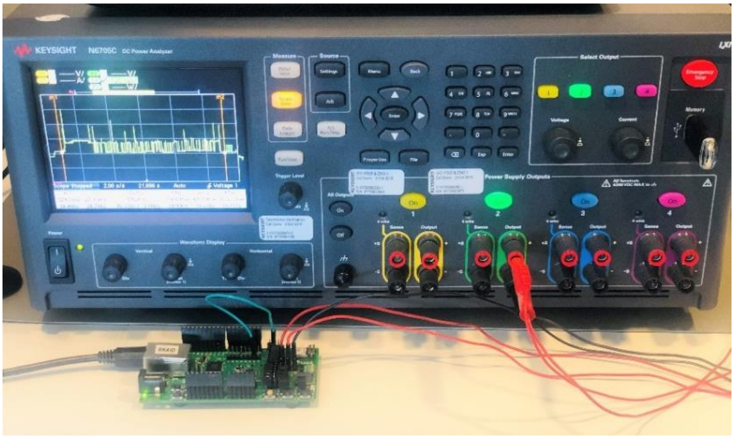


Figure 15: Client Current Measurements Set-Up

With average current consumption and time, we calculated the energy consumption of NINA-W102 for one full handshake and one request/response using the formula:

$$E = (V \times I) \times T\,Joule \qquad (2)$$

In Equation (2) V is the DC power analyzer's voltage, which is set to the specified 3.3 V (recommended voltage of the power supply for ESP32). Also, I is the average current consumption and T is the average time which are captured from DC power analyzer.

The measurement of the handshake time was carried out in the same way as described in section 3.1. For TLS 1.2, the average handshake time achieved for 20 iterations was 2.117 s.

To calculate the CoAP transaction time and throughput, we were using the same method as was used in the simulation. For the 7 bytes GET request with a response size of 17 bytes, the achieved average time and throughput for one CoAP transaction are 18.21 ms and 10.54 kbps. Similarly, for 94 bytes response size, the average time and throughput are 21.05 ms and 38.38 kbps. Also, the security overhead for TLS 1.2 was 29 bytes.

After considering 20 times of iterations, the average current consumption was measured as 80.91 mA in 8.061 s for 7 bytes GET request with a response size of 17 bytes. Similarly, for 94 bytes response, the average current consumption was measured as 81.08 mA in 8.494 s. Figure 16, shown below, represents the client current measurement for a single iteration for two different responses:

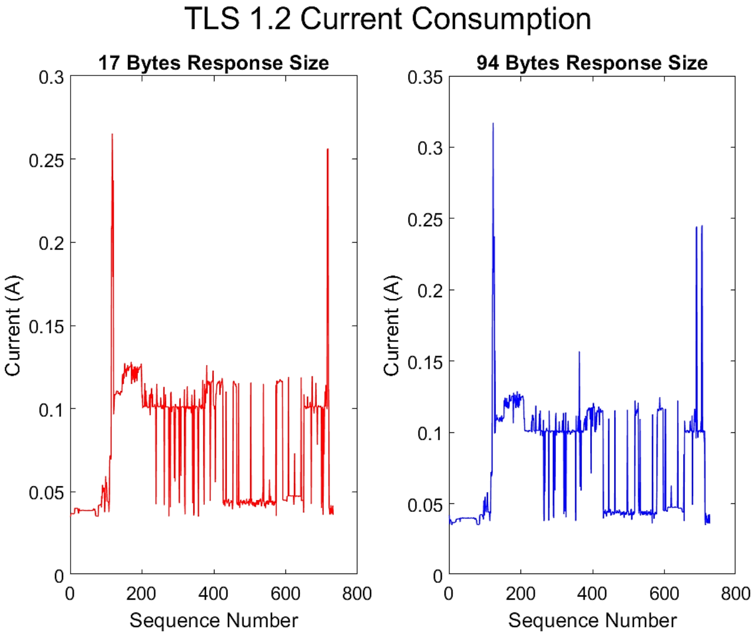


Figure 16: TLS 1.2 Current Consumption

In addition, the average energy consumption was calculated as 2.1523 J for 7 bytes GET request with a response size of 17 bytes. Similarly, for 94 bytes response, the average energy consumption was calculated as 2.2726 J.

4.3 TLS 1.3 Implementation Results

The implantation set-up we used was the same as we described in section 4.1. Also, all tests, certificate files, CoAP request/response, and cipher suites have the same status as in the simulation of TLS 1.3 in section 3.3.

For analyzing the performance of TLS 1.3, the same parameters were measured, as those mentioned in section 4.2.

For handshake time analysis, the average time achieved for 20 iterations was 1.163 s.

For CoAP transaction time and throughput analysis, the average time and throughput for one CoAP transaction was 18.19 ms, and 10.55 kbps, which is for 7 bytes GET request with a response size 17 bytes. Similarly, the average time and throughput are 20.3 ms and 39.802 kbps for the response size of 94 bytes. Moreover, the security overhead for TLS 1.3 was 22 bytes.

Current consumption was measured in the same way as we described in Section 4.1. The average current consumption was measured as 82.47 mA in 6.818 s for 7 bytes GET requests with a response size of 17 bytes. Similarly, for 94 bytes response, the average current consumption was measured as 84.29 mA in 7.169 s. Figure 17, shown below, represents the client current measurement for a single iteration for two different responses:

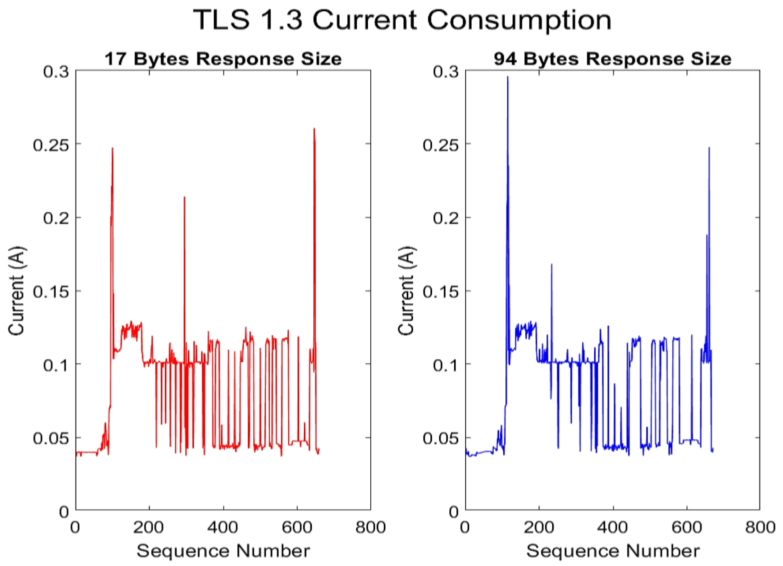


Figure 17: TLS 1.3 Current Consumption

The calculation for the average energy consumption was carried out in the same way as described in Section 4.2. The average energy consumption was calculated as 1.8555 J for 7 bytes GET request with a response size of 17 bytes. Similarly, for 94 bytes response, the average energy consumption was calculated as 1.994 J.

4.4 DTLS 1.2 Implementation Results

The implantation set-up we used was the same as we described in section 4.1. Also, all tests, certificate files, CoAP request/response, and cipher suites have the same status as in the simulation of TLS 1.2 in section 4.2.

For analyzing the performance of DTLS 1.2, the same parameters were measured, as those mentioned in section 4.2.

For handshake time analysis, the average time and throughput achieved for 20 iterations were 2.494 s.

For CoAP transaction time and throughput analysis, the average time and throughput for one CoAP transaction was 130.159 ms, and 1.475 kbps, which is for 7 bytes GET request with a response size 17 bytes. Similarly, the average time and throughput are 162.8 ms and 4.963 kbps for the response size of 94 bytes. Moreover, the security overhead for DTLS 1.2 was 37 bytes.

Current consumption was measured in the same way as we described in Section 4.2. The average current consumption was measured as 88.4 mA in 7.61 s for 7 bytes GET requests with a response size of 17 bytes. Similarly, for 94 bytes response, the average current consumption was measured as 88.6 mA in 8.392 s. Figure 18, shown below, represents the client current measurement for a single iteration for two different responses:

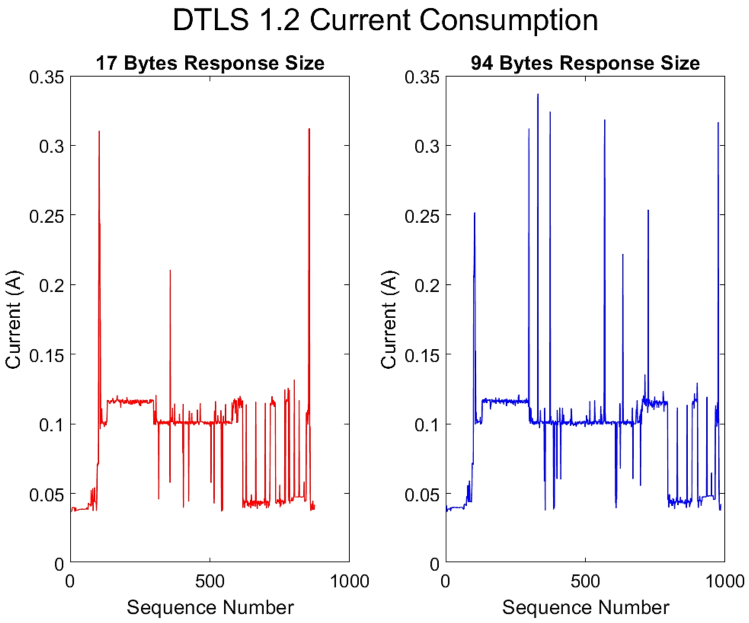


Figure 18: DTLS 1.2 Current Consumption

The average energy consumption was carried out, as mentioned in Section 4.2. The average energy consumption was 2.219 J for 7 bytes

GET request with a response size of 17 bytes. Similarly, for 94 bytes response, the average energy consumption was calculated as 2.453 J.

4.5 OSCORE Implementation Results

The UDP client example from ESP-IDF open source version 4.0 was used to implement the OSCORE client. Also, the same OSCORE + CoAP packet that was used in the simulation and described in Section 3.5, was used for the implementation evaluation. The open-source Eclipse / Californium was used as server.

The following results were achieved for the analysis of the performance of OSCORE using the same simulation method as we described in Section 3.5. For the 7 bytes CoAP GET request with a response size of 17 bytes for one packet exchange, the achieved average time and throughput for one packet exchange were 15.49 ms and 12.395 kbps. Similarly, for 94 bytes response size, the average time and throughput are 16.67 ms and 48.47 kbps. Moreover, the request overhead was 14 bytes, and the response overhead was 10 bytes.

Current consumption was measured in the same way as we described in Section 4.2. The average current consumption was measured as 97.72 mA in 3.366 s for 7 bytes GET requests with a response size of 17 bytes. Similarly, for 94 bytes response, the average current consumption was measured as 98.1 mA in 3.61 s. Figure 19, shown below, represents the client current measurement for a single iteration for two different responses:

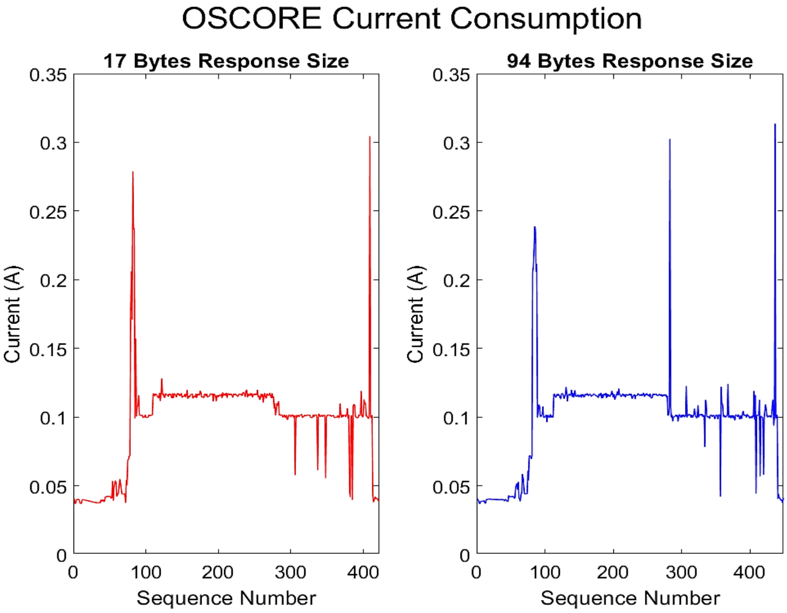


Figure 19: OSCORE Current Consumption

The average energy consumption was carried out as mentioned in Section 4.2. The average energy consumption was 1.085 J for 7 bytes GET request with a response size of 17 bytes. Similarly, for 94 bytes response, the average energy consumption was calculated as 1.169 J.

4.6 CoAP Implementation Results

The UDP client example from ESP-IDF open source version 4.0 was used to implement the CoAP client. Also, the same server with CoAP packet was used as described in Section 3.6.

The following results were achieved for the analysis of the performance of CoAP using the same simulation method as described in section 3.6. For the 7 bytes CoAP GET request with a response size of 17 bytes for one packet exchange, the achieved average time and throughput for one packet exchange were 15 ms and 12.8 kbps. Similarly, for 94 bytes response size, the average time and throughput were 15.6 ms and 51.79 kbps.

Current consumption was measured in the same way as described in Section 4.2. The average current consumption was measured as 97.9 mA in 3.355 s for 7 bytes GET requests with a response size of 17 bytes. Similarly, for 94 bytes response, the average current consumption was measured as 98.22 mA in 3.457 s. Figure 20, shown below, represents the client current measurement for a single iteration for two different responses:

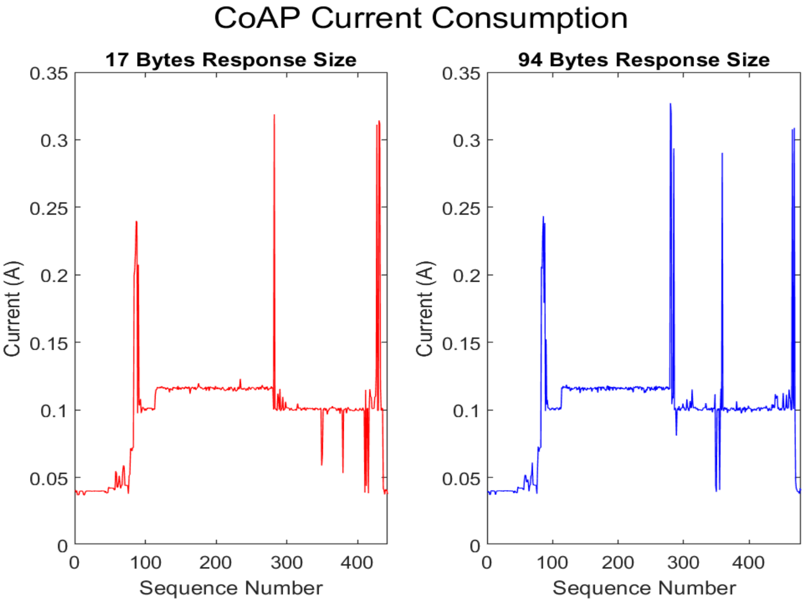


Figure 20: CoAP Current Consumption

The average energy consumption was carried out as mentioned in Section 4.2. The average energy consumption was 1.083 J for 7 bytes GET request with a response size of 17 bytes. Similarly, for 94 bytes response, the average energy consumption was calculated as 1.120 J.

4.7 Implementation Summary

Figure 21 shows the three security protocols' handshake analysis, including TLS 1.2, TLS 1.3, and DTLS 1.2. Due to the new handshake pattern mentioned in chapter 2, the average handshake time for TLS

1.3 outperforms the contrast with the other two protocols, which will affect performance, as multiple handshakes need to be sent for transferring data. Also, DTLS 1.2 has a lower handshake delay than TLS 1.2 as it is sending packets over the UDP socket.

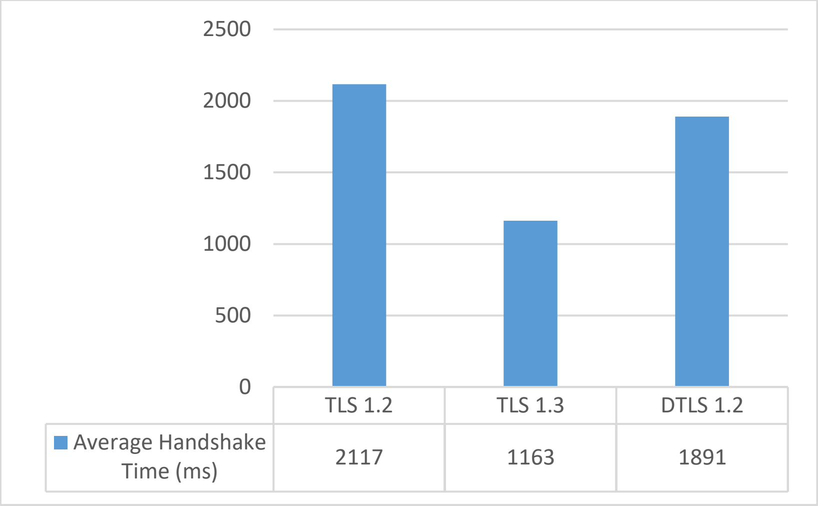
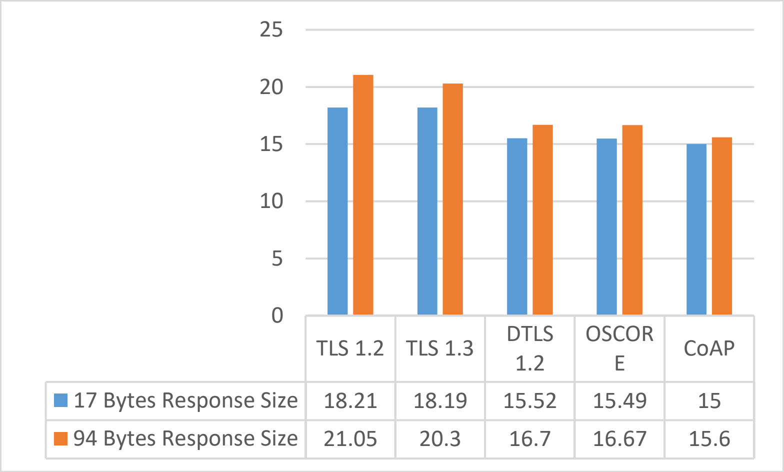


Figure 21: Average Handshake Time (ms)

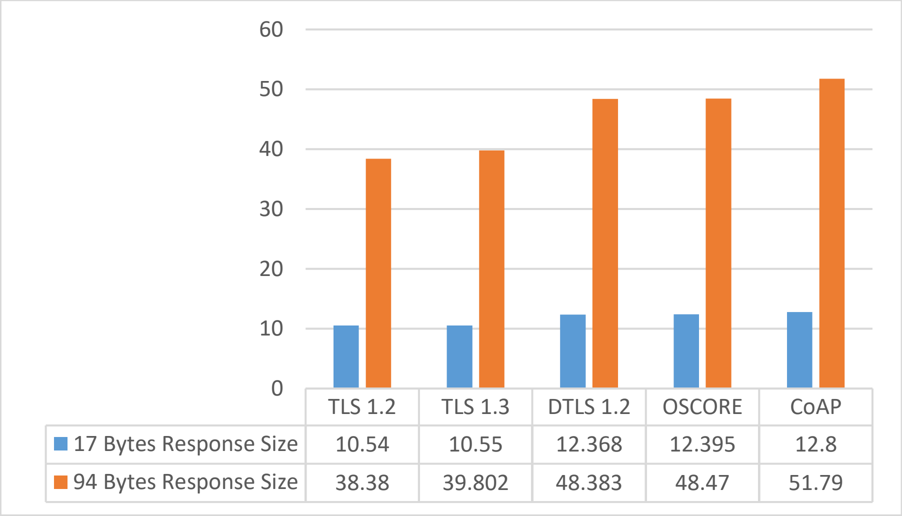
Figure 22 shows average CoAP transaction time for mentioned Security protocols with 17 and 94 bytes response size. By increasing the response size, we can see more delays in the transmission of data. OSCORE delay for data transfer is shorter than for other security protocols. As DTLS 1.2 sends data over UDP, it outperforms TLS 1.2 and TLS 1.3.



	TLS 1.2	TLS 1.3	DTLS 1.2	OSCORE	CoAP
🟦 17 Bytes Response Size	18.21	18.19	15.52	15.49	15
🟧 94 Bytes Response Size	21.05	20.3	16.7	16.67	15.6

Figure 22: Average CoAP Transaction Time (ms)

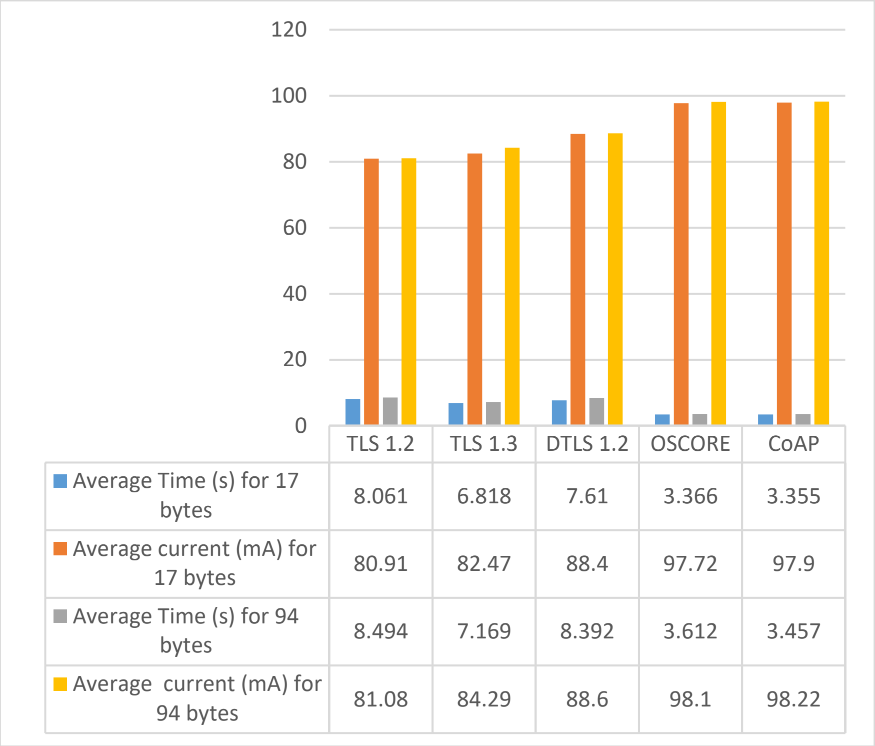
Figure 23 shows average CoAP transaction throughput for mentioned Security protocols with two different response size. By adding a security protocol over CoAP, the overhead will increase, which decreases the throughput. Also, DTLS 1.2, with more overhead, has a better throughput than TLS 1.2 & TLS 1.3 since it has a smaller latency when transmitting data.



	TLS 1.2	TLS 1.3	DTLS 1.2	OSCORE	CoAP
🟦 17 Bytes Response Size	10.54	10.55	12.368	12.395	12.8
🟧 94 Bytes Response Size	38.38	39.802	48.383	48.47	51.79

Figure 23: Average CoAP Transaction Throughput (kbps)

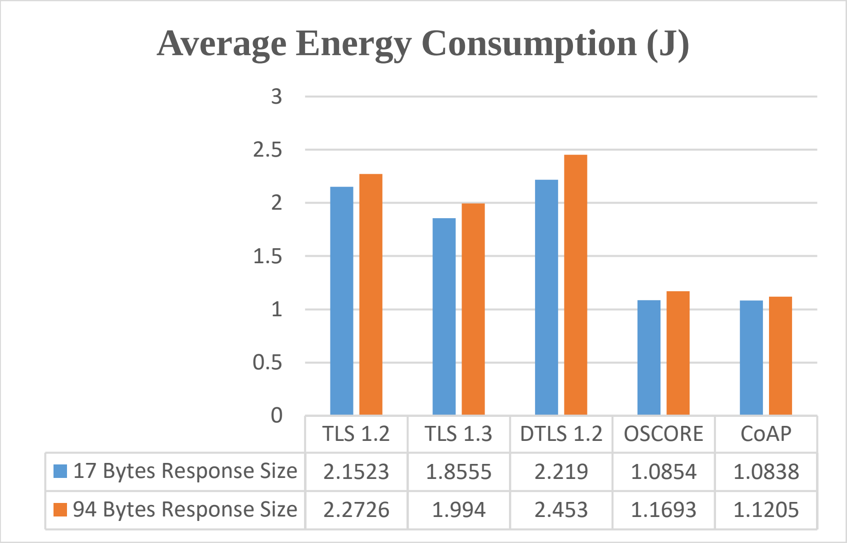
Figure 24 shows the average current consumption and time captured from the DC power analyzer for one full handshake + application data for TLS 1.2, TLS 1.3, and DTLS 1.2 with two different responses. Similarly, the mentioned parameters are shown for one exchange of OSCORE and CoAP packets. As seen in the figure, the average current consumption is high for security protocols with a lower average time as the throughput (bps) is higher.



	TLS 1.2	TLS 1.3	DTLS 1.2	OSCORE	CoAP
🟦 Average Time (s) for 17 bytes	8.061	6.818	7.61	3.366	3.355
🟧 Average current (mA) for 17 bytes	80.91	82.47	88.4	97.72	97.9
⬜ Average Time (s) for 94 bytes	8.494	7.169	8.392	3.612	3.457
🟨 Average current (mA) for 94 bytes	81.08	84.29	88.6	98.1	98.22

Figure 24: Average Current Consumption (mA) and Average Time (s)

The following Figure shows the average energy consumption of the NINA-W102 device with the same condition as mentioned in the previous paragraph. OSCORE will consume less energy for sending data while comparing with the other protocols. Seeing Figure 25, DTLS 1.2 consumes more energy than TLS 1.2. As mentioned in the Equation 2 the energy consumption is related to average current consumption and the average transmission time, so if we use one full handshake + application data, the average current would affect energy consumption more than the time as we can see in Figure 24. Thus, for scenarios where we have multiple handshakes or application data, DTLS 1.2 may outperform, as it is faster than TLS 1.2.



	TLS 1.2	TLS 1.3	DTLS 1.2	OSCORE	CoAP
🟦 17 Bytes Response Size	2.1523	1.8555	2.219	1.0854	1.0838
🟧 94 Bytes Response Size	2.2726	1.994	2.453	1.1693	1.1205

Figure 25: Average Energy Consumption (J)

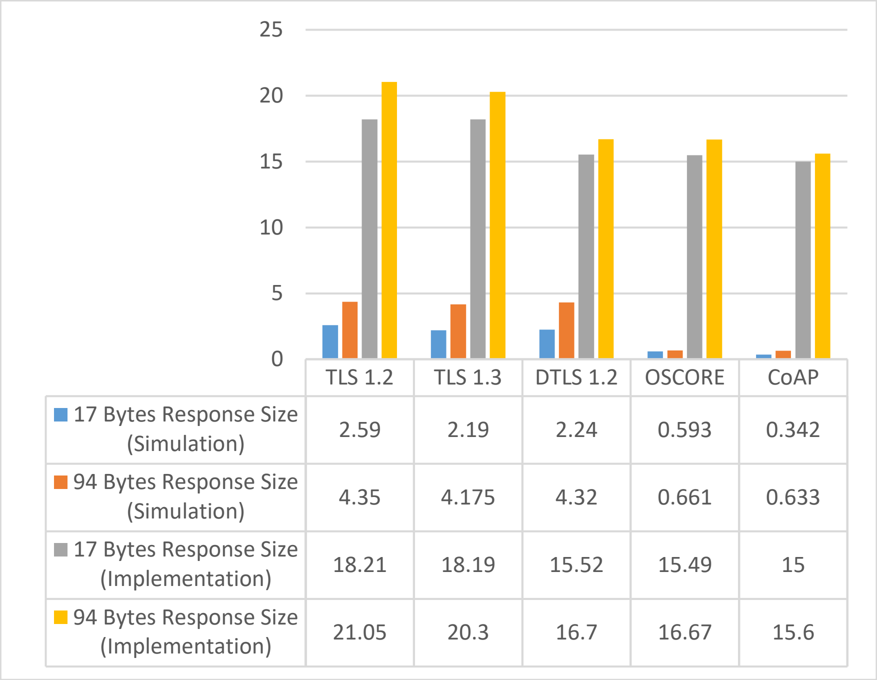
CHAPTER 5

Discussion and Conclusions

In this chapter, we will conclude the results of the implementation and simulation comparison of mentioned security protocols in chapter 3 and 4. In addition, we are going to discuss future work.

5.1 Comparison of Simulation and Implementation Results

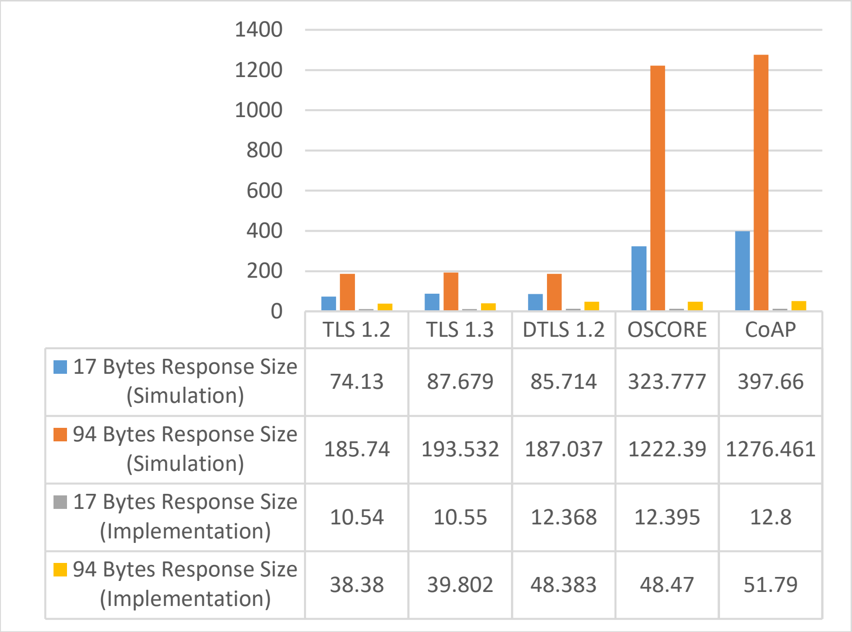
Figure 26 shows the average CoAP transaction time results for simulation and implementation of mentioned security protocols. As can be seen in the figure there is a large difference between the simulations and the implementation. The simulation response time is as expected, much smaller in the simulations than for the device implementation. However, there is in general a good conformity for the two realizations, with respect to response time differences between the protocols. When switching from simulation to implementation, time will reduce more for OSCORE and CoAP than other protocols. In comparison, DTLS 1.2 faces a smaller time reduction between simulation and implementation than other protocols. OSCORE and DTLS 1.2 can perform competitively in implementation, and with these two security protocols, there is not so much difference in time, although there is a considerable difference in simulation between them. Among the security protocols with a handshake, DTLS 1.2 outperforms data transmission during implementation, despite no significant difference between DTLS 1.2 and TLS 1.3 in simulation. As in the simulation, the client and server communicate through localhost, and even the security overhead of DTLS 1.2 is higher as compared to TLS 1.3, there is not much difference between DTLS 1.2 and TLS 1.3 in CoAP Transaction Time. On the other hand, while the client and server are connected to Wi-Fi, TLS over TCP would indicate more latency than DTLS over UDP.



	TLS 1.2	TLS 1.3	DTLS 1.2	OSCORE	CoAP
🟦 17 Bytes Response Size (Simulation)	2.59	2.19	2.24	0.593	0.342
🟧 94 Bytes Response Size (Simulation)	4.35	4.175	4.32	0.661	0.633
⬜ 17 Bytes Response Size (Implementation)	18.21	18.19	15.52	15.49	15
🟨 94 Bytes Response Size (Implementation)	21.05	20.3	16.7	16.67	15.6

Figure 26: Average CoAP Transaction Time (ms)

Figure 27 shows the average CoAP transaction throughput results for the simulation and implementation of the security protocols described above. When moving from simulation to implementation, the throughput will decrease as the average CoAP transaction time increases. OSCORE has a higher throughput in both simulation and implementation than other security protocols since it is faster. DTLS 1.2 outperforms the implementation of TLS 1.2 and TLS 1.3 as it transfers data over UDP. In the simulation, DTLS 1.2 and TLS 1.3 are not so different in throughput as there is not so much variation in CoAP transaction time in simulation between them.



	TLS 1.2	TLS 1.3	DTLS 1.2	OSCORE	CoAP
🟦 17 Bytes Response Size (Simulation)	74.13	87.679	85.714	323.777	397.66
🟧 94 Bytes Response Size (Simulation)	185.74	193.532	187.037	1222.39	1276.461
⬜ 17 Bytes Response Size (Implementation)	10.54	10.55	12.368	12.395	12.8
🟨 94 Bytes Response Size (Implementation)	38.38	39.802	48.383	48.47	51.79

Figure 27: Average CoAP Transaction Throughput (kbps)

5.2 Conclusion

In this thesis, we simulated and implemented different security protocols such as TLS 1.2, TLS 1.3, DTLS 1.2, and OSCORE to find the most effective one out of it. Different evaluation parameters were used to calculate the performance of these security protocols. Through evaluating the performance of the simulation and implementation, we can infer that OSCORE outperforms all other security protocols as it has lower latency, higher throughput, and lower energy consumption for NINA-W102 in data transmission because it does not have a key exchange protocol. However, we should consider that the algorithm used for OSCORE encryption is not the same as the other security protocols used in this thesis. Among security protocols that have a handshake, TLS 1.3 outperforms with a lower handshake latency and security overhead, while DTLS 1.2 outperforms in application data latency. These results are suggesting the need to switch from TLS 1.3 to DTLS 1.3. As DTLS 1.3 has TLS 1.3 handshake messages and flows, with some minor improvements, and the DTLS 1.3 application data is as fast as DTLS 1.2, we can conclude that DTLS 1.3 has lower latency than DTLS 1.2, TLS 1.2, and TLS 1.3. As a result, DTLS 1.3 would have less data transmission latency, greater CoAP message throughput, and reduced energy consumption than other handshake security protocols. Finally, in this thesis, for extrapolating DTLS 1.3, the handshake time and CoAP transaction time of DTLS 1.3 is chosen the same as TLS 1.3 handshake time and DTLS 1.2 CoAP transaction time.

Figure 28 shows the average time and throughput of the CoAP transaction of 94 bytes of response size for the security protocols specified. This response size was used as an example for extrapolating DTLS 1.3 to see the effectiveness of the implementation. Seeing the figure, DTLS 1.3 is supposed to be as fast as DTLS 1.2 when sending the data. As a result, the average CoAP transaction throughput for DTLS 1.3 became approximately as high as DTLS 1.2.

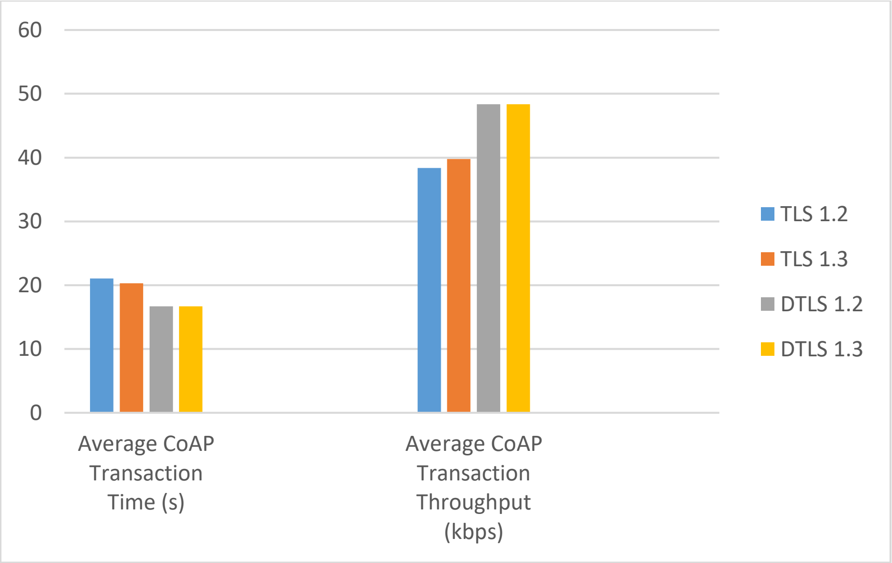


Figure 28: Average CoAP Transaction Time (s) and Throughput (kbps)

Figure 29 indicates the average delay and energy consumption of 94 bytes of response size for the security protocols listed below. Seeing the figure, the extrapolated DTLS 1.3 has a low handshake latency comparable to TLS 1.3. Also, due to the lower latency of DTLS 1.3, lower energy consumption is expected.

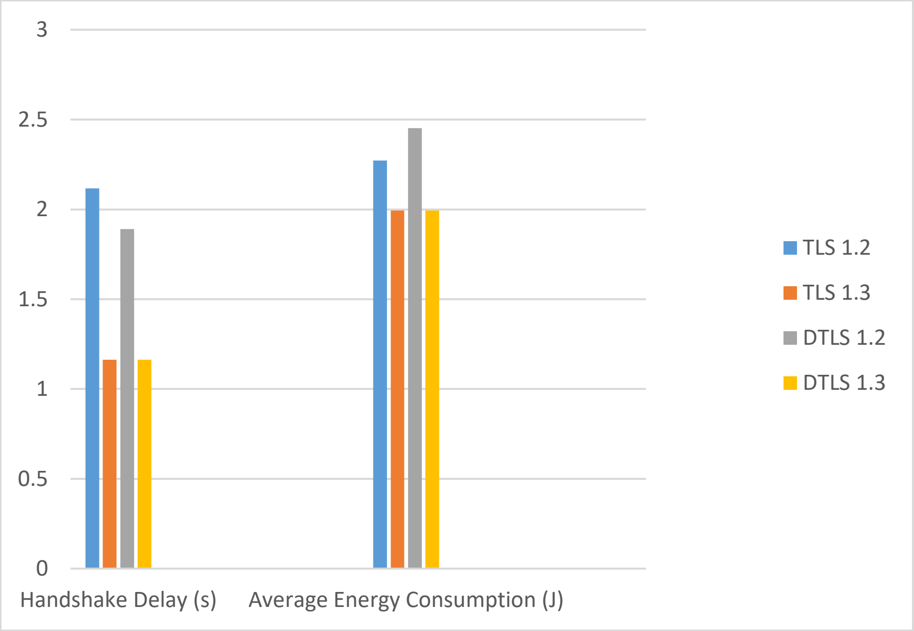


Figure 29: Handshake Delay (s) and Average Energy Consumption (J)

5.3 Future Work

The critical role of security in the real world would open up various fields of research in security protocols, one of which is to add key exchange protocols such as EDHOC over OSCORE to enhance the efficiency of data transmission. In addition, the implementation of stronger security protocol cipher suites would reduce power consumption and memory use on IoT devices and make communications more secure. One more work to be added is the development of DTSL 1.3, which is supposed to have reduced latency and higher throughput in data transmission.

References

[1] E. Rescorla, “Datagram Transport Layer Security”, RFC 4347 - Datagram Transport Layer Security, April 2006.

[2] T. Dierks, “The TLS Protocol Version 1.0”, RFC 2246 - The TLS Protocol Version 1.0, January 1999.

[3] G. Selander, J. Mattsson, and F. Palombini, “Object Security for Constrained RESTful Environments (OSCORE)”, RFC 8613 - Object Security for Constrained RESTful Environments (OSCORE), July 2019.

[4] G. Selander, J. Mattsson, and F. Palombini, “Ephemeral Diffie-Hellman Over COSE (EDHOC) draft-selander-ace-cose-ecdhe-10”,draft-selander-ace-cose-ecdhe-10 - Ephemeral Diffie-Hellman Over COSE (EDHOC), September 18, 2018.

[5] E. Rescorla, and N. Modadugu, “Datagram Transport Layer Security Version 1.2”, RFC 6347 - Datagram Transport Layer Security Version 1.2, January 2012.

[6] T. Dierks, and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2”, RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2, August 2008.

[7] E. Rescorla, H. Tschofenig, and N. Modadugu, “Datagram Transport Layer Security Version 1.3”, draft-ietf-tls-dtls13-34 - The Datagram Transport Layer Security (DTLS) Protocol Version 1.3, March 2020.

[8] E. Rescorla, and Mozilla, “The Transport Layer Security (TLS) Protocol Version 1.3”, RFC 8446 - The Transport Layer Security (TLS) Protocol Version 1.3, August 2018.

[9] Z. Shelby, K. Hartke, and C. Bormann, “The Constrained Application Protocol (CoAP)”, RFC 7252 - The Constrained Application Protocol (CoAP), June 2014.

[10] C. Bormann, M. Ersue, and A. Keranen, “Terminology for Constrained-Node Networks”, RFC 7228 - Terminology for Constrained-Node Networks, May 2014.

[11] A. Castellani, S. Loreto, A. Rahman, T. Fossati, and E. Dijk, “Guidelines for Mapping Implementations: HTTP to the Constrained Application Protocol (CoAP)”, RFC 8075 - Guidelines for Mapping Implementations: HTTP to the Constrained Application Protocol (CoAP), February 2017.

[12] K. Hartke, “Observing Resources in the Constrained Application Protocol (CoAP)”, RFC 7641 - Observing Resources in the Constrained Application Protocol (CoAP), September 2015.

[13] C. Bormann, and Z. Shelby, Ed, “Block-Wise Transfers in the Constrained Application Protocol (CoAP)”, RFC 7959 - Block-Wise Transfers in the Constrained Application Protocol (CoAP), August 2016.

[14] A. Bhattacharyya, S. Bandyopadhyay, A. Pal, and T. Bose, “Constrained Application Protocol (CoAP) Option for No Server Response”, RFC 7967 - Constrained Application Protocol (CoAP) Option for No Server Response, August 2016.

[15] G. Selander, J. Mattsson, F. Palombini, and L. Seitz, “Object Security for Constrained RESTful Environments (OSCORE)”, RFC 8613 - Object Security for Constrained RESTful Environments (OSCORE), July 2019.

[16] J. Schaad, “CBOR Object Signing and Encryption (COSE)”, RFC 8152 - CBOR Object Signing and Encryption (COSE), July 2017.

[17] C. Bormann, and P. Hoffman, “Concise Binary Object Representation (CBOR) draft-ietf-cbor-7049bis-13”, draft-ietf-cbor-7049bis-13 - Concise Binary Object Representation (CBOR), March 2020.

[18] L. Seitz, F. Palombini, M. Gunnarsson, and G. Selander, “OSCORE profile of the Authentication and Authorization for Constrained Environments Framework draft-ietf-ace-oscore-profile-02”, draft-ietf-ace-oscore-profile-02 - OSCORE profile of the Authentication and Authorization for Constrained Environments Framework, June 2018.

[19] J. Mattsson, F. Palombini, and M. Vucinic, “Comparison of CoAP Security Protocols draft-ietf-lwig-security-protocol-comparison-04”,draft-ietf-lwig-security-protocol-comparison-04-Comparison of CoAP Security Protocols, March 2020.

[20] G. Selander, J. Mattsson, and F. Palombini, “OSCORE: A look at the new IoT security protocol”, OSCORE: A look at the new IoT security protocol, November 2019.

[21] T. Dierks and C. Allen. “The TLS Protocol Version 1.0”, RFC 2246 - The TLS Protocol Version 1.0, January 1999.

[22] T. Dierks and E. Rescorla. “The Transport Layer Security (TLS) Protocol Version 1.1”, RFC 4346 - The Transport Layer Security (TLS) Protocol Version 1.1, April 2006.

[23] T. Dierks and E. Rescorla. “The Transport Layer Security (TLS) Protocol Version 1.2”, RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2, August 2008.

[24] M. Crispin, “Internet Message Access Protocol (IMAP) - Version 4rev1”, RFC 3501 - INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1. March 2003.

[25] M. Rose, “Post Office Protocol (POP) - Version 3”, RFC 1081 - Post Office Protocol: Version 3. November 1988.

[26] E. Rescorla and N. Modadugu, “Datagram Transport Layer Security”, RFC 4347 - Datagram Transport Layer Security. April 2006.

[27] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, “SIP: Session Initiation Protocol”, RFC 3261 - SIP: Session Initiation Protocol. June 2002.

[28] P. Karn, Qualcomm, and W. Simpson, “Photuris: Session-Key Management Protocol”, https://tools.ietf.org/pdf/rfc2522.pdf. March 1999.

[29]Bernstein, D., “ChaCha, a variant of Salsa20”, ChaCha, a variant of Salsa20, January 2008.

[30] Babbage, S., DeCanniere, C., Cantenaut, A., Cid, C., Gilbert, H., Johansson, T., Parker, M., Preneel, B., Rijmen, V., and M. Robshaw, “The eSTREAM Portfolio (rev. 1)”, The eSTREAM Project. September 2008.

[31] Y. Nir, and A. Langley, “ChaCha20 and Poly1305 for IETF Protocols”, RFC 7539 - ChaCha20 and Poly1305 for IETF Protocols, May 2015.

[31] Bernstein, D., "The Poly1305-AES message-authentication code", The Poly1305-AES message-authentication code, February 2005.

[32] Isobe, T., Ohigashi, T., Watanabe, Y., and M. Morii, “Full Plaintext Recovery Attack on Broadcast RC4”, Full Plaintext Recovery Attack on Broadcast RC4.

[33] https://github.com/wolfSSL.

[34] https://github.com/contiki-ng/contiki-ng.

[35] https://github.com/eclipse/californium.

[36] https://www.u-blox.com/sites/default/files/NINA-W10_DataSheet_%28UBX-17065507%29.pdf.

[37] https://www.u-blox.com/en/docs/UBX-17051775.

[38] https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[39] https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html.

[40] https://visualgdb.com/download/.

[41] https://en.wikipedia.org/wiki/Cipher_suite#DTLS_with_cipher_suites.

[42] https://www.wolfssl.com/differences-between-tls-1-2-and-tls-1-3/.

[43] https://tools.ietf.org/html/rfc8446#page-8.

APPENDIX A

A.1 TLS and DTLS simulation setup

TLS 1.2:

TLS 1.2 server example starts working with the following commands in the Windows Command Prompt after building the library:

```
>cd wolfssl-master/Debug

>server.exe –v 3
```

TLS 1.2 client example starts working with the following commands in the Windows Command Prompt after building the library:

```
>cd wolfssl-master/Debug

>client.exe –v 3
```

TLS 1.3:

To simulate TLS 1.3, some functions should be enabled in user_settings.h file:

```
#define WOLFSSL_TLS13

#define HAVE_TLS_EXTENSIONS

#define HAVE_SUPPORTED_CURVES

#define HAVE_ECC

#define HAVE_HKDF
```

```
#define HAVE_FFDHE_8192
```

```
#define WC_RSA_PSS
```

TLS 1.3 server example starts working with the following commands in the Windows Command Prompt after building the library:

```
>cd wolfssl-master/Debug

>server.exe –v 4
```

TLS 1.3 client example starts working with the following commands in the Windows Command Prompt after building the library:

```
>cd wolfssl-master/Debug

> client.exe –v 4
```

DTLS 1.2

Using the user_settings.h header file DTLS 1.2 can be enabled by:

```
#define WOLFSSL_DTLS
```

DTLS 1.2 server example starts working with the following commands in the Windows Command Prompt after building the library:

```
>cd wolfssl-master/Debug

>server.exe  –u  –v  3
```

DTLS 1.2 client example starts working with the following commands in the Windows Command Prompt after building the library:

```
>cd wolfssl-master/Debug

> client.exe  –u  –v  3.
```

APPENDIX B

B.1 TLS and DTLS Setup On NINA-W10

Setup Instructions:

- Windows PC running TLS server from WolfSSL.
- NINA-W10 running TLS client from WolfSSL.
- Wi-Fi Access point to which both the server and client are connected.

Instruction for TLS and DTLS client:

1. Download VisualGDB version 5.5.
2. Start Visual Studio, create a new project and open the VisualGDB ESP32 project wizard.
3. On the first page of the wizard select, the CMake build subsystem.
4. Install and select the latest ESP32 under toolchain and the ESP-IDF version 4.0 under SDK Checkout.
5. Set the ESP-IDF path on environment variables to the version 4.0 path.
6. Download WolfSSL library from GitHub.
7. Run setup.sh from wolfssl/IDE/espressif/ESP-IDF to deploy files into the ESP-IDF tree.
8. Uncomment out "#define WOLFSSL_ESPIDF" in path to wolfssl/wolfssl/wolfcrypt/settings.h.
9. Go back to VisualGDB and select Wolfssl Client from project samples.
10. On the Debug settings page, select the JTAG debugger (e.g. Olimex ARM-USB-OCD-H).
11. Press, "Finish" to generate the project. Once the project is loaded, open the "client-tls.c" and replace WolfSSL method to

DTLS 1.2/TLS 1.3/TLS 1.2. Also, to run DTLS there is a need to change the socket to UDP.

12. Make menuconfig to configure the project.
    12.1. Example Configuration:

    Set up Wi-Fi SSID.

    Set up Wi-Fi Password.
13. Target host IP address: Set the server IP address in “#define WEB_SERVER” in the main code.
14. Flash and run the project.