TÉCNICO LISBOA

L-TLS: Lightweight TLS for the New IoT World

Tomás Borges de Castro de Vasconcelos Carrasco

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor(s): Prof. Ricardo Jorge Fernandes Chaves
Prof. Aleksandar Ilic

Examination Committee

Chairperson: Prof. Rui Filipe Fernandes Prada
Supervisor: Prof. Ricardo Jorge Fernandes Chaves
Member of the Committee: Prof. Alberto Manuel Ramos da Cunha

07 2021

Acknowledgments

I would like to thank the supervisors of my work, professor Ricardo Chaves and Aleksander Ilic for all the guidance and help they have given me through this dissertation. I would also like to express my gratitude to my family, especially my mother and father, and all my close friends for all the support, motivation and encouragement they have given me through this work.

Resumo

O *Transport Layer Security* (TLS) é um dos protocolos de comunicação segura mais utilizados no mundo. Através dele, é possível criar canais de comunicação que fornecem serviços de segurança importantes, assim como confidencialidade, integridade, autenticação, estabelecimento de chave e *perfect forward secrecy*. Os serviços são implementados através do uso de algoritmos definidos em ciphersuites TLS. Executar o protocolo TLS requer muitos recursos devido à natureza das suas operações, tornando-o impróprio para dispositivos de *Internet of Things* (IoT). Contudo, é possível seleccionar que algoritmos irão ser utilizados e, consequentemente, que serviços irão ser fornecidos. Caso seja configurado correctamente, o TLS pode ser utilizado em ambientes IoT constrangidos de forma adequada. Este trabalho pretende criar uma ferramenta que permitirá aos utilizadores seleccionar configurações TLS apropriadas, possibilitando a execução do TLS nos seus dispositivos e a optimização do mesmo. Esta ferramenta utiliza a biblioteca Mbed TLS como objecto de estudo, contendo uma extensa análise desta biblioteca. Esta ferramenta permitirá que os utilizadores adicionem métricas de custo e utilizem várias ferramentas para realizar as medições. Esta ferramenta considera métricas básicas, assim como o tempo de execução e o número de ciclos CPU. Esta ferramenta permitirá o uso de implementações de algoritmos alternativas para optimizar ainda mais o protocolo TLS. No final, as capacidades da ferramenta são testadas ao utiliza-lo para analisar a performance do protocolo *Handshake* e o impacto de utilizar uma implementação do AES alternativa que utiliza o conjunto de instruções AES-NI.

Palavras-chave: TLS, SSL, Mbed TLS, IoT, Performance, Serviços de Segurança

Abstract

Transport Layer Security (TLS) is one of the most used communication security protocols in the world. Through its use, it is possible to create a communication channel that provides important security services, such as confidentiality, integrity, authentication, key establishment and Perfect Forward Secrecy (PFS). The services are implemented through the use of algorithms defined in TLS ciphersuites. Executing the TLS protocol requires lots of resources due to the nature of its operations, making it unsuited for Internet of Things (IoT) devices. However, it is possible to select the algorithms that will be used and thus the services will be provided. If configured properly, TLS can even be utilized in constrained IoT environments. This work aims to create a tool that will allow its users to select proper TLS configurations, enabling their devices to execute TLS and optimize it. This tool will make use of the Mbed TLS library as its study target, thus an extensive analysis of this library is present in this work. This tool will allow its users to add cost metrics and use various tools to perform the measurements. This tool will implement basic metrics, such as execution time and the number of CPU cycles. This tool will allow the use of alternate algorithms implementations to further optimize the TLS protocol. In the end, the capabilities of the tool will be tested, by analysing the performance of the Handshake protocol and the impact of using an alternate AES implementation, that makes use of the AES-NI instruction set.

Keywords: TLS, SSL, Mbed TLS, IoT, Performance, Security Services

Contents

Acknowledgments . . . iii
Resumo . . . v
Abstract . . . vii
List of Tables . . . xi
List of Figures . . . xiii
Nomenclature . . . xv

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Objectives . . . 2
1.3 Main Contributions . . . 3

2 Background **5**
2.1 Cryptography and Security Services . . . 5
2.2 Security Protocols . . . 7
2.3 TLS . . . 9
2.3.1 General Overview . . . 9
2.3.2 History . . . 10
2.3.3 Records . . . 10
2.3.4 Handshake Protocol . . . 11
2.3.5 Record Protocol . . . 13
2.3.6 DTLS and Other Variations . . . 14
2.3.7 Vulnerabilities and Common Attack . . . 15
2.3.8 Protocol Version 1.3 . . . 16
2.4 Computing Platforms . . . 16

3 State of Art **21**
3.1 Implementations and Libraries . . . 21

3.2 Performance Evaluation Tools 24

4 Proposed Solution **27**

4.1 System Architecture 27

4.2 Data Acquisition Module 29

4.2.1 Measurement Component 30

4.2.2 TLS Component 32

4.2.3 Communication Component 37

4.3 Data Analysis Module 40

4.3.1 Security Services Analysis 40

4.3.2 Algorithms Analysis 44

4.4 User Interfaces 45

5 Results and Discussion **49**

5.1 Scenario 1: Analysis of the Security Services Provided by the Handshake Protocol 49

5.2 Scenario 2: Comparative Analysis of Different AES and SHA-2 Implementations 53

5.3 The Tool 58

6 Conclusions and Future Work **59**

6.1 Achievements 59

6.2 Future Work 60

Bibliography **61**

A Handshake Protocol Performance **65**

B Software Implementations Comparison **69**

C Shell Tool User Guide **75**

C.1 Data Acquisition Tools 75

C.2 Data Analysis Tools 77

List of Tables

4.1 List of security services and the respective algorithms that provide them. 28
4.2 List of key exchange algorithms and the material each respective algorithm needs to form the premaster secret. 34
4.3 List of the parameters used by the communication endpoints, their default value and description. 38
4.4 Security levels with their corresponding security strength and key sizes for all algorithms used during the handshake. 39
4.5 List of algorithm types and the respective list of algorithms that belong to it. . . 44

List of Figures

2.1 TLS protocol, with the TCP handshake in blue, the handshake protocol in orange and the record protocol in gray . . . 9
2.2 General TLS record structure . . . 11

4.1 Architecture of the tool . . . 28
4.2 Architecture of the Measurement Component . . . 30
4.3 Bar plot showing the server-side performance of the algorithms that provide the Authentication service, in cycles . . . 42
4.4 Main window of the Graphical User Interface . . . 46
4.5 Edit and Profile window of the GUI . . . 46

5.1 Performance of each key exchange algorithm for security strength of 112 bits, in number of CPU cycles. . . . 51
5.2 Performance of each algorithm that provides the authentication security service for all security strengths, in number of CPU cycles. . . . 52
5.3 Mean and standard deviation of the AES operations for all message sizes, in microseconds. . . . 55
5.4 Mean and standard deviation of the SHA-2 operations for all message sizes, in microseconds. . . . 56
5.5 Comparison of the performance of both AES implementations for all message sizes, in microseconds. . . . 57

A.1 Performance of each key exchange algorithm for security strength of 128 bits, in number of CPU cycles. . . . 65
A.2 Performance of each key exchange algorithm for security strength of 192 bits, in number of CPU cycles. . . . 66
A.3 Performance of each algorithm that provides the key establishment security service for all security strengths, in number of CPU cycles. . . . 67

A.4 Performance of each algorithm that provides the perfect forward secrecy security service for all security strengths, in number of CPU cycles. 68

B.1 Data distribution of the AES operations for all message sizes, in microseconds. . 70

B.2 Mean, median and mode of the AES operations for all message sizes, in microseconds. 71

B.3 Data distribution of the SHA-2 operations for all message sizes, in microseconds. 72

B.4 Mean, median and mode of the SHA-2 operations for all message sizes, in microseconds. 73

B.5 Comparison of the performance of both SHA-2 implementations for all message sizes, in microseconds. 74

Nomenclature

Algorithm Names

3DES-EDE Triple DES Encryption-Decryption-Encryption

AES Advanced Encryption Standard

DES Data Encryption Standard

RC4 Rivest Cipher 4

DH Diffie–Hellman

DHE DH Ephemeral

ECDH Elliptic Curve DH

ECDHE Elliptic Curve DHE

ECDSA Elliptic Curve Digital Signature Algorithm

PSK Pre-Shared Key

RSA Rivest–Shamir–Adleman

MD5 Message Digest 5

SHA Secure Hash Algorithm

Others

ECC Elliptic Curve Cryptography

IoT Internet of Things

PFS Perfect Forward Secrecy

Protocols

DTLS Datagram Transport Layer Security

SSL Secure Sockets Layer

TLS Transport Layer Security

Algorithm Categories

AEAD Authenticated Encryption with Associated Data

MAC Message Authentication Code

Chapter 1

Introduction

This chapter contains a small introduction to the topics that will be covered in this project and establishes the main issue that is trying to be solved. It also includes the requirements and objectives of the tool that was developed to solve the main issue, as well as the major contributions provided by this project.

1.1 Motivation

The Internet of Things (IoT) is a system composed of computing devices, as well as mechanical and digital machines, that are connected with each other. Each device or machine has a unique identifier and the ability to transfer data over a network. These devices or machines are usually referred to as IoT devices.

Nowadays, the use of IoT devices is becoming more common, since they can be used to perform a wide variety of tasks. There are many types of IoT devices, such as wireless sensors, software, actuators, and computers. They can be embedded in various environments, such as mobile devices, industrial equipment, environmental sensors, or medical devices, to perform those tasks. IoT devices also play an important role in the concept and execution of smart home technologies, that simply improve the life quality of its user.

Since IoT devices are connected to a network, they are susceptible to various attacks. These attacks can lead to information theft or loss or corruption of critical data, precluding the correct working of a device. Unsecured IoT devices can also be used as a backdoor into a secure network. As such, secure communication protocols were created with the intent of securing commutation channels and preventing such things from happening.

TLS, or Transport Layer Security, is one of those protocols. Currently, it is used in many applications, such as web browsing, email, instant messaging, and voice over IP, to secure com-

munication. Although effective if used properly, TLS requires the execution of many heavy and costly operations to serve its purpose.

IoT devices are usually built to be portable and simple to use, as such they can be composed of specific hardware components, that provide the desired functionality but limit the resources available to the device. Manufacturers also need to consider costs for the mass production of those devices and can opt for the use of cheaper and, in turn, weaker components, further constraining the resources available to the devices.

To be able to use TLS, IoT devices usually must compromise and choose between having weaker security to provide the desired level of performance or having a worse performance to provide the desired level of security.

Nevertheless, it is still possible to achieve a desirable trade-off between security and performance. This is done by efficiently using the resources available to a device and by carefully selecting the configurations that the TLS session will use.

With this idea in mind, this work will create a tool that can provide the desired trade-off between security and performance and enable the use of safer and/or more efficient TLS sessions, by selecting the best available TLS configurations. The tool will also support alternate algorithm implementations that take advantage of hardware acceleration techniques, such as the use of the AES-NI instruction set in Intel and AMD processors.

1.2 Objectives

At the end of this work, it must be implemented a tool that creates a list of possible TLS configurations for a device and then selects the best set of configurations from that list.

The implementation of this tool must also respect the following requirements:

- The tool should take advantage of all the available cryptographic mechanisms provided by the device that is being evaluated
- The tool must provide secure configurations and take into account the specifications and limitations of the target device
- The tool must select the configuration that best provides the desired trade-off between security and performance
- The tool must provide mechanisms that enable the evaluation of each implementation

1.3 Main Contributions

In this work, a tool was created that allows its users to get a detailed analysis of all phases of the TLS protocol. This tool will create a list of possible TLS configurations that can be used by a given device.

The analysis given by the tool consists of a breakdown of the performance of the algorithms used during TLS sessions, as well as the security services provided in the session.

The tool provides a dynamic performance analysis as it provides an interface that allows new metrics to be easily integrated and dynamically choosing which ones to use.

This work contains an extensive analysis of the TLS implementation provided by the Mbed TLS 2.16.5 library, which was used as a research target since it is one of the most popular TLS implementation libraries used in embedded systems.

This work demonstrates the capabilities of the developed tool by using it to analyse the performance of the TLS protocol in two different scenarios.

This work also demonstrates the benefits of integrating hardware acceleration techniques, such as the use of the AES-NI instruction set, to optimize the performance of the TLS session.

Chapter 2

Background

This section introduces cryptography and how its applications can be divided into security services. There is also an overall explanation of those protocols and a more in-depth analysis of the TLS protocol. This analysis focuses mainly on version 1.2 of the protocol and some more details on many aspects of the protocol. Finally, there is an overview of computing platforms and the inner workings of the AES-NI instruction set.

2.1 Cryptography and Security Services

Cryptography is the practice and study of techniques for securing communication from third parties, which involves the creation and analysis of protocols that accomplish this objective.

Cryptography can be used for many purposes, such as verifying authorization access or permissions to modify information [1]. To better group these services, cryptography can be divided into security services, each with its own purpose. These services are confidentiality, integrity, authentication, and non-repudiation.

Confidentiality is used to hide the content of the information from all, except the entities allowed to access it. Typically, confidentiality is assured through ciphering information using cryptographic algorithms and keys. This process will be better explained later in this section.

Integrity is used to preserve the consistency of the information and to prevent unauthorized modifications to it. Cryptographic hash functions are used to achieve this service. Hash functions are one-way functions that generate a fixed dimension value called hash, based on the inputted texts. They can be used by comparing the hash contained in a message and the hash computed from that message on the receiver end, making it possible to detect if the message was altered. Hash functions must follow the following properties:

- Pre-image resistance: Given a hash value h it should be difficult to find any message m

such that h = hash(m);

- Second pre-image resistance: Given an input m1, it should be difficult to find a different input m2 such that hash(m1) = hash(m2);

- Collision resistance: It should be difficult to find two different messages m1 and m2 such that hash(m1) = hash(m2).

Authentication assures the identity of the sender or the origin of a received message. To assure authentication, MACs (Message Authentication Codes) are used. MACs can be created by hashing a message to get a digest and then ciphering that digest using a symmetric cipher algorithm.

Finally, Non-repudiation is a service that prevents an entity from denying its previous commitments or actions. Non-repudiation is achieved using digital signatures. Digital signatures are very similar to MACs but instead of using symmetric ciphering, they use asymmetric ciphering. This means that the sender has a secret key that is only possessed by him. By using that key to sign messages, he cannot refute his ownership over those messages.

As mentioned previously, confidentiality is achieved using cryptographic algorithms and keys. Starting with the algorithms, they are used to cipher plaintext into ciphertext and decipher ciphertext into plaintext and can either be symmetric or asymmetric.

A symmetrical cipher algorithm only uses one key, preferably private, for ciphering and deciphering information, which means that 2 entities use the same key when communicating. An asymmetrical cipher algorithm uses a pair of keys. One of those keys is private, while the other is public. If one key was used to cipher a message, the other one must be used to decipher it. For 2 entities to communicate, each must have its own key pair and the public key of the other party. The sender ciphers the message using the public key of the receiver and the receiver deciphers the message using its own private key.

Symmetrical ciphers can also be classified as stream ciphers or block ciphers. Stream ciphers generate a keystream, from the key, that is then combined with the plaintext, byte by byte, to create the ciphertext. In block ciphers, the plaintext is grouped into blocks with a fixed size (64, 128, etc bits), and the blocks are then ciphered one at a time. Smaller amounts of data can be padded onto the block, to assure that it has the pretended size.

To cipher data that does not fit in a single block, cipher modes were created [2]. Each cipher mode describes how to repeatedly apply a block cipher using a different method. Some of the modes are ECB (Electronic CodeBook), CBC (Cipher Block Chaining), OFB (Output Feedback mode), and CTR (CounTeR mode).

In ECB mode, the data is divided into blocks and each block is encrypted separately. In CBC mode, each block of plaintext is XORed with the previous ciphertext block before being ciphered. This way, each ciphertext block depends on all the previously processed plaintext blocks. To assure the uniqueness of the ciphertext, an IV (Initialization Vector) must be used when ciphering the first block. The OFB mode turns a block cipher into a stream cipher. It generates keystream blocks, from an IV, which are then XORed with the plaintext blocks to get the ciphertext. Finally, CTR mode, like OFB, also turns a block cipher into a stream cipher. It generates the next keystream block by ciphering successive values given by a function. This function can simply be a regular counter that increments its value by one.

There is also a hybrid ciphering method, that uses both symmetric ciphering and asymmetric ciphering. It works by sharing a key using an asymmetric cipher, and, from there on, symmetric ciphering is applied using that key. The shared key can also be called a session key.

Cryptographic algorithms take keys as input to perform their operations. These keys are a piece of information that determines the functional output of the algorithm. Keys should be generated using a truly random key generator that is cryptographically strong. The key size depends on the algorithm it will be used on, as such, keys can have various sizes. 128-bit keys are commonly considered strong.

As mentioned previously, when using asymmetric ciphers, one of the keys is public, but that does not mean its distribution is not a challenge. The sender needs to be sure that it is using the public key of the receiver when communicating, else the message can never get to the receiver or the sender can even be tricked into communicating with a third party. To address this challenge, public key certificates and certification authorities were created.

Public key certificates are public documents used to prove ownership over a public key. These certificates contain information regarding the key, the identity of its owner, and the digital signature of a certification entity. The certificate receiver can validate the certificate signature using the public key of the certification authority. Certification authorities are official organizations that manage certificates and CRLs (Certificates Revocation Lists). They also define policies and mechanisms for the generation and distribution of certificates. If an entity trusts in the certificate authority and the signature is valid, then he can trust the certified public key.

2.2 Security Protocols

A security protocol is a sequence of operations that ensure the protection of data. When used with a communication protocol, it provides secure delivery of data between 2 entities. These

protocols must incorporate some of the security services described in the previous section along with other aspects, such as:

- Key agreement or establishment
- Entity authentication
- Symmetric ciphers and message authentication
- Secured application-level data transport
- Non-repudiation methods
- Secret sharing methods
- Secure multi-party computation

There are various security protocols, such as IPsec, Kerberos, SSH, and TLS, each with its purposes.

IPsec, or Internet Protocol Security, is a secure network protocol that protects all IP traffic between machines. It provides confidentiality and integrity to the packets of data and is currently used in VPNs.

Kerberos is an authentication protocol that works by giving tickets to network nodes, allowing them to communicate over a non-secure network by proving their identity to one another in a secure manner. It is primarily aimed at client-server models and provides mutual authentication, as well as eavesdropping and replay attack protection.

SSH, or Secure Shell, is a secure communication application and protocol that works over TCP/IP. It establishes secure remote sessions and multiplexes other information flows over a secure session. It provides confidentiality and integrity of the communication, key distribution, and authentication of the communicating parties. The secure channel is provided using client-server architecture, connecting an SSH client to an SSH server.

TLS, or Transport Layer Security, is a communication protocol that operates over TCP/IP and manages secure sessions. Like SSH, TLS also provides confidentiality and integrity of the communication, key distribution, and authentication of the communicating parties. It also uses a client-server architecture.

2.3 TLS

2.3.1 General Overview

As mentioned previously, TLS is a communication protocol that provides secure communication over computer networks. The protocol is comprised of the handshake protocol and the record protocol. Figure 2.1 shows the establishment of a TLS session.

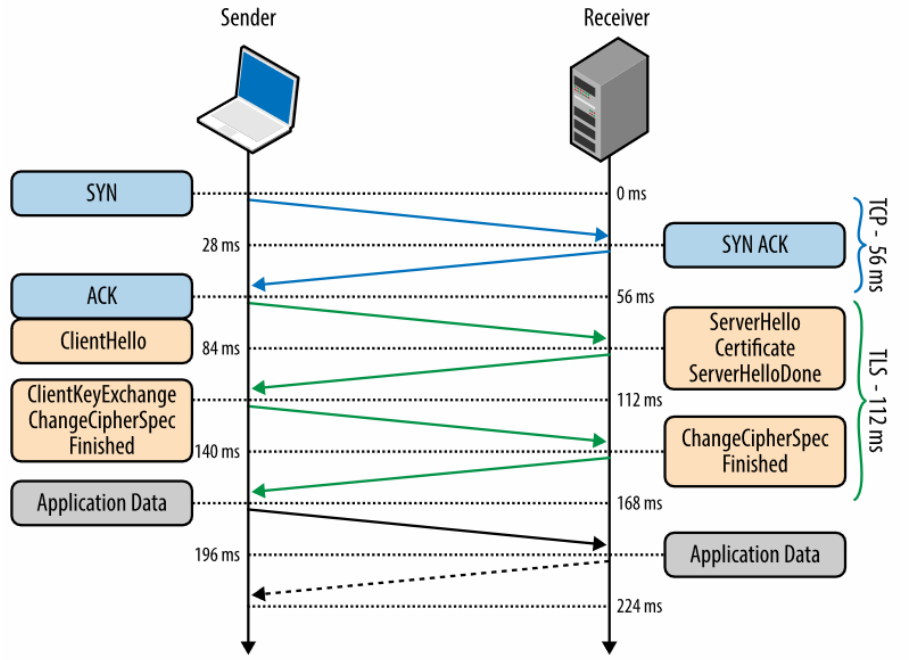


Figure 2.1: TLS protocol, with the TCP handshake in blue, the handshake protocol in orange and the record protocol in gray

In the handshake protocol, the parties negotiate the settings that will be used for the duration of the session. These settings define which cipher, message authentication and key distribution algorithm will be used, along with other configurations. During this phase, the parties will also exchange their certificates to share their public keys and agree on a session key. In the record protocol, the parties will start securely exchanging messages, using the settings defined in the handshake.

TLS communications have the following properties:

- Confidentiality using symmetric cryptography to cipher exchange data. The key used for ciphering is the shared key, which is different for each session.
- Authentication through the use of asymmetric cryptography.
- Integrity through the use of message authentication codes.

- Optionally, perfect forward secrecy to ensure that any future disclosure of keys cannot be used to decipher past communications.

2.3.2 History

TLS was not defined from the ground but is instead an upgrade of the now deprecated SSL.

SSL was first released by Taher Elgamal and Netscape, in 1995 as SSL 2.0, since SSL 1.0 was never publicly released as it had serious security flaws. In 1996, SSL 3.0 was released, as version 2.0 still had many flaws. SSL 3.0 was redesigned by Paul Kocher, but it was still was not robust enough and is deprecated as of 2015.

In 1999, TLS 1.0 was released as an upgrade to SSL 3.0. The differences between both are not huge, and it is even possible to perform attacks to downgrade TLS 1.0 into SSL 3.0. Much later, in 2006, TLS 1.1 was defined, and it added protection against attacks to CBC mode ciphers.

TLS 1.2 was then defined in 2008 and it is a major upgrade. It specified acceptable hash and signature algorithms, added support for authenticated ciphering, and added AES cipher suites, SHA-256 hashes, and elliptical curve cryptography, along with other changes. TLS 1.2 is currently the most used version of TLS.

TLS 1.3 was recently released, in 2018, and it removed support for weaker or deprecated algorithms and weaker elliptical curves. It also integrated the use of session hashes and changed the structure of ciphersuites.

The next subsections will focus on the inner workings of TLS 1.2.

2.3.3 Records

All data exchanged in a TLS session is framed in well-defined structures, called records. The records are used in both the handshake and the record protocol. There are 4 different types of records: the handshake record, the alert record, the ChangeCipherSpec record, and the application record. Each record is used for a different purpose, but all have the same header and similar structures [3]. Figure 2.2 shows the general structure of a record.

The header is composed of 5 bytes, where byte 0 is the content type, bytes 1 and 2 are the legacy version, and bytes 3 and 4 are the message length. The content type is used to differentiate the purpose of the record and its structure.

For handshake records, byte 5 identifies the message type, bytes 6 to 8 the message data length, and bytes 9 to the last, the message data. Message type will be further explained in the next subsection. These are only used in the handshake protocol and their content type value is

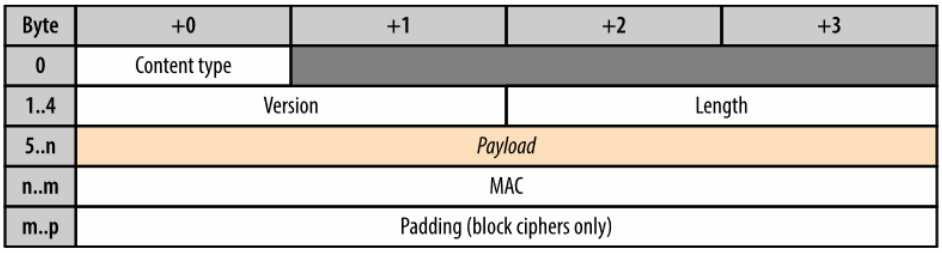


Figure 2.2: General TLS record structure

22.

Alert records are only used if an error occurs. Bytes 5 and 6 represent the level and description of the error, respectively. The level field indicates the severity of the error, which can be "warning" or "fatal". These records can also have a MAC and padding. The content type value for these records is 21.

ChangeCipherSpec records only use byte 5. It indicates the protocol type, which can only be 1, currently. These are only used in the handshake protocol and their content type value is 20.

Application records have the application data, the MAC, and padding and are only used in the record protocol. The content type value for these records is 23.

2.3.4 Handshake Protocol

The handshake protocol is responsible for making the entities agree on the specifications that will be used to protect the communication between the entities [3]. These specifications include the algorithms that will be used in the message exchanges and the definition of the session key [4].

The handshake protocol has an initial negotiation phase, where the specifications are agreed upon. After agreeing on those specifications, the entities exchange a final record, to assure that both are using the same specifications.

To make the negotiation easier, TLS created ciphersuites. A ciphersuite is a set of algorithms that will be used during the session. Since SSL 3.0, this set of algorithms is composed of three algorithms, each with its role: a key exchange algorithm, a cipher algorithm, and a MAC algorithm. The key exchange algorithm defines the session key that will be used for cryptographic operations, while the cipher and MAC algorithms define how the exchanged data will be protected and validated, to guarantee data integrity and confidentiality. A ciphersuite can also include a signature and an authentication algorithm.

Each cipersuite has a unique name that indicates its algorithmic contents. Each segment

of the name stands for a different algorithm type. An example of a ciphersuite name would be: TLS_DHE_RSA_WITH_AES_256_CBC_SHA256, where DHE_RSA is the key exchange algorithm, AES_256_CBC is the cipher algorithm and SHA256 is the MAC algorithm. All names follow the structure and algorithm order of the example used.

As previously mentioned, TLS 1.3 uses different ciphersuites. The main differences are that cipher and authentication algorithms were combined into authenticated encryption with associated data (AEAD) algorithms and that hash algorithms must now be used in HMAC-based key derivation (HKDF).

For the handshake, the client does not need to authenticate himself to the server, but if the client does so, it is mandatory for the server to also authenticate himself. The handshake protocol where only the server authenticates himself, depicted in Figure 2.1, proceeds as the following:

- Negotiation phase:

    1. ClientHello: the client sends the highest TLS version supported, a random number, and the list of supported ciphersuites and compression methods.
    2. ServerHello: the server responds with the highest version that can be support by both, a different random number, and the selected ciphersuite and compression method.
    3. Certificate: the server sends its certificate. Depending on the ciphersuite selected, this may be omitted.
        (a) The client verifies the certificate. If the certificate is valid, the client extracts the public key of the server. If not, the handshake ends there, and the protocol fails.
    4. ServerKeyExchange: the server sends its Diffie-Hellman key. Only used for DH Ephemeral (DHE) or DH anon (DH_anon) ciphersuites.
    5. ServerHelloDone: the server indicates that he is done with the handshake negotiation.
    6. ClientKeyExchange: the client sends, depending on the selected ciphersuite, a PreMasterSecret, his public key, or nothing. If it sends the PreMasterSecret, it will also cipher it using the public key of the server.
        (a) Now that both parties have the 2 random numbers and the PreMasterSecret, they compute the session key using a pseudo-random function using those parameters.
    7. ChangeCipherSpec: The client sends the ChangeCipherSpec record. This informs the server that from now on, the client will only send ciphered and authenticated records, as per the negotiated settings.

8. Finished: the client sends the Finished record that contains a hash and MAC of all the previously exchanged messages.
    (a) The server verifies both the hash and the MAC of the Finished record. If one of the verifications fails, the handshake fails and is terminated.
9. ChangeCipherSpec: The server sends the ChangeCipherSpec record. It proceeds identically to the client.
10. Finished: the server sends its Finished record.
    (a) The client verifies the Finished record the same way as the server does.

- Application phase: at this point, the handshake is complete, and all messages exchanged between the parties will be ciphered and authenticated the same way as the Finished record.

The handshake where the client also authenticates himself is very similar to this one. The only differences are that the server requests the client certificate, using the CertificateRequest message, before the ServerHelloDone message and the client sends its certificate after the ServerHelloDone message using the Certificate message. After the ClientKeyExchange, the client also sends a CertificateVerify message, which contains the signature of the previously exchanged messages. If the client certificate is indeed his, the server will be able to verify the signature using its key, proving that the client owns that certificate.

To improve the handshake protocol, TLS 1.3 only has one round trip, instead of two round trips [5]. To achieve this, the client tries to guess the key exchange protocol that the server will likely select and sends the key or secret that will be used in that protocol along with the ClientHello message. The server then proceeds as normal but before the ServerHelloDone, it also sends its Finished message.

2.3.5 Record Protocol

The record protocol starts as soon as the handshake protocol ends [3]. As said previously, only now do the client and server start to exchange application data between themselves.

The record protocol is responsible for taking the messages that are going to be transmitted, fragment the data into manageable blocks, compress the data, if required or enabled, apply a MAC, encrypt the data and send the result. When receiving data, the opposite operations are executed, i.e. the data is decrypted, verified, decompressed, if applicable, reassembled and then delivered to the client.

It is also at the beginning of the record protocol that the keys that will be used in the encryption and MAC algorithm are derived. As mentioned above, after the premaster secret is established, a pseudo-random function (PRF) will be used to generate the master secret.

The master secret is then partitioned into blocks that will form the client write MAC key, the server write MAC key, the client write key, the server write key, the client write IV, and the server write IV. The partitions of the master secret always correspond to the key or IV following the order described above. The client and server write IVs are created only if the encryption algorithm needs to use them.

The maximum allowed fragment length is 2^{14} bytes (16KB). Through the use of the TLS extensions, the communication peers can define a different value for the maximum fragment length during the established session, but that value must never be bigger than 16KB. This implies that a message with a bigger size than the maximum fragment length must be sent in multiple records and reconstructed when received.

When using stream cipher modes or the CBC block cipher mode, the MAC of the record is generated by concatenating a sequence number, associated with the record, to the values of the type, protocol version, message size and the message itself. The MAC algorithm will use the respective write MAC key. AEAD ciphers guarantee data integrity when performing the cipher operations and thus generate the MAC differently.

The encryption only occurs after the MAC is computed. This technique is called MAC-then-encrypt. This is technique has been proven secure for certain combinations of encryption and MAC algorithms but does not guarantee security in general [6]. This is the default technique used by TLS since it was considered secure at the time of the original SSL protocol.

To provide further security to the records, a TLS extension was created that enables the encryption-then-MAC technique to be used. This technique has been proven secure for any pair of encryption and MAC algorithms, if the former is secure against chosen plaintext attacks and the latter is secure against chosen-message attacks.

Using this technique the data within the record is first encrypted and then MAC is calculated and concatenated to the record. For TLS 1.1 and newer versions, the MAC is calculated using the same fields as the previous technique but their values are encrypted, except for the sequence number. The IV is also included in the MAC [7].

2.3.6 DTLS and Other Variations

TLS is used over TCP, but it can also be used over UDP, which is described in DTLS. It provides similar security guarantees as TLS since it is a stream-oriented version of it [8].

Since DTLS uses UDP, it has to deal with all of its disadvantages and constraints like packet reordering, loss of datagrams, and issues concerning data larger than the size of a datagram. But there are also advantages in using DTLS. Applications have lower delays since they are using a stream protocol and DTLS avoids the "TCP meltdown problem" when being used to create a VPN tunnel [9].

There are also other TLS variations such as the opportunistic TLS or the use of resumed TLS handshakes over regular TLS [4]. Opportunistic TLS is an extension in plaintext communication platforms, which offers a way to upgrade plaintext connections to use a TLS connection, instead of using separate ports for ciphered communication. Opportunistic TLS is mainly used for the communication protocol of email clients.

Resumed TLS handshakes provide a secure shortcut of the regular handshake protocol. This way the entities can avoid expending computational power for operations related to asymmetric ciphers. Resumed handshakes are achieved using resumed sessions, which are implemented using session IDs or session tickets. Simply put, these IDs or tickets contain the specifications of the previous session and they allow the negotiation phase of the handshake protocol to be skipped.

2.3.7 Vulnerabilities and Common Attack

Like every other protocol, TLS is not perfect, and it is possible to perform attacks and exploit vulnerabilities in order to hijack connections [10]. Some vulnerabilities lie in conceptual flaws of the TLS standards, such as protocol downgrade and connection renegotiation.

Protocol downgrade attacks, such as the FREAK, the POODLE, or the Logjam attack, aim to downgrade the TLS version of a connection into older, insecure versions. This allows attackers to have access to the data being exchanged.

Connection renegotiation attacks, such as the triple handshake or the Renego MITM attack, aim to perform a man-in-the-middle attack to inject packets in the connection or force a renegotiation with the server in order to impersonate a legitimate user [11].

There are also attack vectors that exploit the use of weaker cryptographic primitives in a TLS session. TLS versions up to 1.2 still allow the use of these weaker primitives, which means that some services are still vulnerable despise performing the protocol correctly [12].

Finally, since there is not a defined implementation of the protocol, there can be implementation mistakes that create vulnerabilities that attackers can exploit. For example, the TLS/SSL implementation provided by OpenSSL had a serious vulnerability that gave rise to the Heartbleed bug [13]. This vulnerability allowed attackers to steal private keys from servers that were using this TLS implementation.

2.3.8 Protocol Version 1.3

As mentioned before, the TLS 1.3 version was released in 2018. This new version of TLS aimed at improving some security issues present in version 1.2 and also improve the performance of the protocol. Some of the major differences between the version 1.2 and 1.3 are [5]:

- Only Authenticated Encryption with Associated Data (AEAD) algorithms are allowed, removing support from all the legacy symmetric encryption algorithms.
- The ciphersuite concept has changed. Now the authentication and key exchange mechaniscs are separated from the record protection algorithm and a hash algorithm used in the key derivation function and handshake message authentication code.
- All ciphersuites must provide perfect forward secrecy. This means static RSA and DH suites were removed.
- A zero round-trip time (0-RTT) mode was added. This saves a round trip at connection setup, at the cost of some security properties.
- All handshake messages after ServerHello are now encrypted.
- The key derivation functions have been redesigned.
- Some superfluous messages such as ChangeCipherSpec were removed
- Elliptic curve algorithms are now in the base spec, and new signature algorithms, such as ECDSA, are included.

Although this version of the protocol improves security and performance, compared to version 1.2, it is still not as widely supported as its predecessor and thus it is still more common to use version 1.2 instead of version 1.3.

2.4 Computing Platforms

A computing platform is an environment in which a piece of software is executed. It may be a piece of hardware, an operating system (OS), or even a web browser, as long as code is executed within it. Computing platforms are important tools for software development since they can either constrain or assist in the performance of the software. SoCs (System on a Chip) and FPGAs (Field-Programmable Gate Array) are examples of computing platforms.

An SoC is an integrated circuit, also known as chip, that integrates all or most of the components of a computer or other electronic systems [14]. These components usually include

a CPU (Central Processing Unit), memory, input and output ports, and secondary storage. As the components are all integrated into a single substrate or microchip, SoCs consume much less power and take much less space than other multi-chip equivalent designs. SoCs are most commonly used in mobile systems and IoT.

An FPGA, as the name implies, is an integrated circuit designed to be configured by a customer or designer after being manufactured. The FPGA configuration is generally specified using a hardware description language (HDL). FPGAs contain an array of programmable logic blocks and a hierarchy of "reconfigurable interconnections" that allows the blocks to be connected in various manners, creating different configurations. The logic blocks can be configured to perform complex combinational functions or can simply act as logic gates. The logic blocks also contain memory elements.

By combining an SoC and an FPGA into a single device, it is possible to create another computing platform, called SoC FPGA [15]. Besides integrating the functionalities of both components, an SoC FPGA also provides higher integration, lower power consumption, smaller board size, and higher-bandwidth communication between the components than other architectures that would use the components separately.

Currently, there are 3 major manufacturers of SoC FPGAs: Intel, which acquired the original manufacturer, Altera; Xilinx; and Microsemi. Each company has a wide array of products.

SmartFusion2 is the latest product that was developed by Microsemi. It was designed to be used in critical industrial, defence, aviation, communication, and medical applications [16].

The SmartFusion2 includes capabilities that protect designs against tampering, cloning, overbuilding, reverse engineering, counterfeiting, and differential power analysis (DPA) attacks. Users may also use built-in cryptographic processing accelerators, including AES-256, SHA-256, and 384-bit elliptical curve cryptographic (ECC) engine, and a non-deterministic random bit generator (NRBG) [17].

Xilinx also developed a product that assists the performance of the handshake protocol, called TLS Handshake Hardware Accelerator [18].

The TLS Handshake Hardware Accelerator is a secure connection engine that can be used to offload intensive public key operations, such as Diffie-Hellman (DH) key exchange and signature generation and verification. It combines a load dispatcher with several instances of another product, also developed by them, the Public Key Crypto Engine. The number of Public Key Crypto Engine instances is configurable.

The major features provided by the engine include support to the RSA algorithm and algorithms that use RSA keys, such as DH and DH Ephemeral (DHE), all algorithms that use

Elliptic Curve (EC) Cryptography, such as ECDSA, ECDH, and ECDHE. It also features Mbed TLS integration and high performance on off-the-shelf FPGAs.

Intel provides various SoC FPGAs, with the Intel Stratix 10 SX being the most used for embedded applications. It combines a quad-core ARM Cortex–A53 MPCore hard processor system with the Hyperflex FPGA Architecture developed by Intel to deliver the necessary embedded performance, power efficiency, density, and system integration [19].

Besides SoC FPGAs, Intel implemented an instruction set that improves the speed and security of applications that use the AES algorithm to perform encryption or decryption operations. This implementation is known as the Advanced Encryption Standard New Instructions (AES-NI).

AES-NI is currently supported by many different processors, mainly Intel and AMD ones. The AES-NI is comprised of six new instructions that perform several computational intensive parts of the algorithm [20]. The new instructions are:

- AESENC: Instruction used to perform a single round of encryption, by combining the ShiftRows, SubBytes, MixCollumns and AddRoundKey steps of the AES algorithm into a single instruction.

- AESENCLAST: Instruction used for the last round of encryption. It combines the ShiftRows, SubBytes and AddRoundKey into a single instruction.

- AESDEC: This instruction performs a single round of decryption, by combining the InvShiftRows, InvSubBytes, InvMixCollumns and AddRoundKey steps of the AES algorithm into a single instruction.

- AESDECLAST: Instruction used for the last round of decryption. It combines the InvShiftRows, InvSubBytes and AddRoundKey into a single instruction.

- AESKEYGENASSIST: Instruction used for generating the round keys used for encryption.

- AESIMC: Instruction used for converting the encryption round keys to a form usable for decryption using the Equivalent Inverse Cipher.

The improvement in performance gained from using the AES-NI depends on the conditions that it is being used, but for non-parallel modes of AES operation, such as CBC-encrypt, it can be expected to improve the performance 2 to 3 fold. This is assuming the AES implementation uses a software-only approach.

Besides improving performance, AES-NI helps to address recently discovered side-channel attacks on the algorithm. This is due to the instructions being performed completely in hardware, removing the need for software lookup tables.

Although SoC FPGAs are emphasized in this section, they were not used in this work, as it was never possible to obtain and use them.

Chapter 3

State of Art

This section contains an overview of existing TLS implementations and libraries, as well as a more in-depth analysis of the Mbed TLS library, followed by an overview on TLS acceleration. Besides implementations and libraries, this section also covers some tools that can be used to check the correct use of TLS sessions, both in the client and server endpoint, the robustness of TLS implementations, and the performance of the protocol.

3.1 Implementations and Libraries

As mentioned previously, the TLS protocol is a standard defined by the IETF (Internet Engineering Task Force) and as such, there is no default implementation of it. By following those standards, many companies, organizations, and development teams created their own TLS implementations or libraries.

A library is a collection of implementations of behaviour that has a well-defined interface by which the behaviour is invoked, while implementation is a realization of a technical specification or algorithm as a program, software component, or another computer system. Contrary to the code of an implementation, which is structured to only work within a certain program, the code from a library must be organized in a way that it can be used by different programs that have no relation or defined structure between each other. For example, Bouncy Castle and the Java Secure Socket Extension (JSSE) only provide a TLS implementation, while LibreSSL, Mbed TLS, and wolfSSL provide a TLS library.

LibreSSL is an open-source library, that was developed by the OpenBSD Project [21]. After the Heartbleed bug present in OpenSSL was discovered, OpenBSD Project forked LibreSSL from OpenSSL 1.0.1g, in April 2014, with the goals of modernizing the codebase, improving security, and applying development best practices. Like OpenSSL, LibreSSL was developed using C.

One of the first changes performed by the OpenBSD Project was the removal of insecure and obsolete code from OpenSSL. In June 2014, several other OpenSSL vulnerabilities were made public, none of which affected LibreSSL. It currently has support for TLS 1.0 to 1.3 and DTLS 1.0.

WolfSSL is also an open-source library, that was developed by Todd Ouska [22]. It is the predecessor of yaSSL (yet another SSL) that was created in 2004. WolfSSL features an embedded lightweight SSL/TLS library written in ANSI C that primarily targets embedded, RTOS (Real-Time Operating System), and resource-constrained environments. The library also includes an OpenSSL compatibility interface that includes the most commonly used OpenSSL functions. It also has support for various platforms, along with hardware cryptography and acceleration for some of them. Currently, it supports TLS 1.0 to 1.3, DTLS 1.0 and 1.2, and SSL 3.0.

Mbed TLS, previously known as PolarSSL, is an open-source library, that was developed in a collaborative project managed by ARM Holdings [23]. It is the official fork continuation of the XySSL. XySSL was first released in November 2006 by Christophe Devine, but, as of 2008, he was no longer able to support it and, thus, allowed Paul Bakker to create the official fork, named PolarSSL. In November 2014, PolarSSL was acquired by ARM Holdings. In 2011, the Dutch government approved an integration between OpenVPN and PolarSSL named OpenVPN-NL to be used for the protection of government communications up to the level of "Restricted". As of version 1.3.10, PolarSSL was rebranded into Mbed TLS to better show its fit inside the Mbed ecosystem. Mbed TLS was developed in C and currently supports TLS 1.0 to 1.2, DTLS 1.0 and 1.2, and SSL 3.0, although it is disabled by default.

Mbed TLS is divided into various modules, each with its own purpose. The modules are the following: TCP/IP communication, SSL/TLS communication, X.509 (a certificate format), random number generation (RNG), hashing, and encryption/decryption. The hashing and encryption/decryption modules contain the implementations of their respective cryptographic algorithms. The TCP/IP and SSL/TLS communication modules provide a communication channel for the SSL/TLS communication to use and the SSL/TLS communication itself, respectively. The X.509 module provides support for reading, writing, and verifying certificates with that format, and the RNG module provides random number generation.

To allow the separation of its modules, the library makes use of C macros. Each macro can define an implemented module, a functional part it or a configuration option that improves or adds new functionality to a program. These macros are then included in a configuration file that defines which modules or functionalities of the library will be used in a program or project.

The encryption/decryption and hashing modules use a generic structure that serves as a

wrapper for algorithms they implement. The generic structures contain a vtable (virtual table) and data fields that are necessary for characterizing the behaviour and specifications of an algorithm. Each algorithm type, such as hashing, symmetric cipher and key distribution has its own specific structure.

A vtable is a mechanism that allows non-object-oriented languages to support polymorphism by storing function pointers and defining the data types of the inputs and outputs of each function in a data structure. To call a stored function, a program only needs to access the vtable as it would access a regular data field. By using this mechanism, it is possible to implement the functions necessary to perform each algorithm separately and save the pointers to each function, along with the data that specifies the algorithm, in a wrapper. That wrapper represents the implemented algorithm and is then added to a list of wrappers of the same algorithm type. Because these wrappers are generic structures, it is possible to add new algorithms to the library or simply change the implementation of the already existing ones without changing the source code. To do so, it is necessary to disable the old macros and define new ones that will include the new implementations, in the configuration file.

The SSL/TLS communication module implements the SSL and TLS protocols. To do so, there is a sub-module that implements TLS generic functions, along with two sub-modules that implement the client and server-specific functions. This allows the library to be used only to create a TLS client or server. The client and server sub-modules are responsible for performing each phase of the handshake protocol. During the handshake protocol, a structure is created in the general sub-module that is used to save all the settings that the client and the server are trying to agree on. After finishing the agreement, the settings saved in that structure can no longer be changed and are then assigned to a new structure. This new structure also contains all the data necessary to perform the record protocol using the settings defined in the handshake, such as the wrapper structure of each algorithm specified in the ciphersuite. The operations regarding the records and the record protocol are also performed by the general sub-module.

Some libraries, such as Mbed TLS, also provide support for accelerating the TLS protocol. TLS acceleration is a method of offloading processor-intensive operations to a hardware accelerator. These operations are usually public-key cryptographic operations, such as asymmetric ciphering or signing, that are performed at the beginning of the TLS handshake since they require much more computational power than other operations, such as symmetric ciphering or hashing. This, however, does not mean that lighter operations cannot be offloaded [24].

The process of allowing Mbed TLS to use hardware acceleration is very similar to that of changing the implementation of a module or part of it. It is simply needed to provide the new

implementation that uses those external hardware accelerators and change the configuration file to use the new implementation, instead of the one provided by the library. Some hardware acceleration engines require an initial setup to be done on the platform before they start working. With that in mind, Mbed TLS also provides functions that can be used to set up and tear down the platform. By default, they do nothing, but using the module replacing method it is possible to provide the desired functionality to those functions. The setup and teardown functions must be the first and last functions to be called, respectively, if used.

Due to all of the capabilities explained above, Mbed TLS was chosen as the research target for this work. By taking advantage of the modularity provided by the library, it will be possible to reduce the size of the compiled code as much as possible. The library will also ease the creation of new implementations allowing a better fine-tuning of the algorithms that will be used and improving their performance when integrated with hardware accelerators.

3.2 Performance Evaluation Tools

First and foremost, a TLS implementation must provide secure communication, but that does not mean that performance or resource management is negligible. Some devices, such as mobile or IoT ones, have limited access to resources, precluding the use of some more resource-heavy TLS implementations. Besides the use of resources, there is also a need for implementations to be fast since there are systems that rely heavily on the use of the protocol. As such, there is a need to create metrics that allow for the evaluation of the performance of the TLS protocol [25, 26].

The most common metric that is used is the time certain operations take to do or the number of operations made in a certain time interval. For example, Mbed TLS also includes a set of test suites and benchmarking tests. These tests evaluate the performance of the algorithms implemented in the library using similar metrics to the ones mentioned above.

For symmetric ciphering and hashing algorithms, they measure the number of clock cycles the processor performs per byte, in cycles/byte, and the amount of data that can be ciphered and deciphered or hashed in one second, in KiB/s. For key generation algorithms, they measure the number of public and private keys that can be created in one second, in public/s and private/s, respectively. For key distribution algorithms, they measure the number of handshakes that can be made in one second, in handshake/s. Finally, for signing algorithms, they measure the number of signing and verifying operations that can be made in one second, in sign/s and verify/s, respectively. Since the implementation of those algorithms can be changed, it is possible to benchmark different implementations using only these provided tests [27].

Another useful metric for evaluating TLS is the amount of power required to perform the algorithms used by the protocol. To evaluate this, the Embedded Microprocessor Benchmark Consortium (EEMBC) developed a software that benchmarks the performance and energy cost of certain cryptographic operations required for the TLS protocol. The EEMBC is a non-profit, member-funded organization formed in 1997 to create standard benchmarks for the hardware and software used in embedded systems. Since its foundation, it has created various benchmarks, each with a different purpose or to be used in a different environment, including the SecureMark-TLS.

SecureMark-TLS is a benchmarking tool that was created using the SecureMark framework and it focuses on the analysis of TLS for IoT [28]. It measures the performance and energy consumption of a physical device for a prescribed set of cryptographic functions. These functions are the ones considered common for ciphersuites used in IoT devices and are comprised of ECC and ECDSA on the NIST secp256r1 curve, SHA256, and AES-128 in CCM/ECB mode. The energy measurements are then aggregated into a final score that is representative of the TLS operations. Other measurements, such as the size and security robustness of the implementation are described in a disclosure report. This disclosure report also includes a description of all relevant implementation details, such as the hardware device tested, the software library version used, compiler options and flags, and hardware crypto engine details if applicable.

The SecureMark-TLS software consists of a host PC application and an embedded DUT (device under test) software. The host application executes the benchmark by using the testbed hardware boards to send commands to the DUT for it to perform cryptographic operations. It then receives the results of those operations and obtains the power and timing measurements from the Energy Monitor. Since the implementation of the cryptographic operations can be any combination of hardware and software, the EEMBC only defines an API, that needs to be implemented according to the combination used. The tool is currently being worked on to provide benchmarking for more algorithms, such as the Chacha20-Poly1305 cipher algorithm and the Ed25519 digital signature algorithm.

Another interesting metric would be the RAM consumption of the operations executed by the protocol since IoT devices have limited access to that resource. This is a less used metric and there are currently no tools that measure it, although the SecureMark-TLS also plans to include that analysis in future upgrades.

Despise also having no support for RAM benchmarking, Mbed TLS provides a way to reduce the RAM and ROM footprint of the library. This is done by changing the values of some macros present in the configuration file. This reduces the amount of resources that will be used to the

minimum required by a program. The default values for those macros are the maximum values that the library can support.

Chapter 4

Proposed Solution

This section contains a description of the proposed tool and resulting tool, based on the objective and requirements that were given in the introduction. The description goes over all the functionalities and capabilities of the tool, as well as the methodology that will be used to evaluate and assure the correct working of the tool.

4.1 System Architecture

TLS operations can be quite taxing on a device and hinder its performance. Particularly mobile and other weaker IoT devices suffer even more from this problem since they have restricted access to or limited resources. By properly configuring the TLS session, it is possible to enable the use of the protocol in those devices or considerably improve the performance of TLS. This tool will allow its users to select a better TLS configuration for the device, taking into account the resources available to the device.

As mentioned previously, there are many versions of the TLS protocol, but this tool will mainly consider the configurations available to TLS 1.2. This is due to the lack of support for TLS 1.3, which was only recently developed, and TLS 1.1 or lower versions, which are currently considered insecure and, thus, deprecated.

In the TLS context, the configuration refers to the ciphersuite that will be used during a session. The ciphersuite specifies which algorithms will be used by the communication peers, which will also specify the security services that the session will guarantee since the security services are achieved through the use of certain algorithms. Table 4.1 shows a list of all security services and the list of respective algorithms, implemented in Mbed TLS [29], that can be used to provide them. The table does not include the use of AEAD modes for symmetric cipher algorithms. It also includes algorithms that are no longer considered safe to use, such as DES,

RC4, MD5 and 3DES-EDE.

Security Service	List of implemented algorithms
Authentication	ECDSA, PSK, RSA
Confidentiality	AES, ARIA, CAMELLIA, DES, RC4, 3DES-EDE
Integrity	MD5, SHA, SHA256, SHA384
Key Establishment	DHE, ECDH, ECDHE, PSK, SHA256
Perfect Forward Secrecy	DHE, ECDHE

Table 4.1: List of security services and the respective algorithms that provide them.

The tool is composed of two major modules, the data acquisition module and the data analysis module. As the name implies, the data acquisition module is responsible for generating all the data related to the metrics that the user wants to know, while the data analysis module is responsible for generating statistics and plots using the acquired data. All of the generated statistics and plots will then be used by the user to decide which configuration better suites its needs. Fig. 4.1 better illustrates the architecture of this tool.

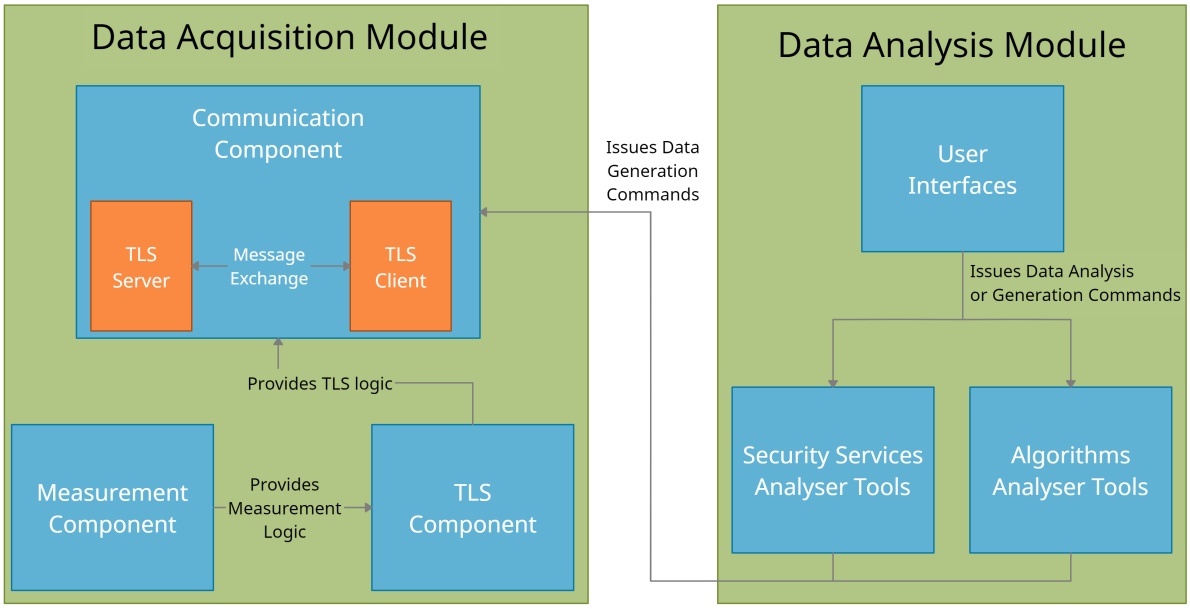


Figure 4.1: Architecture of the tool

Even if connected, the two modules can be used separately, i.e. the data can be acquired in the target device and then be analysed in another device.

To use the data acquisition module, the target device only needs to be able to establish TCP sockets, so that the server and client can execute the TLS protocol, and create CSV files, as that is how the generated data is saved. Using the default configuration this module occupies around

2.4 MB of memory. This value can be reduced by removing unnecessary capabilities from the module.

There are no special requirements to use the data analysis module save for the graphical user interface (GUI). The device must support graphical applications in order to use it.

The data acquisition module was mainly written in C code and is composed of various components, namely the measurement component, the communication component and the TLS component.

There are various reasons as to why C was chosen as the programming language, but the main one is because the TLS library that was chosen as the research subject of this project is Mbed TLS. Since we were using Mbed TLS, we wanted to make it easy to integrate the TLS library component, which uses the library, with the other components which meant that they also needed to be written in C code.

The data analysis module was mainly written in Python code and it contains various programs that will assist the users in analysing the data collected by the acquisition module. This module also contains an User Interface (UI) from where the user can issues commands to the tool. This module was written in Python since it is a programming language that is easy to learn and use and it also contains a lot of libraries that were created with the purpose of analysing data, shortening the development time of this module dramatically.

Two different methodologies can be used to analyse the data. The first is by grouping the data by algorithm, while the second is by grouping the data by security service. Although these methodologies are similar, they allow the user to choose which one better meets his interests.

For example, the user might not have restrictions on which algorithms he can use and, as such, is only focused on the impact caused by each security service. Alternatively, he might want to focus solely on the cost in performance that each algorithm has when being used.

Using these two methodologies, the algorithm data grouping and the security service data grouping, the user has access to much more detailed information since they complement each other.

4.2 Data Acquisition Module

This subsection focuses on explaining how the Data Acquisition module was developed, the reasoning behind the approaches that were used during development as well as other alternatives that were considered and how the components that constitute this module communicate and interact with each other.

4.2.1 Measurement Component

The measurement component is the one responsible for implementing all the behaviour, data structures and functionality that is needed to make measurements. For this work, a metric refers to a type of data that is measured using a specific measurement tool. As such, if multiple tools measure the same data type, those data types will be considered different metrics. This component must allow the users to do the following:

- To use multiple measurement tools at the same time
- To measure all the enabled metrics
- To allow a measurement tool to measure different metrics, if possible

In order to meet all of the requirements listed above, it was decided that this component should use a structure similar to the one used by Mbed TLS. As such, each metric is implemented separately in its own module. Lastly, there is a wrapper module that serves as an interface for all the metric modules and the main module that uses the wrapper module to perform the measurements themselves. Figure 4.2 better illustrates how the modules are connected.

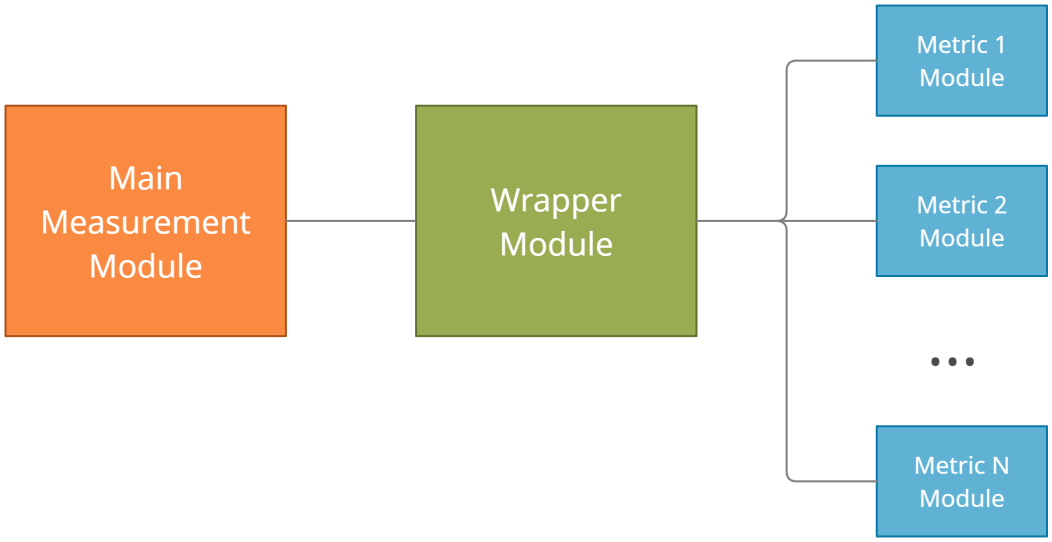


Figure 4.2: Architecture of the Measurement Component

All the metric modules follow a specific structure, consisting of a data structure that will store two measured values, since this component obtains the value of the metric by subtracting two measured values, and functions that allow the following:

- Allocate the memory needed to store the measured values
- Measure the values and store them in the data structure

- Calculate the metric value
- Reset or free memory in the data structure

The wrapper module serves two major purposes. The first one is to create a virtual table for each metric module, storing all the implemented functions as pointers and the second one is to convert a generic pointer into the corresponding metrics data structure. This will facilitate the addition, removal or change of metrics that will be used since new metrics can be created by simply implementing a new module following the structure mentioned above and creating the corresponding virtual table in the wrapper module.

The main module is the only one in this component that directly interacts with the other components. It contains all the functions needed to perform the measurements and save the acquired values in a file, as well as a list of generic pointers that can be converted into the metrics data structure.

Each metric module is also associated to a macro. These macros will allow the users to select which metrics to use, by enabling or disabling the macros in a configuration file. If a macro is enabled, the corresponding module will be compiled as well as the virtual table in the wrapper module, allowing it to be used by the main module. The opposite will occur if the macro is disabled.

To get the values of the metrics, it is necessary to perform two measurements. The first one is before the execution of what wants to be evaluated and, the second one is after the execution. The first measurement is called the starting measurement while the second one is called the ending measurement. Then, the metric value is calculated by subtracting the value of the starting measurement from the value of the ending measurement.

This component allows for either all the measurements to be made first and to then calculate and save the values in a file or to calculate and save the values after two measurements are completed. The values are saved in a CSV (Comma-Separated Value) format as it would make it easier to parse the data from files in this format when performing the analysis of the data.

This component was implemented in this way since it would allow its users to perform the measurements using their preferred method and since it would give them the most freedom when instrumenting the other components with the calls from the main module.

The currently implemented metrics are: number of virtual CPU cycles measured using the PAPI library, time using the PAPI library, in microseconds, and time using the time.h standard library from C.

PAPI (Performance Application Programming Interface) is a tool used to measure the performance of programs and applications by accessing the performance counter hardware found in

most major microprocessors. PAPI is a reliable tool that results in near real-time, can be used to generate many different metric values and is easy to use.

PAPI is only being used to acquire the number of cycles and time needed to perform a cryptographic algorithm since it was not possible to add more metrics due to time constraints. Although PAPI is a great tool, it has limitations. One of them is that it is machine-dependent and, as of version 3.7, does not work on Windows architectures as its components no longer target Windows. The PAPI version that is currently being used is 6.0.0.

To make up for this issue, it was decided that a metric that would be OS independent should be created, even if the values it would acquire would be less precise than the ones acquired from PAPI. Thus it was decided that a time metric would be created using the time.h library. Since time.h is a standard C library it can be used on any system that can also run and compile C code.

For the purpose of this work, calls from the main module from this component were integrated into the TLS component to measure the performance of algorithms used during a TLS session. The next subsection further explains how the integration was done and how the measurements were made.

4.2.2 TLS Component

The TLS component is the one responsible for implementing the TLS protocol as well as all the cryptographic algorithms that it uses. This component also contains the measurement functions that were mentioned in the previous subsection.

Since implementing the TLS protocol and the cryptographic algorithms would take a massive development time and this work does not have the purpose of implementing the protocol, it was decided that a library that already implements the protocol and the cryptographic algorithms would be used. The System Architecture subsection already explains the reasons as to why Mbed TLS was chosen.

As the TLS library is already implemented, the only thing left to do is to place the measurement functions. Depending on where they are placed, the obtained data can be noisy as it is possible to get the metric values for a single function call as well as for a block of code that contains multiple complex functions. In this work, the purpose is to evaluate the performance of the individual cryptographic algorithms as they mostly make up the configuration of the TLS session and, thus, majorly influence the performance of the protocol.

As mentioned in the “Implementations and Libraries” subsection, Mbed TLS is divided into various modules where each implements an algorithm or a component needed for TLS to work.

Taking this into account, some of the approaches that could be used to instrument the library with measurement functions that were considered were:

- Placing the measurement functions in the algorithms respective module
- Placing the measurement functions in a module that communicates with all the cryptographic modules

The first approach has the advantage that it is easier to find the places where to put the measurement functions. One would only need to go to the module of each algorithm and place the measurement functions at the start and end of each relevant function. The relevant functions are only the ones that involve the cryptographic procedure, as the purpose of some functions is to allocate memory, make some minor verifications or decide which relevant functions need to be used.

Although simple, this approach has two major disadvantages. The first one is that since it needs more instrumenting than the second approach, it increases the code size of the component much more and is also more susceptible to mistakes such as misplacing the measurement functions or not placing them at all. The second one is that if new cryptographic algorithms are implemented they need to be instrumented anew. This is relevant since it is a possible method to create different implementations of the same algorithm, which will be further explained later.

The second approach is the opposite of the first one, as it is needed to analyse the workflow and data flow of Mbed TLS to know where to instrument the library, making it harder to execute than the first approach, but creating lower overheads in the code and making it possible to create new algorithms without the need to instrument them. This is due to the communication with other modules being done via the wrapper structures. One only needs to find where the wrapper functions are being used to instrument multiple algorithms at once.

This approach, besides being harder to execute, has the issue that it can measure some irrelevant operations along with the cryptographic functions since the instrumentation needs to be placed in a module that calls the functions of the algorithms posteriorly.

Taking both approaches into account, it was decided that the second one is the most suited as it is the one that takes the most benefits from the mechanisms used to implement the library. After analysing the workflow of the library, it was decided that the best module to place the measurement functions in, is the SSL/TLS communication module [30].

This module is the central point of this library, making use of the cipher, hashing and public key module to implement the TLS specific behaviour, such as the handshake protocol. As mentioned in the “Handshake Protocol” subsection, a premaster secret and master secret are

formed during the handshake.

The premaster secret is used to provide greater consistency between all TLS ciphersuites since each key exchange algorithm uses different cryptographic materials to form the premaster secret. Table 4.2 shows all the key exchange algorithms along with the respective cryptographic material needed to form the premaster secret [3, 31, 32, 33].

Key Exchange Algorithm	Premaster Secret Material	Details
PSK	N || 0^N || N || PSK	PSK: pre-shared key, N: size of PSK (in octets), 0^N: zero octets, N times
RSA	R	R: 2-byte version number and 46-byte random value
RSA-PSK	48 || R || N || PSK	R: 2-byte version number and 46-byte random value, PSK: pre-shared key, N: size of PSK (in octets)
DHE-PSK	M || Z || N || PSK	Z: DH session key, M: size of Z (in octets), PSK: pre-shared key, N: size of PSK (in octets)
DHE-RSA	Z	Z: DH session key
ECDH-RSA	X	X: x-coordinate of the ECDH session key
ECDH-ECDSA	X	X: x-coordinate of the ECDH session key
ECDHE-PSK	L || X || N || PSK	X: x-coordinate of the ECDH session key, L: size of X (in octets), PSK: pre-shared key, N: size of PSK (in octets)
ECDHE-RSA	X	X: x-coordinate of the ECDH session key
ECDHE-ECDSA	X	X: x-coordinate of the ECDH session key

Table 4.2: List of key exchange algorithms and the material each respective algorithm needs to form the premaster secret.

After establishing the premaster secret, the master secret is generated by applying a pseudo-random function (PRF) to the premaster secret, along with the two random numbers that were exchanged in the Client and Server Hello messages. Later in the handshake, the master secret is used to generate the keys used for the cipher and hash operations of the session data.

The messages exchanged by the communication peers, during the handshake, vary depending on the chosen key exchange algorithm. Taking into consideration the TLS standards, the SSL/TLS communication module implements the particular interactions between the communication peers as followed [3, 31, 32, 33]:

- PSK, RSA-PSK, ECDH-RSA, ECDH-ECDSA: The ServerKeyExchange message is not sent. The other algorithms will send this message.

- RSA, RSA-PSK: The random value is generated by the client, which is sent in the ClientKeyExchange message. This message is also encrypted using the public key of the server.

- DHE-RSA, ECDHE-RSA, ECDHE-ECDSA: The ServerKeyExchange message contains a signature of the server DH or ECDH parameters and the random values of the communication endpoints. The signature uses the private key of the server.

- DHE-PSK, ECDHE-PSK: The ServerKeyExchange message may not include the server identity hint field.
- PSK, RSA-PSK, DHE-PSK, ECDHE-PSK: The CertificateRequest message is not sent. The other algorithms may send this message.

In this module, all the cipher and hashing algorithms are called via the wrapper structures and the calls from the public key module are well contained and defined. These characteristics help to limit the overhead placed on the code by the instrumentation, as well as the measurement of irrelevant operations as much as possible.

The code created by the instrumentation is all associated with four macros that can be enabled or disabled in a configuration file, to make it easy to identify the code that was added to the library and to allow the user to select which type of algorithms he wants to evaluate. Three of those macros are associated with the algorithms types, meaning there is one for cipher algorithms, one for hashing algorithms and one for key exchange algorithms.

The last macro is associated with the measurement of the handshake protocol including the irrelevant operations. Although this is not the purpose of this work, it was decided that acquiring this data could be relevant to users as the handshake protocol contains many irrelevant operations and their sum significantly influences the performance of the TLS protocol.

Cipher and hashing algorithms are only evaluated during the record protocol, as it is during that phase of the TLS protocol that they influence performance the most. The SHA-2 algorithms are the only ones that are evaluated in the handshake protocol, but only when being used as pseudo-random functions to generate the master secret, from the premaster secret, and to derive the keys used in the record protocol. Cipher modes that use AEAD algorithms are not being analysed as they use structures and mechanisms that are different from regular cipher algorithms and could not be analysed properly due to time constraints.

The public key module implements two types of algorithms: signing algorithms, such as RSA and ECDSA, and key exchange algorithms, such as DHE, ECDH and ECDHE. The signing algorithms also use wrapper structures, while the key exchange algorithms do not follow a specific structure.

Taking all of this into account and following the second approach, the instrumentation of the algorithms in the cipher and hashing module and the signing algorithms in the public key module was done by finding the calls from the wrapper structures in the SSL/TLS module. In turn, for the key exchange algorithms and the SHA-2 algorithms when used for key derivation, it was needed to find the relevant function calls. This was done by following the stack of functions calls, during runtime.

To ensure that the measurement was done correctly, global variables that represent the state of the measurement component were created. The states are "currently working" and "not working". If the current state is "currently working" and a new starting measurement is going to be done, an error occurs and the execution of the program is terminated. Likewise, if the current state is "not working" and an ending measurement is going to be done, the program is also terminated with an error.

The instrumentation was also done in a way that allows for all metric values to be collected first and, only when the algorithm is completed, saved in a file. By instrumenting the SSL/TLS communication module this way, the overall performance of the algorithms used during the session suffers the least possible change.

Since this work also has the aim of enabling the use of hardware accelerators to assist and improve the execution of the TLS protocol, a method needed to be created that would allow just that. There were two methods considered for this task:

- Using the alternative implementation mechanism of the Mbed TLS library
- Creating the new implementations as new algorithms and add them to the library

The first method is the simplest as it is already a mechanism that is implemented in the library itself. As mentioned in the "Implementations and Libraries" subsection, Mbed TLS allows its users to create an alternative implementation of an algorithm by replacing relevant functions in its module or by replacing the whole module through the use of macros in a configuration file. The downside of using this method is that the user cannot change the implementation it wants to use during runtime, since only one of the implementations can be compiled. This means that if a user would like to, for example, compare the performance of a certain algorithm when using hardware accelerators against the native Mbed TLS implementation, he would need to compile and run his program twice.

The second method works on the idea that an alternate implementation of an algorithm is a "new" algorithm. This would allow the user to compile both implementations at the same time. To create an alternate implementation using this method, one would need to create a new module following the structure of the module of the original implementation, create its respective virtual table in the corresponding wrapper module and, finally, create new ciphersuites, in the ciphersuite module, that would use that algorithm.

Although this method allows more flexibility than the first method, it has some issues when trying to provide alternative implementations to algorithms that do not have a wrapper structure. Since those algorithms, such as DHE or ECDHE, do not use a wrapper structure the

functions of the new implementation module must be integrated directly into the SSL/TLS communication module. This demands a deeper understanding of the structure used by Mbed TLS, thus being more prone to errors. Using this method, new implementations for these algorithms also need to be instrumented as they create new cryptographic functions in the SSL/TLS communication module.

4.2.3 Communication Component

The communication component is the one responsible for implementing the means to enable the evaluation of the TLS protocol. That is to say that this component implements not only a server and client program but is also responsible for creating all the keys and certificates necessary to perform the protocol.

The client and the server are both implemented using the TLS component and make use of a configuration file in order to select which features need to be used. This structure allows both this component and the TLS component to share the same settings. It also helps to reduce the size of the compiled code by only enabling and compiling the necessary TLS and algorithm modules and features. This configuration file imports the configurations used by the measurement component to enable the use of the measurement functions in the TLS component.

Before starting the execution of the TLS protocol, the client and server need to perform some setup operations. The communication endpoints also need to enable the handshake and record protocols to be executed multiple times to generate the data more effectively. With these requirements in mind, the endpoints were implemented following these steps:

1. Parse some session parameters sent as user input. These parameters will be further explained in this subsection.

2. Seed the random number generator used to generate the messages that will be exchanged. This allows multiple message sizes to be tested during the same runtime.

3. The server will create and bind a TCP socket to a certain IP and port.

4. Load the certificates of the opposite endpoint. The server only performs this step if the client is going to authenticate itself to the server.

5. Load the keys and certificates of the corresponding endpoint. The client only performs this step if it is going to authenticate itself to the server.

6. Setup the settings of the TLS session. These settings refer to the parameters received as input and the keys and certificates that were loaded in the previous steps.

7. Perform the handshake protocol in a loop:
   (a) Reset the session state
   (b) The server waits for and accepts a valid client connection, while the client creates a socket and connects to the server
   (c) Perform the handshake protocol
   (d) Verify the certificate of the opposite endpoint. The server only performs this step if the client authenticated himself with a certificate of his own.
8. Perform the record protocol in a loop:
   (a) The client generates a request, while the server generates a response. This is done using the random number generator that was seeded in Step 2.
   (b) The client sends its request and the server receives it
   (c) The server sends its response and the client receives it
9. Close the connection with the opposite endpoint
10. Show the status of the last connection that was done

As mentioned above, the user needs to send some parameters to these programs for them to work properly. These parameters are: ciphersuite, sec_lvl, max_sec_lvl, msg_size, max_msg_size, n_tests, path and debug_lvl. Ciphersuite is the only parameter that is mandatory and it serves to indicate which ciphersuite will be used for the TLS session. The debug_lvl parameter can only be used if the debug module is enabled. It serves to set the level of debug logs, ranging from 0 to 5, where 0 is no logs and 5 is the most logs [34]. Table 4.3 contains a description and limit value of all the optional parameters.

Parameter	Default Value	Description
sec_lvl	0	Minimum security level that will be used
max_sec_lvl	4	Maximum security level that will be used
msg_size	32 B	Smallest message size (in bytes) that will be used
max_msg_size	1 GB	Largest message size (in bytes) that will be used
n_tests	20000	Number of tests that will be executed
path	ddMMyyyy.hhmm	Name of the directory where the data will be saved

Table 4.3: List of the parameters used by the communication endpoints, their default value and description.

The server and client will execute the handshake protocol using all the security levels within the range of sec_lvl and max_sec_lvl and will execute the record protocol using messages with base two exponential sizes within the range of msg_size and max_msg_size.

The sec_lvl, max_sec_lvl, msg_size and max_msg_size parameters can all be set within the default values shown in Table 4.3, while n_tests can be any integer bigger than 0 and path can be any string that does not contain any special symbol. The data is always saved in a directory defined in the configuration file, so the path parameter serves to create a sub-directory to allow multiple sets of data to be saved simultaneously.

The security provided by the algorithms used in the handshake protocol is not set. By using the algorithms with bigger keys it is possible to provide more robust security. The security level is a value that represents the size of the cryptographic keys, and thus the degree of security, that will be tested during the data acquisition.

Table 4.4 shows all the security levels considered in this work, along with their security strength and the corresponding key size for all the algorithms used during the handshake protocol. The security strength is a number associated with the amount of work that is required to break a cryptographic algorithm or system. In this work, it is measured in bits and security strength of 80 bits or lower, thus a security level 0, is no longer considered sufficiently secure [35]. The minimum required security, as per the current TLS standards, corresponds to security level 1.

Security Level	Security Strength (in bits)	Key size (in bits)		
		PSK	RSA, DHE	ECDSA, ECDH(E)
0	80	80	1K = 1024	192
1	112	112	2K = 2048	224
2	128	128	3K = 3072	256
3	192	192	7.5K = 7680	384
4	256	256	15K = 15360	521

Table 4.4: Security levels with their corresponding security strength and key sizes for all algorithms used during the handshake.

It was decided that the communication endpoints would use the localhost as the IP and 8080 as the port number of the TCP sockets. This is because the focus of this work is, again, to measure the performance of the cryptographic algorithms. Thus the impact in performance the program suffers from sending or receiving messages is irrelevant.

Some TLS ciphersuites make use of certificates to authenticate the communication endpoints. As such, if the ciphersuite requires the use of certificates, they must be generated and loaded into the endpoints before the start of the TLS protocol.

The generated certificates are all self-signed. This means that they do not use the private key of a certificate authority (CA) to generate a digital signature and, instead, have a digital signature generated using the private key of the certificate itself.

Although it is more correct to use a CA certificate and build a proper certificate chain, this work does not focus on analysing the impact in performance created by verifying a certificate chain. Additionally, Mbed TLS allows the use of self-signed certificates since self-signed end-entity certificates can be used as trust CA certificates [36]. As such, it was decided to use self-signed certificates as it would be simpler and would save development time.

The RSA certificates and keys that are loaded before the start of the TLS session were generated using OpenSSL. Although Mbed TLS has key and certificate generating capabilities, it cannot generate RSA keys bigger than 8192 bits (or 8 Kb). As such, OpenSSL was used. The ECC certificates were generated using Mbed TLS.

The certificates and keys that are loaded at the start of the programs do not influence the size of the DHE or ECDHE key that will be generated. By default, Mbed TLS uses a 2048 bit key for DHE and a 521 bit key for ECHDE. To properly evaluate the security levels for DHE and ECDHE, the endpoints force the use of externally generated DHE parameters and elliptic curves already implemented in Mbed TLS, respectively. The DHE parameters were generated using OpenSSL since Mbed TLS cannot generate parameters bigger than 8192 bits.

4.3 Data Analysis Module

The next subsections focus on explaining how the Data Analysis module was developed. It includes the reasoning behind the approaches that were used as well as other alternatives that were considered. The subsections will also explain the purpose of the various tools that were developed to analyse the performance of the algorithms used and the impact of the security services provided by each algorithm or ciphersuite.

4.3.1 Security Services Analysis

As mentioned in the "System Architecture" subsection, the data analysis can be done by grouping the data into the security services provided by each algorithm or ciphersuite. Each algorithm can provide one or more security services.

With this in mind and taking into account that each algorithm in Mbed TLS is implemented through the use of many functions, it is possible to assign each individual function to a security service.

Another approach to this data grouping would be to assign a single security service to an algorithm, but using the method described above, it is possible to get a clearer understanding of the impact each algorithm has regarding a specific security service.

For example, DHE provides the key establishment service, as would DH, but it also provides

the perfect forward secrecy service. If this algorithm were to be assigned to a single service, the data that would be analysed regarding that service was going to be polluted since it would also contain data regarding a different service.

Following the first approach, all the functions in each Mbed TLS algorithm module were analysed, regarding their purpose in implementing the algorithm and assigned to a security service. Table 4.1 located in the "System Architecture" subsection shows all the security services that each algorithm provides.

Taking this data grouping into account, some analysis tools were developed to give the user all the necessary information to decide which ciphersuite better provides the security services the user deems relevant. The developed tools were: profiler, comparator, analyser and calculator.

These tools were all developed using Python. They all follow a specific procedure, except for profiler:

- Parse and group the data that will be used.
- Filter the data outliers from the data population using z-scores. This step is optional.
- Calculate the statistics from the remaining data.
- Save the statistics in a file.
- Generate plots using the statistics.

In statistics, the z-score is the number of standard deviations by which an observed value is above or below the mean value of what is being measured. Observed values above the mean have positive z-scores, while those below the mean have negative z-scores. It is calculated using the following equation:

$$z = \frac{x - \mu}{\sigma} \tag{4.1}$$

In this equation, z is the z-score, x is the observed value, μ is the population mean and σ is the standard deviation of the population. For this work, an observed value is considered an outlier if the modulus of its z-score is bigger than a weight parameter. By default this parameter is equal to 2, the user can input the value of this parameter.

All tools also use the same basic data grouping procedure. The idea of this procedure is to group the data by security service, metric, operation and id. The operation is the entity that performed the algorithm in case the service is provided in the handshake protocol, i.e "authentication", "key establishment" or "perfect forward secrecy", or the algorithm operation

if the service is provided during the record protocol, i.e "confidentiality" and "integrity". The id corresponds to the security level or strength of the keys used in algorithms during the handshake protocol and the size of the message used as input in the algorithms during the record protocol.

The generated statistics are always saved in CSV files since that format is easier to understand and use in external programs. Apart from data filtering and grouping, each tool performs each step in the procedure differently.

The comparator tool is used to compare the performance of individual algorithms within a certain security service. To use this tool, the user must first select the security services and the data set that are going to be analysed, and provide the tool with a file that contains the list of algorithms that are to be evaluated. The format of each line of the file needs to be "security service, algorithm". Algorithms that provide multiple security services need to be paired with all the services they provide in different lines.

The tool then performs the basic data grouping, and further groups the data by algorithm, i.e data from different ciphersuites are grouped if those ciphersuites use the same algorithm to provide a certain service. After filtering the data, the tool calculates the sample mean and standard deviation for each collection of data and then saves those values.

Lastly, the tool creates a bar plot for each combination of security service, metric and operation. Each bar represents the sample mean of the combination of an individual algorithm and data-id. Each bar also contains an error bar with the sample standard deviation of the corresponding algorithm. The bars are then grouped by their data-id. Figure 4.3 shows an example of a single bar plot produced by the comparator tool.

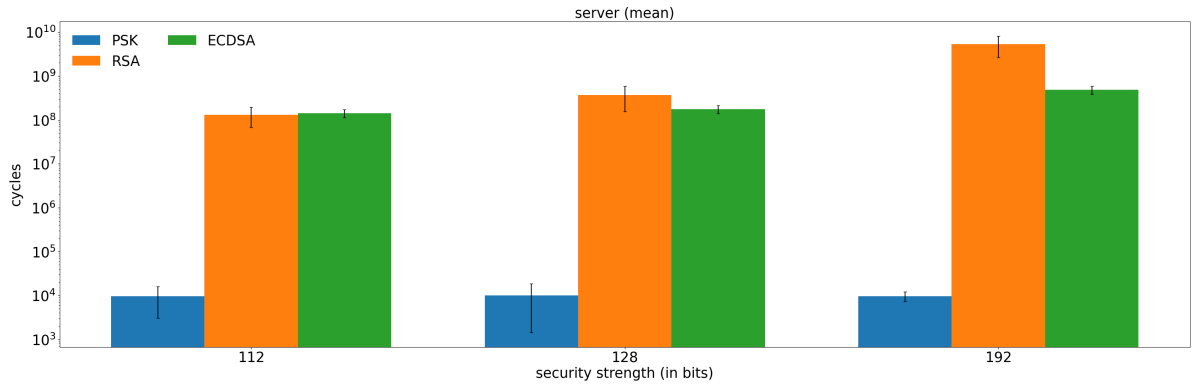


Figure 4.3: Bar plot showing the server-side performance of the algorithms that provide the Authentication service, in cycles

The analyser tool is used to analyse the performance of each ciphersuite when performing the handshake protocol. To use this tool, the user must first select the security services and the data set that are going to be analysed. The user can also include the performance of the whole

handshake protocol if he so desires. The tool then performs the basic data grouping, and further groups the data by algorithm and ciphersuite. The total handshake data is also grouped as if it were an algorithm. After filtering the data, the tool calculates the sample mean of each data group and saves those values.

Lastly, the tool creates a plot of stacked bars for each combination of operation, data-id and metric. Each stack of bars represents the total performance of the services provided by the ciphersuite. Each bar in the stack represents the performance of a different algorithm, within that ciphersuite, and the value of each algorithm is the sum of its sample mean from each security service. The total handshake value for each ciphersuite is represented in a single bar and placed in the background of the stacked bar of the corresponding ciphersuite.

The calculator tool is used to calculate the performance of each ciphersuite in the handshake protocol and record protocol. To use this tool, the user must first select the security services and data set that are going to be analysed. The tool then performs the basic data grouping, but, instead of grouping by security services, it groups the data by protocol, handshake or record, i.e only two groups are created and each group contains all the data regarding the security services that are provided in each protocol. The data is then further grouped by ciphersuite.

After filtering the data, the tool calculates the sample mean of each data group and saves those values. These statistics are then presented in tables, grouped by the data-id, that highlights the biggest and smallest values in each metric, per group, in red and blue, respectively. Each table contains the statistics of one protocol operation, i.e the server and client performance for the handshake protocol and the sending and receiving performance for the record protocol.

The profiler tool is used to automate the acquisition of data and the generation of statistics. To use this tool, the user needs to input all the parameters mentioned in the "Communication Component" subsection, select which security services are going to be analysed, including the total handshake performance, and provide the tool with a file that contains the list of algorithms that are to be evaluated. The format of this file is the same as the one used in the comparator tool.

The user can choose not to provide some parameters, in which case their default values will be used instead, as shown in Table 4.3. The user can also implement a custom version of the "Communication Component", and input its directory to this tool to use it instead of the one provided in this project. If the user chooses to do this, he needs to ensure that his version follows the same structure as the one used in the "Communication Component".

This tool then generates all possible ciphersuite combinations using the algorithms in the file and compiles the Data Acquisition module using a different thread. After the compilation

is done, the tool creates two threads, one for the server and one for the client, and executes the TLS protocol according to the parameters that were chosen. After the data acquisition is done, this tool will run all other tools in order to generate their respective statistics. The user can also choose to only execute the data acquisition procedure or only execute the statistics generation procedure, instead of executing both.

4.3.2 Algorithms Analysis

Besides security services, there is another grouping methodology that was used to analyse the data. By grouping the data into the algorithm type, it is possible to get more information regarding the performance of the algorithms themselves.

As mentioned in the "Handshake Protocol" subsection, there are three types of algorithms, namely cipher, MAC and key exchange, that compose a ciphersuite. By grouping the data by those algorithms, it is possible to make a direct comparison between them since they will have the same purpose within the TLS protocol. Table 4.5 shows the list of algorithm types and all the possible algorithms that belong in that category. All the listed algorithms are used in ciphersuites and implemented in Mbed TLS [29].

Algorithm Type	List of implemented algorithms
Cipher	AES, ARIA, CAMELLIA, DES, RC4, 3DES-EDE
MAC	MD5, SHA, SHA256, SHA512
Key Exchange	DHE-PSK, DHE-RSA, ECDH-ECDSA, ECDH-RSA, ECDHE-ECDSA, ECDHE-PSK, ECDHE-RSA, PSK, RSA, RSA-PSK

Table 4.5: List of algorithm types and the respective list of algorithms that belong to it.

AEAD modes for cipher algorithms are not being considered, as mentioned before.

Taking this grouping into account some tools were developed in order to help the user understand the performance of each algorithm and decide on which ciphersuite to use. The developed tools were: profiler, comparator and plotter. These tools were also developed in Python and follow the procedure described in the "Security Services Analysis" subsection. They also make use of z-scores to filter the data outliers, as described in the previous section.

A general data grouping procedure, used by all the tools described in this subsection, was also created. It follows the same logic as the one in the "Security Services Analysis" subsection, but instead of grouping the data by security services, it the data groups by algorithm type.

The purpose of the plotter tool is to thoroughly analyse the data set being used by generating various statistics that describe it. To use this tool, the user must first select the algorithm types and the data set that are going to be analysed. The tool then performs the basic data grouping, and further groups the data by ciphersuite.

After filtering the data, the tool calculates the sample mean, median, mode and standard deviation for each collection of data and then saves those values. Lastly, the tool creates three plots for each combination of algorithm type, metric and ciphersuite. The three plots are:

- A plot containing two scatter subplots. Each subplot shows the measured values relative to an algorithm operation, grouped by the data-id.
- A plot containing two error subplots. Each subplot shows the sample mean and standard deviation relative to an algorithm operation, grouped by the data-id.
- A plot containing three regular subplots. Each subplot shows the statistical values of each algorithm operation, for each data-id. The statistical values shown are the sample mean, median and mode.

The comparator and profiler tool from this section work mostly in the same way as the ones described in the "Security Services Analysis" subsection. The major difference between them is that the user must select which algorithm types are going to be analysed instead of the security services and the file provided must contain lines in the following form: "algorithm type, algorithm".

The other existing difference between tools is that the profiler tool does not analyse total handshake performance, while the comparator tool will generate a plot for each combination of algorithm type, metric and operation.

4.4 User Interfaces

All tools described in the previous subsections need a command line to interact with them. To make them more accessible to users, a GUI was also developed. Figure 4.4 shows the main window of the GUI.

As can be seen, the layout of the GUI is divided into two sections, the "Services" section and the "Algorithms" section. The "Services" section makes use of the tools described in the "Security Services Analysis" subsection, while the "Algorithms" section makes use of the "Algorithms Analysis" subsection. Each section contains a collection of checkboxes that are used to select which category of algorithms are going to be considered when using the tools. Both sections also contain a collection of buttons, each with its own purpose.

The "Edit Services" or "Edit Algorithms" button will make the edit window appear. The edit window is used to edit the files that contain the list of algorithms, which will be provided to the tools. In this window, the users will be able to check which algorithms each category contains

Figure 4.4: Main window of the Graphical User Interface

and will also be able to add new algorithms or remove existing ones. There is also a "Restore Defaults" button that will allow the user to restore the files to their original configuration. The existing categories in the "Services" section are the security services, while the categories in the "Algorithms" section are the algorithm types. The layout of the edit window can be seen on the left side of Figure 4.5.

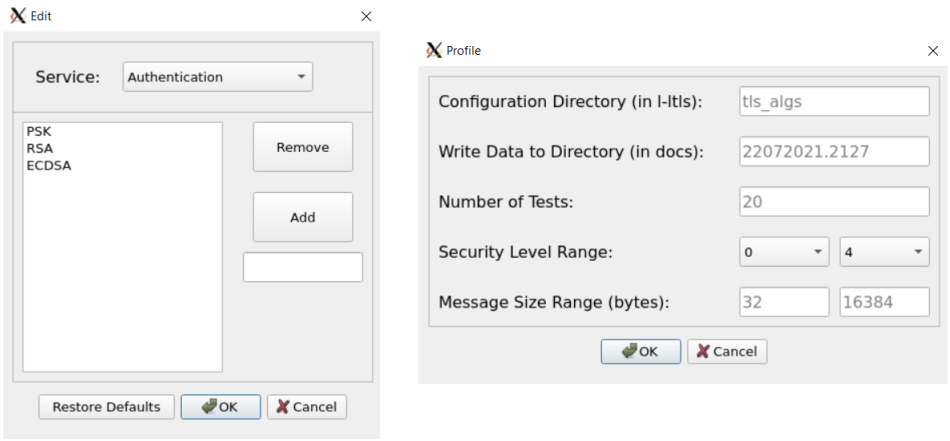


Figure 4.5: Edit and Profile window of the GUI

The "Acquire Data" buttons in the main window will make the profile window appear. The profile window is used to input the parameters needed to run the profiler tool from the respective section. The layout of the profile window can be seen on the right side of Figure 4.5.

After the user inputs the parameters, the GUI will run the data acquisition process. Upon conclusion, the user can either choose to run the data analysis process or not. If the user chooses to continue, the GUI will prompt the user to input the filter parameter and run the data analysis procedure.

In the "Services" section, the user will also be prompt if he wants to include the total handshake performance in the analysis. Additionally, after running the data analysis process the user will be prompt on which tables, from the ones generated by the calculator tool, he wants to see.

The "Generate Statistics" buttons in the main window will run the data analysis process from the profiler tool of the section. The GUI will prompt the user for a directory that contains the data set to be analysed and the proceeds the same as described above.

Through the GUI it is not possible to run the tools individually, as the GUI only interacts with the profiler tool. To use those tools individually, the user must use the command line. The tools also have a helper function that shows how to use them. Appendix C contains a user guide with all command line tools and how to use them.

Chapter 5

Results and Discussion

This section focuses on showing the capabilities of the tool that was developed. This will be done by using the tool to obtain and analyse data in two different scenarios. The section will also include a description of the tests that were made and an analysis of the obtained results.

5.1 Scenario 1: Analysis of the Security Services Provided by the Handshake Protocol

For the first scenario, it was decided to use the developed tool to analyse the performance of each security service provided during the handshake protocol. The services that can be provided during the handshake are authentication, key establishment and perfect forward secrecy. This is a relevant test since, in the TLS protocol, the handshake is the most taxing part of the session.

As mentioned previously it is during the handshake that the communication endpoints make use of asymmetric cryptography techniques, such as digital signatures, DH key generation and/or elliptic-curve cryptography, to authenticate themselves, generate symmetric keys and/or assure perfect forward secrecy. Each technique provides different security services and has a different cost associated with it. It is also possible to strengthen the security provided to the session by using the algorithms with keys bigger than the minimum required size.

For this scenario, each key exchange algorithm that can be used in a ciphersuite will be tested. Table 4.5 of the "Communication Component" section contains all the key exchange algorithms that were tested. The server and client authenticate themselves mutually, when possible, and used the same symmetric encryption and MAC algorithms.

Additionally, each key exchange algorithm was tested using keys that provide a security level of 1 to 3. Table 4.4 of the "Communication Component" section contains the respective key sizes for each algorithm. This range of security levels was chosen since level 1 is the minimum

required security [35] and level 4 is already excessive.

The services profiler tool was used to execute the tests and acquire the data. The number of tests executed for each combination of key exchange algorithm and security level was 30. After executing the tool using many different numbers of tests, this was the smallest number of tests needed to produce reliable data. The data acquisition module was only compiled once since the services profiler tool was used. This means that no data discrepancies were created by compiler optimizations.

The metric used was the number of CPU cycles used during algorithm execution, which was obtained using the PAPI library. The tests were executed using the VirtualBox software to emulate a pre-built Ubuntu (32-bits) image. The image was provided by SEED Labs [37]. This was done so because the native device cannot use PAPI and PAPI was the most precise measurement tool available.

The host device uses an Intel(R) Core(TM) i7-4720HQ processor [38] and the virtual environment made use of all 4 CPU cores. The tests also use the default filter weight mentioned in the "Security Services Analysis" subsection.

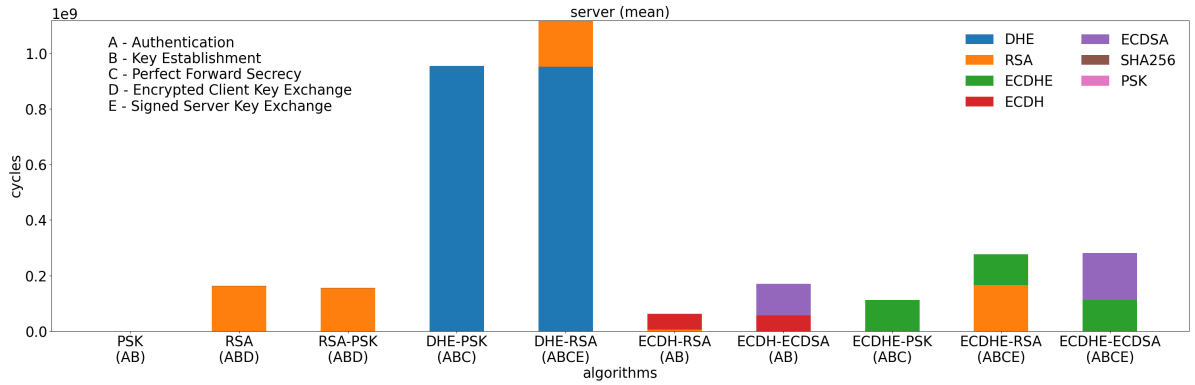
After getting the data, the services analyser and comparator tools were used to generate plots. The produced plots can be seen in the "Handshake Protocol Performance" section of the Appendix A.

Figures 5.1, A.1 and A.2 show the performance of each key exchange algorithm when using keys that provide 112, 128 and 192 bits of security, respectively. Each layer of a stacked bar represents the performance of an individual algorithm and each key exchange algorithm has an extra-label that indicates which security services are provided by that algorithm.

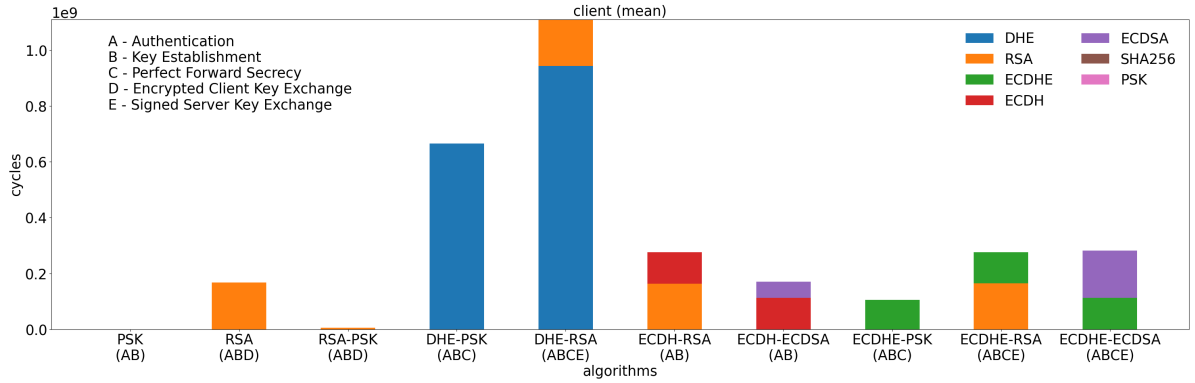
As can be seen from all those plots, the most taxing key exchange algorithm is DHE-RSA followed by DHE-PSK. The difference is even more apparent the lower the provided security strength is. This is due to the use of extremely large key sizes by DHE and RSA to produce equivalent security levels. DHE and RSA need to generate a key that is around 30 times bigger than the ones used by ECDHE and ECDSA to produce the same level of security.

Figures 5.2, A.3 and A.4 show the performance of each algorithm used to provide the authentication, key establishment and perfect forward secrecy services, respectively. The plot uses a logarithmic scale and groups the bars by security strength. The bars also contain an error bar corresponding to the standard deviation of the data sample.

As can be seen in Figures 5.2 and A.3, PSK is the least taxing configuration when used to provide authentication and key establishment, respectively. This is due to the operations that are used by this algorithm being mostly simple data reading and parsing. In Figure A.4, the



(a) Server-side



(b) Client-side

Figure 5.1: Performance of each key exchange algorithm for security strength of 112 bits, in number of CPU cycles.

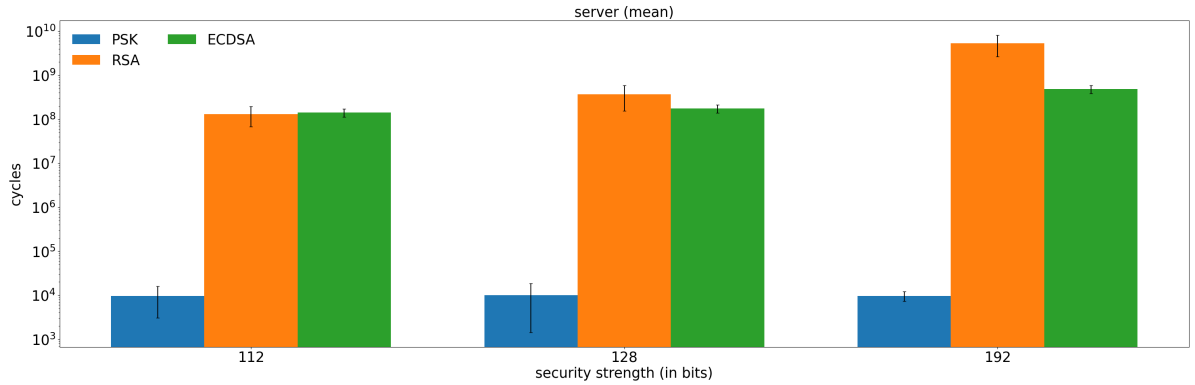
effect of the difference between DHE and ECDHE key sizes can be seen again.

All the produced statistics have expected values, relative to each other, except for the ones produced by DHE. The DHE bars for the Client-side of Figures 5.1 and A.1 show an abnormal difference in performance when using the DHE-PSK and DHE-RSA algorithms. This can also be seen by the large DHE error bars in the Client-side of Figure A.4.

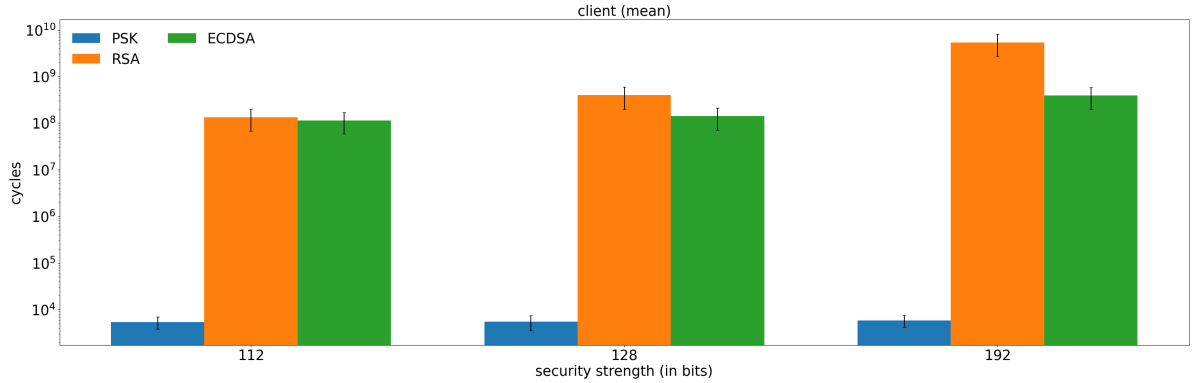
Upon further inspection of the data produced, it was discovered that the sample standard deviation for the client-side DHE-PSK data using security levels 1 and 2 and for the server-side DHE-PSK and DHE-RSA data using security level 2 was relatively high. This points out inconsistencies in the data itself. This is most likely due to the processor making unexpected calls when running tests for one of the ciphersuites as both key exchange algorithms use DHE in the same way, i.e. they make the same calls to the DHE module.

The statistics produced by the ECDHE and ECDH algorithms are reasonable since their values are similar for all ciphersuites that use them.

As for the ECDSA algorithm, the difference in values in Figures 5.1, A.1 and A.2 and the reasonably sized error bar in Figure 5.2 exists due to the fact that ECDHE-ECDSA performs



(a) Server-side



(b) Client-side

Figure 5.2: Performance of each algorithm that provides the authentication security service for all security strengths, in number of CPU cycles.

the signing of the ServerKeyExchange message, using ECDSA. That operation does not exist in ECDH-ECDSA since that ciphersuite does not send the ServerKeyExchange message as the ServerCertificate contains all the necessary information to generate the EC key.

More complex analysis is needed for the RSA algorithm. The RSA and RSA-PSK key exchange algorithms encrypt the ClientKeyExchange message using the server public key, while the DHE-RSA and ECDHE-RSA sign the ServerKeyExchange message using the private key from the server. The ECDH-RSA key exchange algorithm does not send the ServerKeyExchange message nor it encrypts or signs the ClientKeyExchange message. Additionally, the RSA, DHE-RSA, ECDH-RSA and ECDHE-RSA algorithms perform a signature in the CertificateVerify message, while RSA-PSK cannot perform this operation since the CertificateRequest message is never sent.

Therefore, the client-side in RSA-PSK ciphersuites only use RSA to encrypt the ClientKeyExchange message using the public key from the server. Meanwhile, server-side in ECDH-RSA ciphersuites only use RSA to verify the CertificateVerify message using the public key from the client. This makes RSA-PSK and ECDH-RSA ciphersuites the ones with the least taxing

use of the RSA algorithm for the client and server endpoint, respectively.

The above analysis can be seen in Figures 5.1, A.1 and A.2. The analysis also explains the error bars in Figure 5.2, since the bar is made from the accumulated use of the algorithm from all different ciphersuites.

The PSK and SHA256 have such little impact, relative to the other algorithms that they cannot be seen in Figures 5.1, A.1 and A.2. In Figures 5.2 and A.3 it can be seen that these two algorithms have a relatively high standard deviation.

Upon further inspection of the data produced, it was discovered that the sample standard deviation of many key exchange algorithms that use the SHA256 algorithm was relatively high. For the PSK algorithm, this issue is also seen in some key exchange algorithms. This points out inconsistencies in the data itself.

The cause of these values is likely the same as the one that caused data inconsistencies for the DHE algorithm. The inconsistencies in the data are also more visible for these two algorithms, since the scale of their values is much smaller compared to other algorithms and, thus, more easily influence the standard deviation.

In the case of the client-side PSK algorithm, the relatively large standard deviations are also due to PSK and RSA-PSK key exchanges not sending the ServerKeyExchange message while the DHE-PSK and ECDHE-PSK do so.

Concluding, the key exchange algorithm that provides the best trade-off between security and performance, assuming the objective is to have the most robust security possible, is the ECDHE-ECDSA. This key exchange algorithm provides all the security services, i.e. authentication, key establishment and perfect forward secrecy, and also adds an extra layer of security by signing the ServerKeyExchange message. Additionally, it uses the algorithms that have the best performance taking into account the services provided.

From all this data, it can also be concluded that, overall, ECC algorithms have much better performance, both in the server-side and client-side, since they can use much smaller keys to provide equivalent security levels when compared to the RSA or DHE algorithms.

5.2 Scenario 2: Comparative Analysis of Different AES and SHA-2 Implementations

For the second scenario, it was decided to use the developed tool to analyse the performance of different implementations of the AES and SHA-2 algorithms. This test was made to show that it is possible to improve or deteriorate the performance of the TLS session by using algorithm

implementations that make better or least use of the architecture of the device used to perform the protocol.

For this scenario, it was decided to use two different implementations of the algorithms being tested. For both algorithms, the first implementation of both algorithms is the native Mbed TLS implementation. This implementation will be mentioned as the native implementation for the duration of this section.

The second SHA-2 implementation was created using the code from an open-source project [39] and adapting it to be used in the Mbed TLS library. The second AES implementation was created using the example code found in the white paper that gives an overview of the AES-NI instruction set [20] implemented by Intel. This code was also adapted to be used in the Mbed TLS library.

The adaptations were made using the chosen alternate implementation method described in the "TLS Component" subsection. These implementations will be mentioned as alternate implementations for the duration of this section.

The algorithms profiler tool was used to acquire the data for this scenario. By only generating data relative to the record protocol, it is possible to directly compare the implementations of both algorithms. The records were secured using the default MAC-then-encrypt technique.

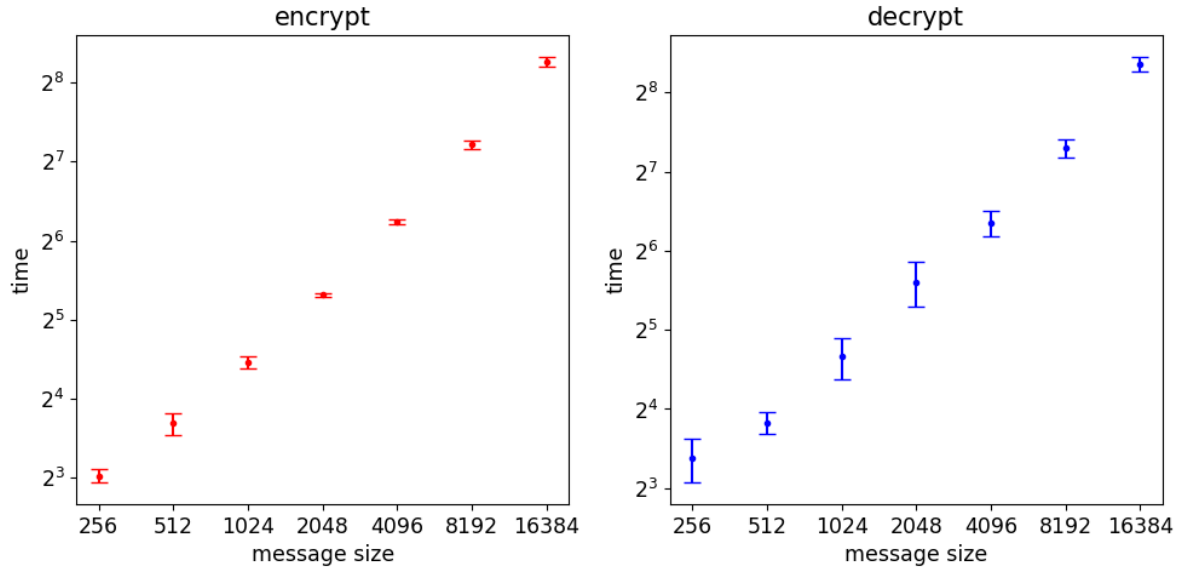
To generate the data, only the TLS-PSK-WITH-AES-256-CBC-SHA384 ciphersuite was tested. This ciphersuite is sufficient as it contains both the algorithms whose implementations are being compared. To get all the necessary data, the tool needed to be used twice, once for the native implementations and another for the alternate implementations. This is because Mbed TLS only allows the use of a single implementation of an algorithm at a time.

To get a better understanding of the performance of both implementations, various message sizes were tested. The tested message size ranges were from 256 bytes to 16384 bytes (16 KB), which is the maximum plaintext size that can be used in TLS [3].

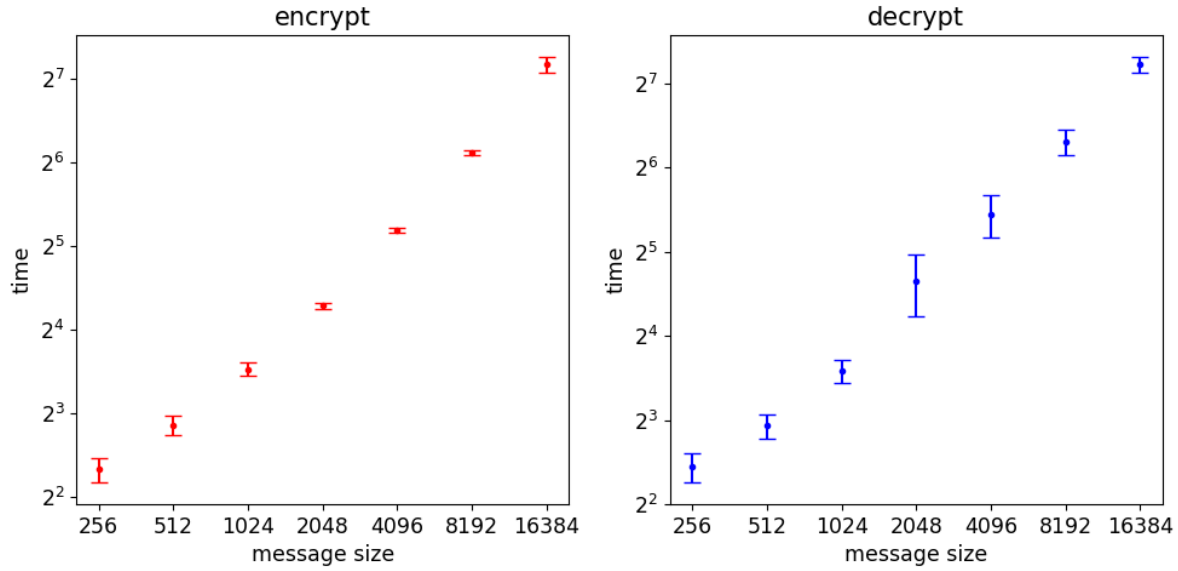
The number of tests that were executed for each combination of algorithm and message size was 400. After much experimentation, this is the minimum number of tests required to produce reliable data.

The metric used for this scenario was CPU time, in microseconds, using the time.h standard library from C. The tests were performed using the host device mentioned in the previous scenario. This is because the virtual environment that allows the use of the PAPI library does not have access to the AES-NI instructions. As such, the tests needed to be performed in the host device and use the time.h library to perform the measurements. The tests also use the default filter weight mentioned in the "Algorithms Analysis" subsection.

After getting the data, the algorithms plotter and comparator tools were used to generate plots. The produced plots can be seen in the "Software Implementations Comparison" section of the Appendix B.



(a) Native AES Implementation

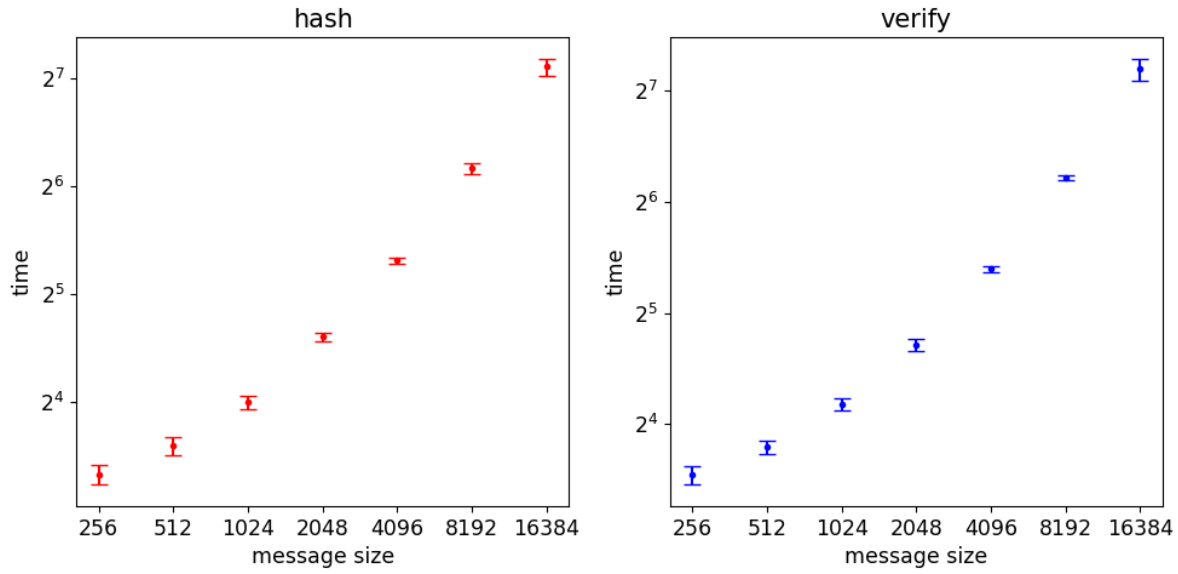


(b) Alternate AES-NI Implementation

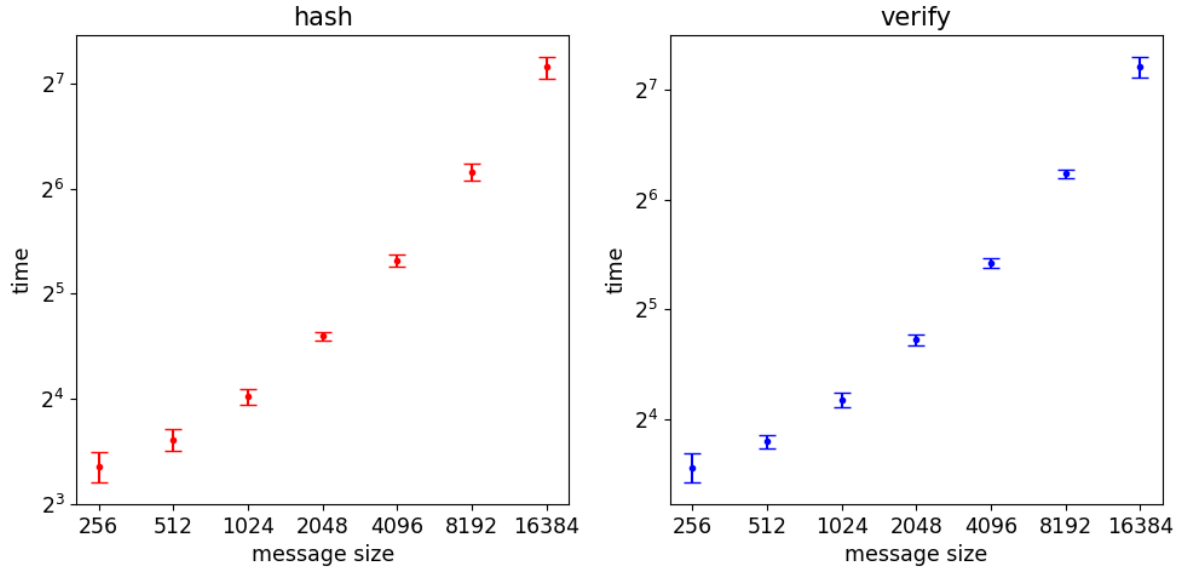
Figure 5.3: Mean and standard deviation of the AES operations for all message sizes, in microseconds.

Figures 5.3, B.1 and B.2 show the plots relative to the implementations of the AES algorithm, while Figures 5.4, B.3 and B.4 show the plots relative to the implementations of the SHA-2 algorithm. The data coloured in red represents the encrypt or hash operation, whereas the blue coloured data represents the decrypt or verify operation from the respective algorithm. For the

rest of this section, the encrypt and hash operations will be referred to as out operations while the decrypt and verify operations will be referred as to in operations.



(a) Native SHA-2 Implementation



(b) Alternate SHA-2 Implementation

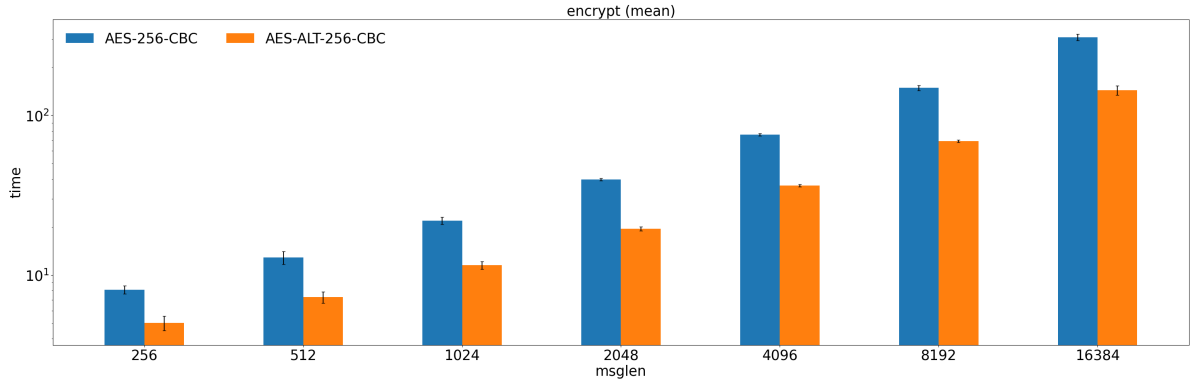
Figure 5.4: Mean and standard deviation of the SHA-2 operations for all message sizes, in microseconds.

As can be seen in Figures B.2 and B.4, the in operations are slightly slower than the out operations. For the decrypt operations only, this might be because the data set has higher standard deviations and the data points are also more dispersed compared to the ones from the encrypt operations, as can be seen in Figures 5.3 and B.1.

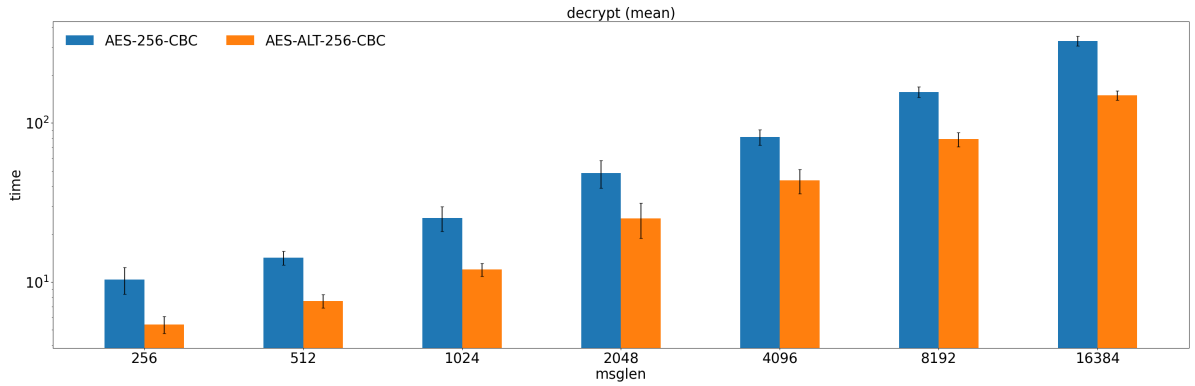
No proper reason could be pointed out as to why all decrypt operations have such values.

Every single operation is done separately so there is no interference from multiples operations being executed at the same time. Additionally, it was noted that this behaviour is persistent no matter the number of that were run when acquiring the data. Apart from this particularity, all the generated plots show reasonable results.

Figures 5.5 and B.5 show the performance of the AES and SHA-2 algorithms, respectively, using the native and alternate implementation. The native implementations are represented in blue while the alternate ones are represented in orange. All the plots present in both Figures use a logarithmic scale.



(a) Encryption Operation



(b) Decryption Operation

Figure 5.5: Comparison of the performance of both AES implementations for all message sizes, in microseconds.

As can be seen in Figure 5.5 the alternate AES implementation is a lot less taxing on the device, as it takes around half the time to perform cipher operations than the native implementation. This can also be seen when comparing the scale on of the plots in Figure B.2. This result is as expected since the alternate implementation uses the AES-NI instruction set which optimizes the use of the AES algorithm in Intel processors, such as the one used by the host device.

As can be seen in Figure B.5, the alternate implementation of the SHA-2 algorithm is

marginally slower than the native implementation, as there is not even that much of a difference between their performances. This result is within expectations as the SHA-2 alternate implementation only focused on implementing the algorithm and not optimize it.

Concluding, for this device, the best implementations, from the ones tested, is the alternate AES implementation and the native SHA-2 implementation. It can also be concluded that the AES-NI instruction set provides a great optimization for the performance AES algorithm of devices that use Intel processors.

5.3 The Tool

Although the objectives of each scenario are very different from each other, the tool contributed to the performance analysis of the available ciphersuites in each scenario to a great extend. In the first scenario, the objective is to compare the impact of different algorithms when providing certain security services, whereas the second one intended to make a direct comparison of the performance of different algorithm implementations.

By making use of different capabilities provided by the tool, it was possible to generate relevant and intelligible statistics regarding the performance of the algorithms that were used as well as the security services that they provide. All of the generated results and statistics are also reliable as they are within realistic expectations and are coherent between them.

The tool can also take into account different algorithm implementations. This allows its user to further improve the performance of the TLS protocol as they can analyse different implementations and choose the one that better suits the situation they find themselves in.

Concluding, the tool greatly contributes to selecting better TLS configurations by providing an extensive, relevant and intelligible performance analysis of all the tested algorithms as well as the security services provided by them. The tool can also be used in many different scenarios and even take into account different algorithm implementations.

Chapter 6

Conclusions and Future Work

This chapter contains the conclusions that were drawn from this project as well as the accomplishments that were made from it. The chapter also proposes some ideas to further improve this work or to extend its uses.

6.1 Achievements

This work proposes a tool that allows its users to get a detailed analysis of the TLS protocol, in all its phases. After providing the analysis, the tool creates a list of possible TLS configurations that can be used by a given device.

The tool provides a dynamic performance analysis as it allows its users to enable and disable the metrics that are going to be evaluated, as well as implement new ones. The analysis given by the tool is not only limited to the performance of algorithms that are used during TLS sessions, but also to the security services that are provided during the session.

Through this tool, devices, particularly IoT ones that have limited access to resources, can secure their communications by properly configuring TLS sessions. Although the main targets of this tool are IoT devices, other devices, that have access to more resources, can also use this tool to further increase the performance and/or robustness of their TLS sessions.

Currently, this work also provides the most extensive analysis of the Mbed TLS 2.16.5 library, which was used as a research target. The analysis includes a detailed explanation of the mechanisms used by the library, as well as its structure and its modules are connected.

This work also demonstrates the capabilities of the developed tool by using it to analyse the performance of the TLS protocol in two different scenarios. The first scenario focuses more on the analysis of the security services provided by the handshake protocol, while the second scenario focuses on the analysis of using different algorithm implementations, with the focus of

using the AES-NI instruction set developed by Intel.

6.2 Future Work

For future work, it would be interesting to extend the profiling capability of the tool by including other relevant metrics. As it stands, the tool can only generate data regarding time or CPU clock cycles. These metrics are strongly related and may not provide enough relevant information to allow its users to choose a good TLS configuration.

The most interesting metrics to include would be power consumption and memory usage. Both of these metrics are relevant because they greatly complement the information provided by time metrics. These are also two resources that are usually lacking in IoT devices, due to their nature, and can even, ultimately be the bottlenecks of those devices.

Another relevant study that this project did not cover, due to time constraints, would be analysing the use of AEAD ciphersuites within Mbed TLS. Although AEAD algorithms or AEAD cipher modes are more relevant for TLS 1.3, they can still be used in version 1.2 and are even supported by Mbed TLS. This study would require the analysis of the flow of the information flow of Mbed TLS when using such ciphersuites.

This study would allow to further increase the scope of possible configurations that devices may use and simultaneously strengthen the TLS sessions, since AEAD algorithms are considered safer than using both an encryption and message authentication algorithm.

Lastly, it would be interesting to use this tool to evaluate the performance of an actual IoT device, since the testing performed in this work was all done using a general-purpose computer.

Although the tool can also be used to profile the performance of general devices, its main focus is still to be used within an IoT environment and understand how it can truly benefit that device.

Ideally, only the Data Acquisition module would be put in the target device, as IoT devices have limited memory. After generating the data from that device and testing different configurations, the data can be transferred to another device where, the Data Analysis module would be used to generate statistics, plots and tables.

Bibliography

[1] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976. doi:10.1109/tit.1976.1055638.

[2] ISO/IEC JTC 1/SC 27. Information technology — security techniques — modes of operation for an n-bit block cipher. ISO/IEC 10116:2017, jul 2017.

[3] E. Rescorla and T. Dierks. The transport layer security (TLS) protocol version 1.2. IETF RFC 5246, aug 2008.

[4] D. Eastlake 3rd. Transport layer security (TLS) extensions: Extension definitions. IETF RFC 6066, jan 2011.

[5] E. Rescorla. The transport layer security (TLS) protocol version 1.3. IETF RFC 8446, aug 2018.

[6] H. Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is ssl?). *Proceedings of Crypto '01, Springer-Verlag LNCS No. 2139*, aug 2001.

[7] P. Gutmann. Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). IETF RFC 7366, September 2014.

[8] E. Rescorla and N. Modadugu. Datagram transport layer security version 1.2. IETF RFC 6347, jan 2012.

[9] O. Honda, H. Ohsaki, M. Imase, M. Ishizuka, and J. Murayama. Understanding TCP over TCP: effects of TCP tunneling on end-to-end throughput and latency. *Performance, Quality of Service, and Control of Next-Generation Communication and Sensor Networks III*, 2005. doi:10.1117/12.630496.

[10] Y. Sheffer, R. Holz, and P. Saint-Andre. Summarizing known attacks on transport layer security (TLS) and datagram TLS (DTLS). IETF RFC 7457, feb 2015.

[11] CVE-2009-3555. Available from MITRE, CVE-ID CVE-2009-3555, oct 2009. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2009-3555. Last accessed: 29.07.2021.

[12] K. McKay and D. Cooper. Guidelines for the selection, configuration, and use of transport layer security (TLS) implementations, section 3.3.2.1. *Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, [online]*, aug 2019. doi:10.6028/NIST.SP.800-52r2.

[13] CVE-2014-0160. Available from MITRE, CVE-ID CVE-2014-0160, dec 2013. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160. Last accessed: 29.07.2021.

[14] S. M. Nolan and Inc Vidatronic. Power management for internet of things (IoT) system on a chip (SoC) development. URL https://www.design-reuse.com/articles/42705/power-management-for-iot-soc-development.html. Last accessed: 29.07.2021.

[15] M. Abramovici, C. Stroud, and M. Emmert. Using embedded FPGAs for SoC yield improvement. *Proceedings of the 39th Conference on Design Automation - DAC '02*, 2002. doi:10.1145/513918.514099.

[16] SmartFusion2 SoC FPGA. URL https://www.microsemi.com/product-directory/soc-fpgas/1692-smartfusion2#overview. Last accessed: 29.07.2021.

[17] Eetimes: Microsemi announces smartfusion2 soc fpga, aug 2012. URL https://www.eetimes.com/microsemi-announces-smartfusion2-soc-fpga/. Last accessed: 29.07.2021.

[18] TLS handshake hardware accelerator. URL https://www.xilinx.com/products/intellectual-property/1-x0s0op.html. Last accessed: 29.07.2021.

[19] Intel Stratix 10 SX SoC FPGAs. URL https://www.intel.com/content/www/us/en/products/programmable/soc/stratix-10.html. Last accessed: 29.07.2021.

[20] S. Gueron. Intel advanced encryption standard (AES) instruction set white paper, 2010. Intel. Last accessed: 29.07.2021.

[21] LibreSSL. URL https://www.libressl.org/. Last accessed: 29.07.2021.

[22] WolfSSL. URL https://www.wolfssl.com/products/wolfssl/. Last accessed: 29.07.2021.

[23] Mbed TLS modules, . URL https://tls.mbed.org. Last accessed: 29.07.2021.

[24] M. Khalil-Hani, V. P. Nambiar, and M. N. Marsono. Hardware acceleration of OpenSSL cryptographic functions for high-performance internet security. *2010 International Conference on Intelligent Systems, Modelling and Simulation*, 2010. doi:10.1109/isms.2010.89.

[25] C. Coarfa, P. Druschel, and D. S. Wallach. Performance analysis of TLS web servers. *ACM Transactions on Computer Systems*, 24(1):39–69, feb 2006. doi:10.1145/1124153.1124155.

[26] S. Muller, D. Bermbach, S. Tai, and F. Pallas. Benchmarking the performance impact of transport layer security in cloud database systems. *2014 IEEE International Conference on Cloud Engineering*, 2014. doi:10.1109/ic2e.2014.48.

[27] Mbed TLS benchmarking application, . URL https://github.com/ARMmbed/mbedtls/blob/master/programs/test/benchmark.c. Last accessed: 29.07.2021.

[28] SecureMark-TLS. URL https://www.eembc.org/securemark/. Last accessed: 29.07.2021.

[29] Mbed TLS - core features, . URL https://tls.mbed.org/core-features. Last accessed: 29.07.2021.

[30] Mbed TLS - SSL/TLS module design, . URL https://tls.mbed.org/module-level-design-ssl-tls. Last accessed: 29.07.2021.

[31] H. Tschofenig and P. Eronen. Pre-shared key ciphersuites for transport layer security (TLS). IETF RFC 4279, dec 2005.

[32] B. Moeller, N. Bolyard, V. Gupta, S. Blake-Wilson, and C. Hawk. Elliptic curve cryptography (ECC) cipher suites for transport layer security (TLS). IETF RFC 4492, may 2006.

[33] I. Hajjeh and M. Badra. ECDHE_PSK cipher suites for transport layer security (TLS). IETF RFC 5489, mar 2009.

[34] Mbed TLS - debugging tls sessions, . URL https://tls.mbed.org/kb/development/debugging-tls. Last accessed: 29.07.2021.

[35] E. Barker. Recommendation for key management: Part 1 - general. *Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, [online]*, may 2020. doi:10.6028/NIST.SP.800-57pt1r5.

[36] Mbed TLS - API documentation: X.509 module, mbedtls_x509_crt_verify function, . URL https://tls.mbed.org/api/group__x509__module.html#ga98ed4504e4f832b735a230acf54fcde3. Last accessed: 29.07.2021.

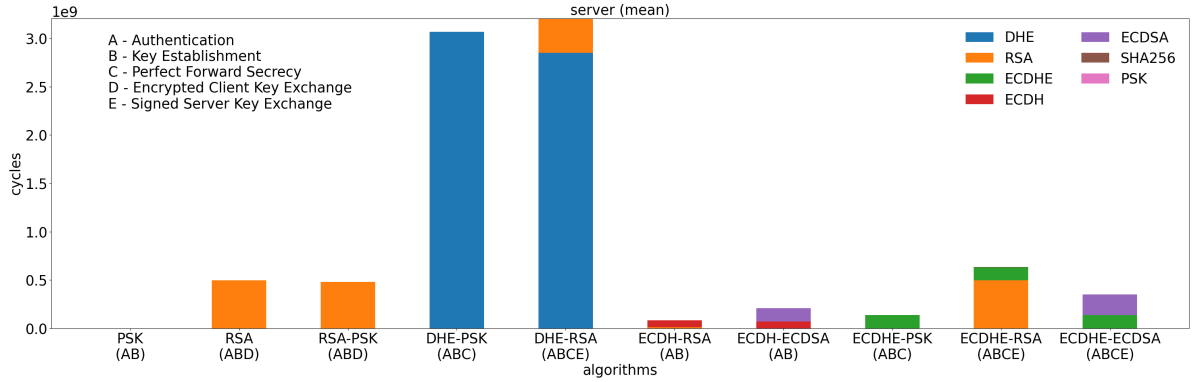
[37] SEED labs - lab environment setup. URL https://seedsecuritylabs.org/lab_env.html. Last accessed: 29.07.2021.

[38] Intel(R) Core(TM) i7-4720HQ Processor - Specifications. URL https://ark.intel.com/content/www/us/en/ark/products/78934/intel-core-i74720hq-processor-6m-cache-up-to-3-60-ghz.html. Last accessed: 29.07.2021.

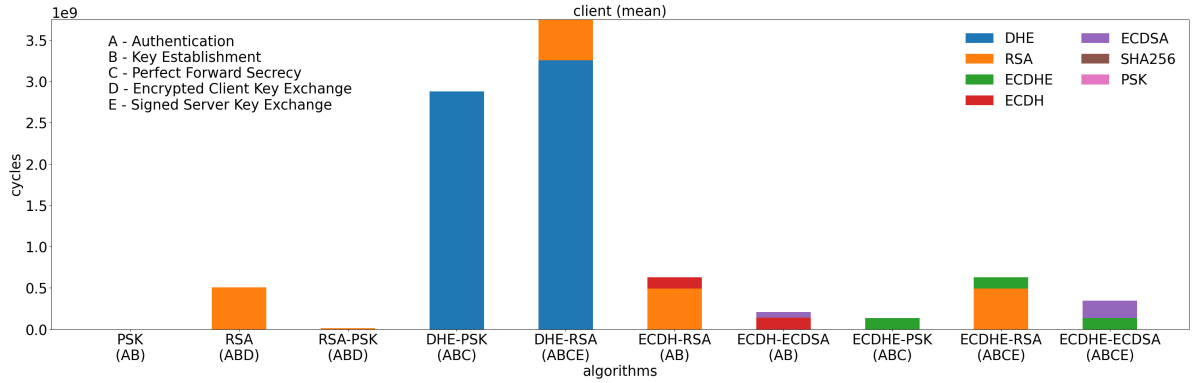
[39] Crypto Algorithms. crypto-algorithms, sha256.c. URL https://github.com/B-Con/crypto-algorithms/blob/master/sha256.c. Last accessed: 29.07.2021.

Appendix A

Handshake Protocol Performance

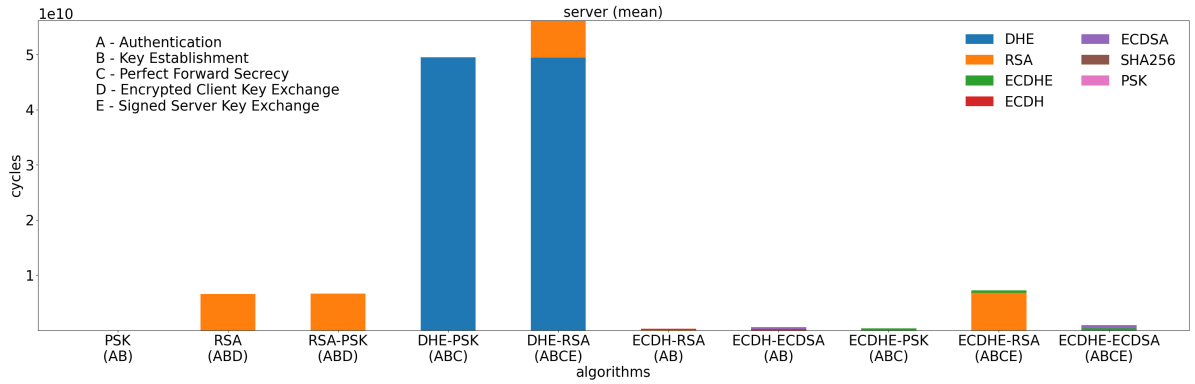


(a) Server-side

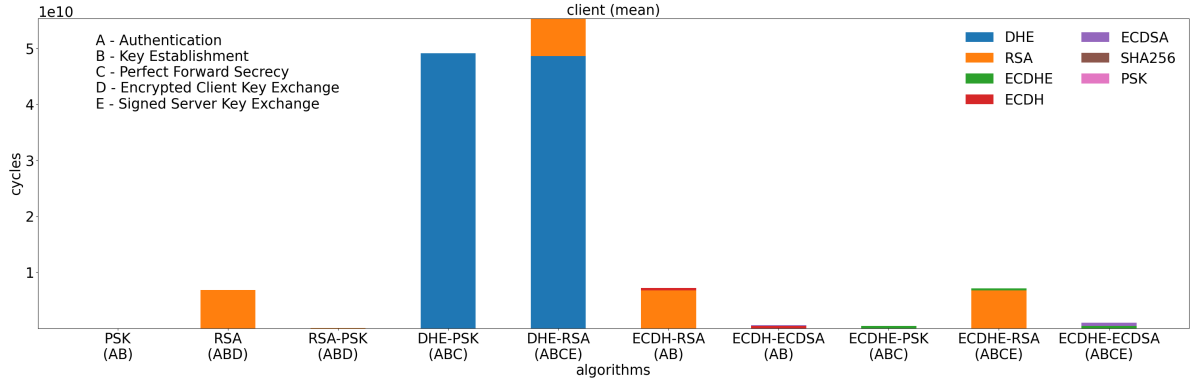


(b) Client-side

Figure A.1: Performance of each key exchange algorithm for security strength of 128 bits, in number of CPU cycles.

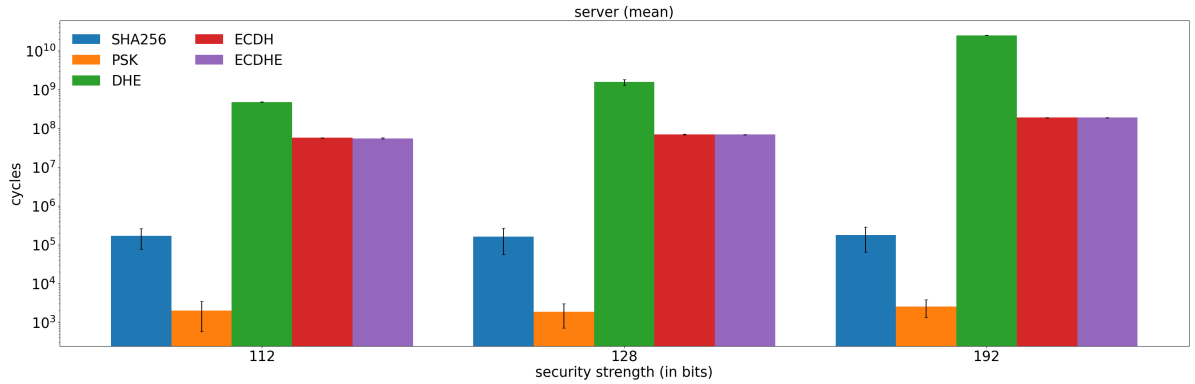


(a) Server-side

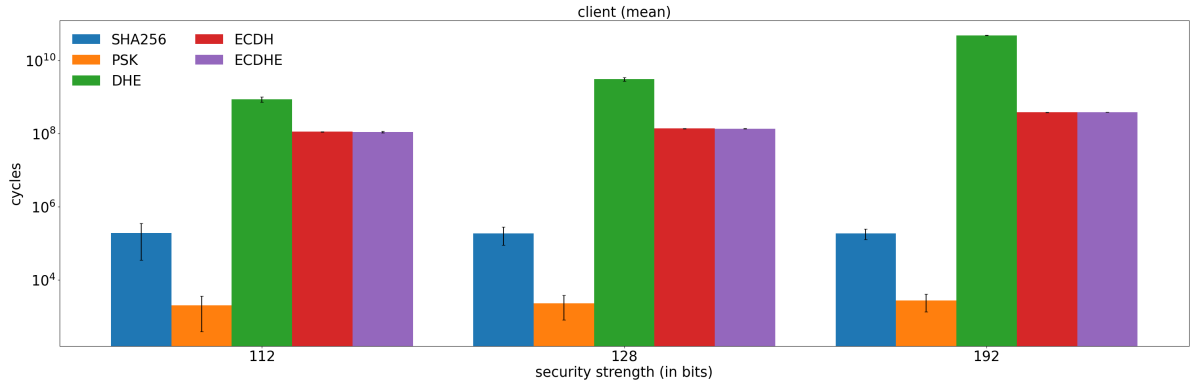


(b) Client-side

Figure A.2: Performance of each key exchange algorithm for security strength of 192 bits, in number of CPU cycles.

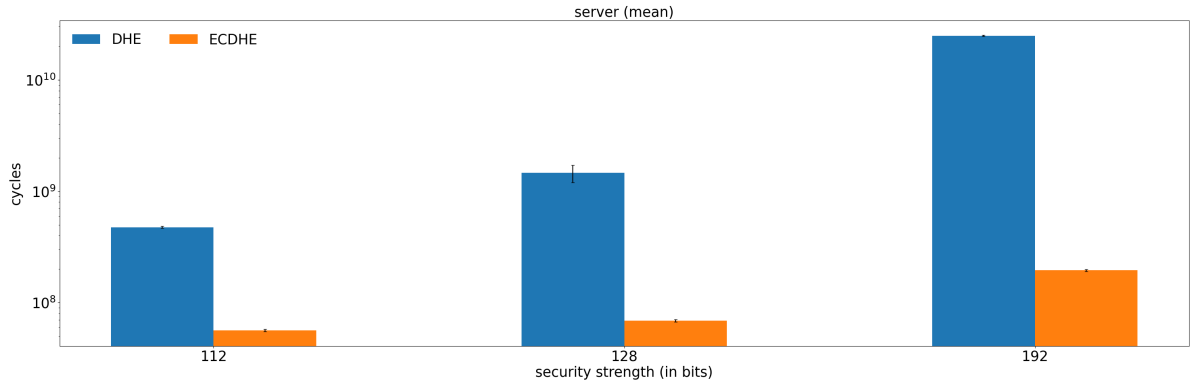


(a) Server-side

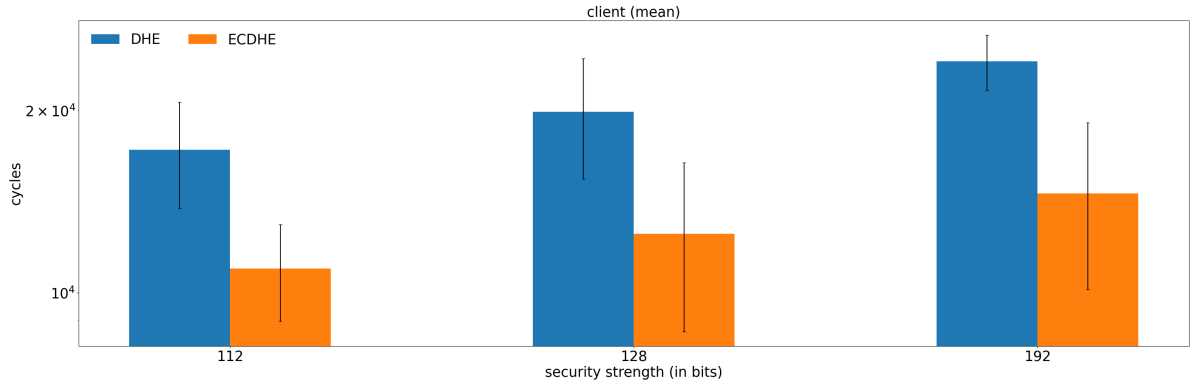


(b) Client-side

Figure A.3: Performance of each algorithm that provides the key establishment security service for all security strengths, in number of CPU cycles.



(a) Server-side

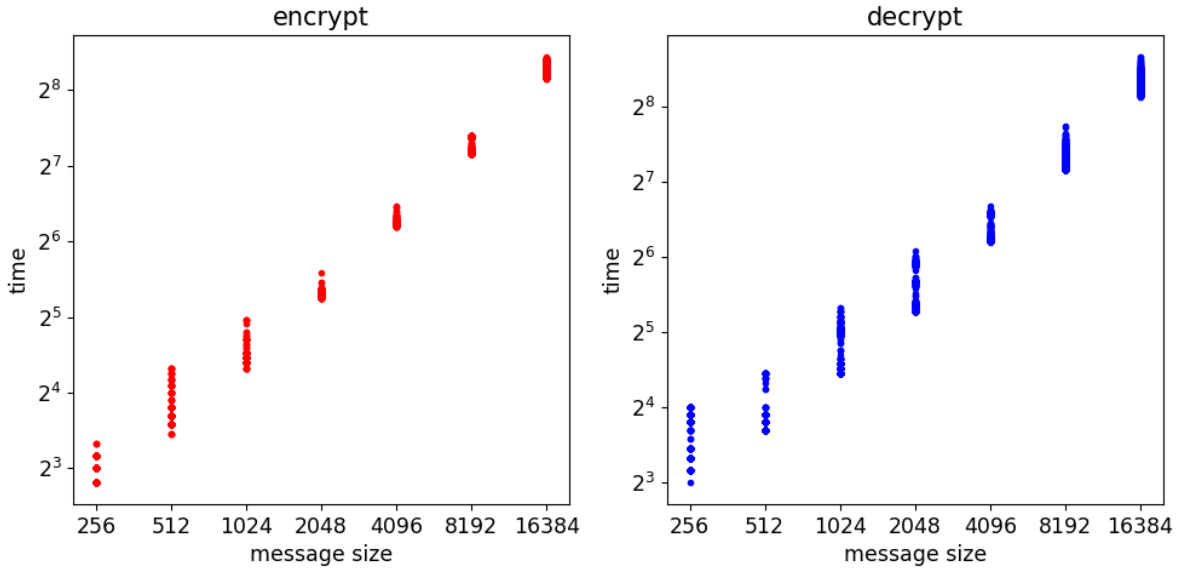


(b) Client-side

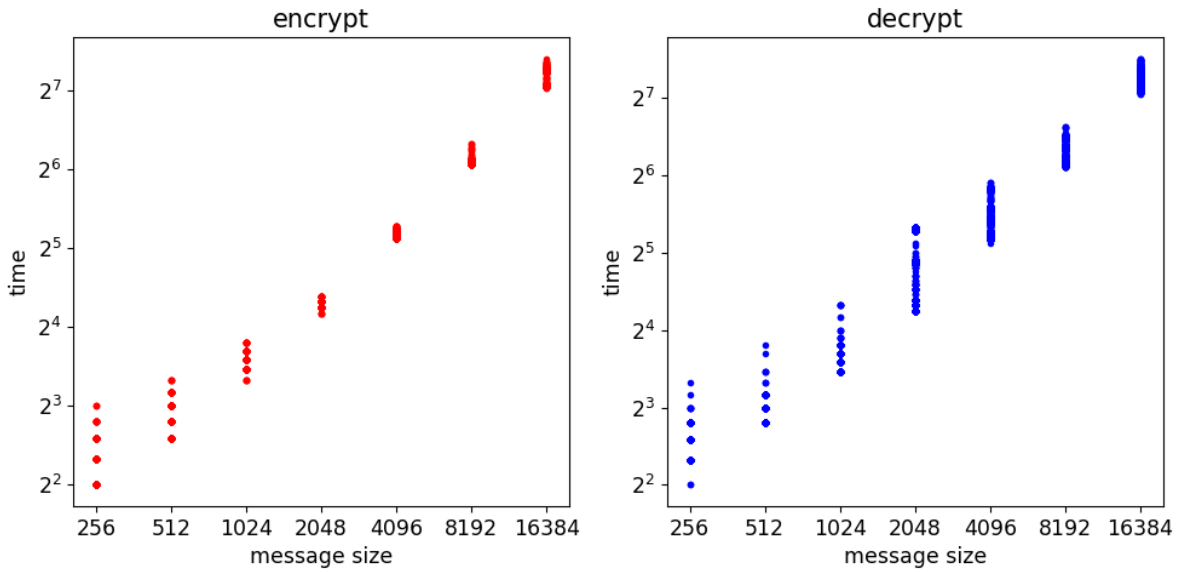
Figure A.4: Performance of each algorithm that provides the perfect forward secrecy security service for all security strengths, in number of CPU cycles.

Appendix B

Software Implementations Comparison

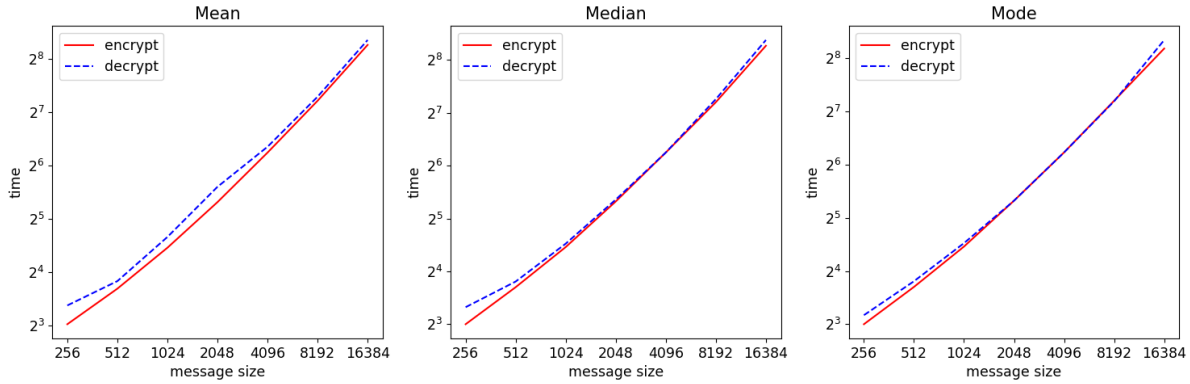


(a) Native AES Implementation

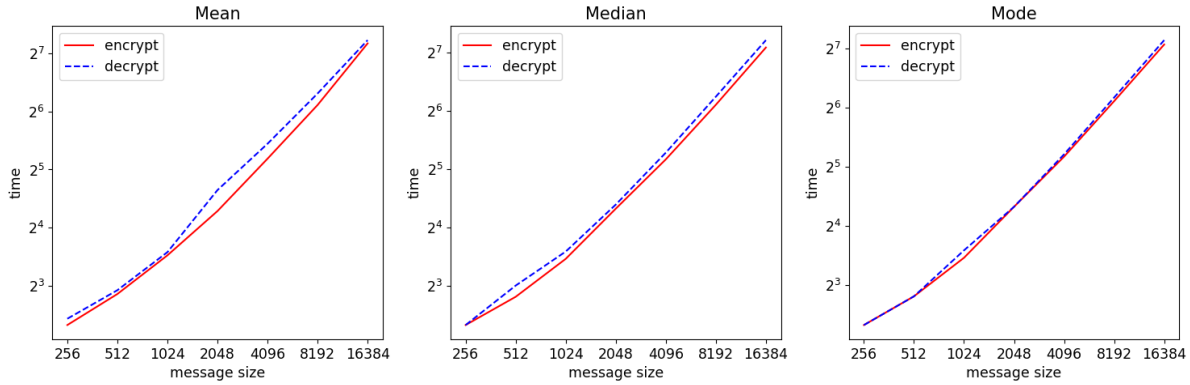


(b) Alternate AES-NI Implementation

Figure B.1: Data distribution of the AES operations for all message sizes, in microseconds.

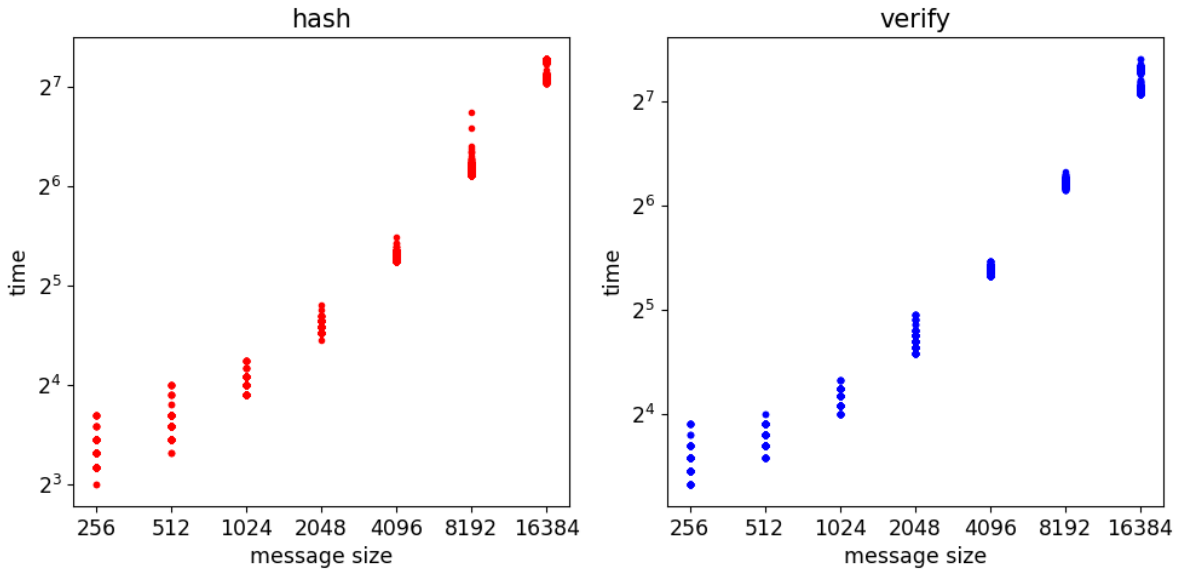


(a) Native AES Implementation

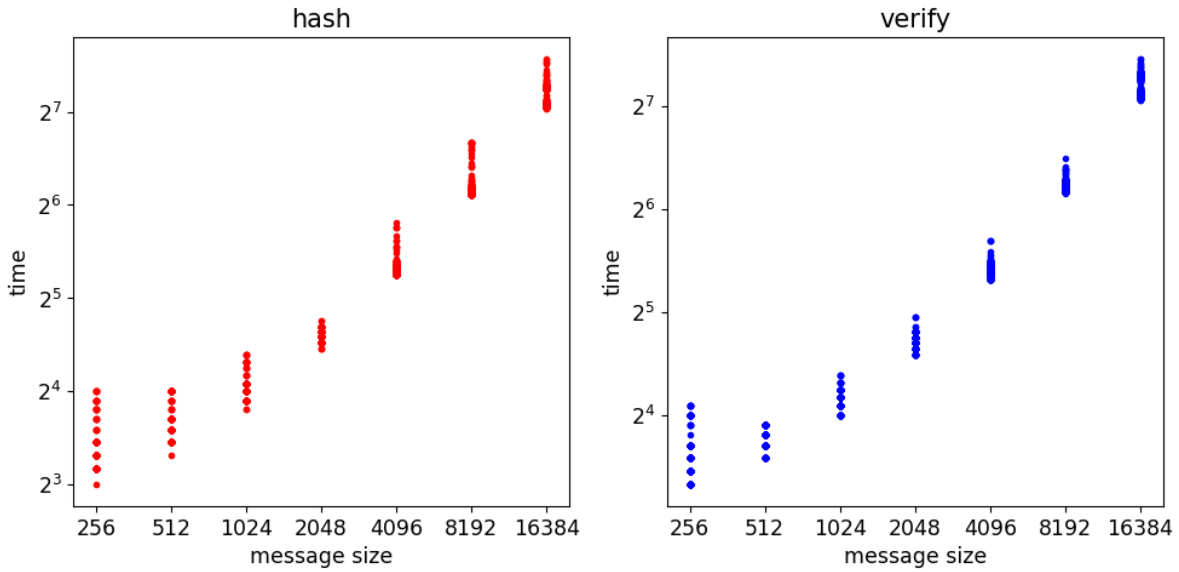


(b) Alternate AES-NI Implementation

Figure B.2: Mean, median and mode of the AES operations for all message sizes, in microseconds.

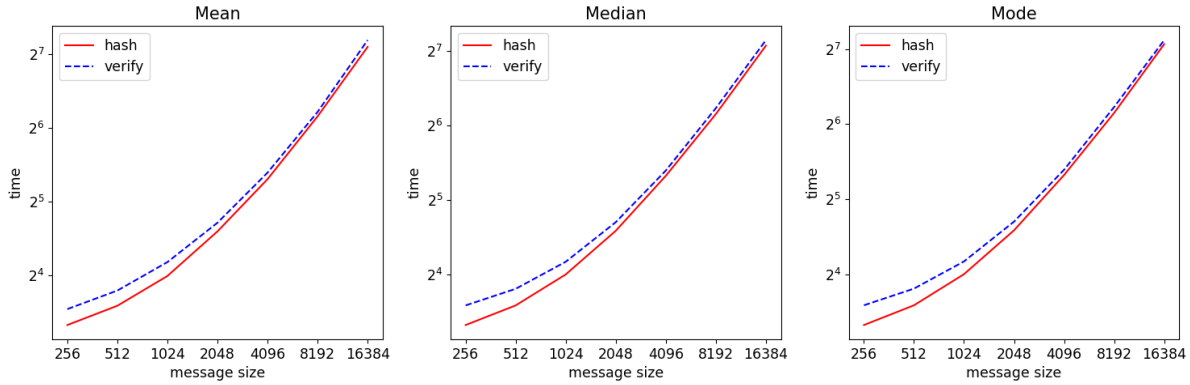


(a) Native SHA-2 Implementation

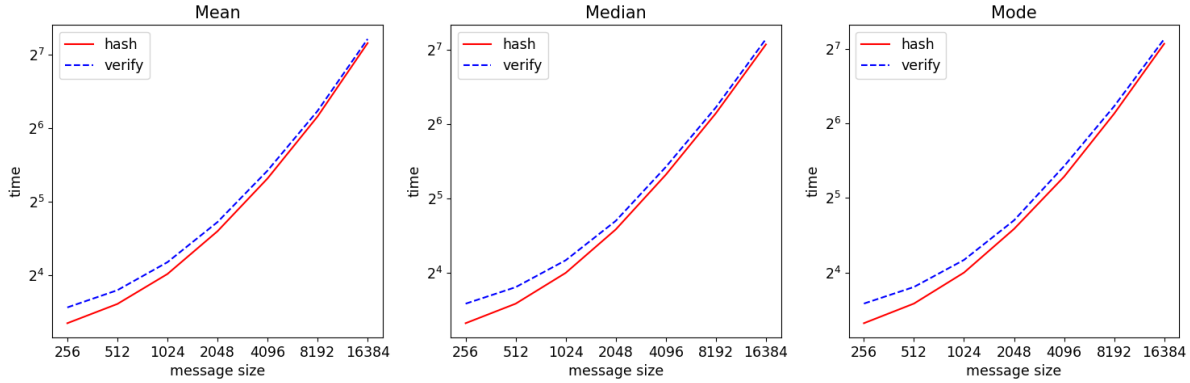


(b) Alternate SHA-2 Implementation

Figure B.3: Data distribution of the SHA-2 operations for all message sizes, in microseconds.

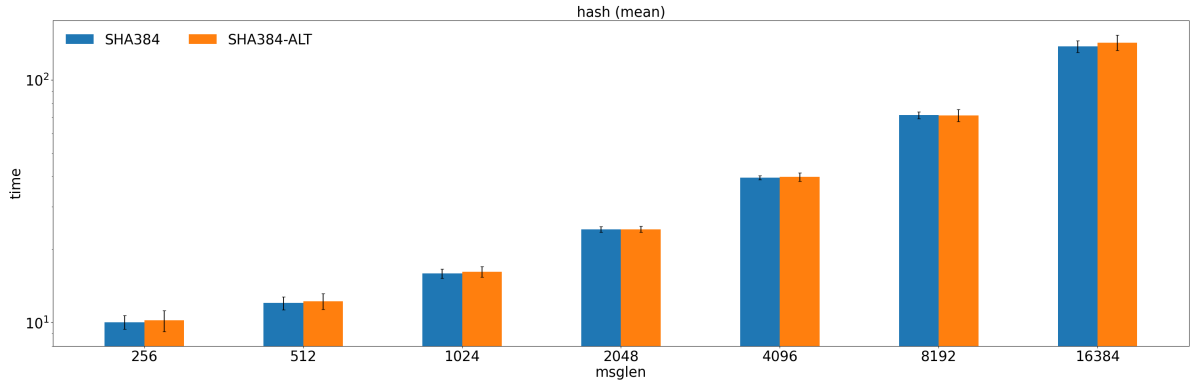


(a) Native SHA-2 Implementation

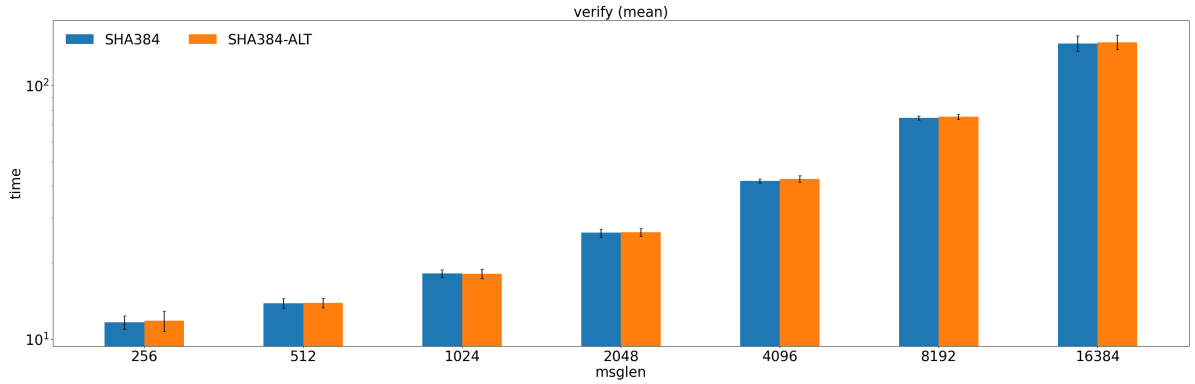


(b) Alternate SHA-2 Implementation

Figure B.4: Mean, median and mode of the SHA-2 operations for all message sizes, in microseconds.



(a) Hash Operation



(b) Verify Operation

Figure B.5: Comparison of the performance of both SHA-2 implementations for all message sizes, in microseconds.

Appendix C

Shell Tool User Guide

C.1 Data Acquisition Tools

```
services_analyser.py [-w <filter_weight>] [-H] [-a] [-k] [-p]
                     <path_to_data>

services_calculator.py [-w <filter_weight>] [-c] [-i] [-a] [-k] [-p]
                       <path_to_data>

services_comparator.py [-w <filter_weight>] [-c] [-i] [-a] [-k] [-p]
                       <path_to_data> <services_list>

services_profiler.py [-t <compilation_target>] [-w <filter_weight>]
                     [-s <initial_lvl>,<final_lvl>]
                     [-m <initial_size>,<final_size>] [-n <n_tests>]
                     [-d <data_directory>] [-H] [-c] [-i] [-a] [-k]
                     [-p] <services_list>

positional arguments:
  path_to_data        Relative path from the ./docs directory where the
                      data is stored
  services_list       File with list of security services and
                      algorithms that provide them. Example found in
                      examples/ke_servs.txt
```

```
optional arguments:
  -h, --help            show help message and exit
  -t <compilation_target>, --target=<compilation_target>
                        Path to endpoint implementation relative to
                        l-tls/directory. Default is tls_algs
  -w <filter_weight>, --weight=<filter_weight>
                        Weight of the z-score filter parameter. The
                        default is 2. filter_weight=0 means no data is
                        filtered
  -s <initial_lvl>,<final_lvl>, --sec_lvl=<initial_lvl>,<final_lvl>
                        Range of security levels to be considered. From 0
                        to 4, where 0 is considered insecure and 4 is
                        maximum security
  -m <initial_size>,<final_size>,
  --message_size=<initial_size>,<final_size>
                        Range of message sizes to be considered, in
                        bytes. From 32 to 16384 (16KB)
  -n <n_tests>, --n_tests=<n_tests>
                        Number of iterations
  -d <data_directory>, --data_path=<data_directory>
                        Name of the directory where the data will be
                        stored and used. Root directory is docs/
  -H, --handshake       Analyse overall handshake performance
  -c, --conf            Analyse performance of the confidentiality
                        security service
  -i, --int             Analyse performance of the integrity security
                        service
  -a, --auth            Analyse performance of the authentication
                        security service
  -k, --ke              Analyse performance of the key establishment
                        security service
  -p, --pfs             Analyse performance of the perfect forward
                        secrecy security service
```

C.2 Data Analysis Tools

```
algs_comparator.py [-w <filter_weight>] [-c] [-m] [-k] <path_to_data>
                   <algorithm_list>

algs_plotter.py [-w <filter_weight>] [-c] [-m] [-k] <path_to_data>

algs_profiler.py [-t <compilation_target>] [-w <filter_weight>]
                 [-s <initial_lvl>,<final_lvl>]
                 [-i <initial_size>,<final_size>] [-n <n_tests>]
                 [-d <data_directory>] [-c] [-m] [-k]
                 <algorithm_list>

positional arguments:
  path_to_data       Relative path from the ./docs directory where the
                     data is stored
  algorithm_list     File with list of algorithm types and algorithms
                     that belong in them. Example found in
                     examples/ke_algs.txt

optional arguments:
  -h, --help         show help message and exit
  -t <compilation_target>, --target=<compilation_target>
                     Path to endpoint implementation relative to
                     l-tls/ directory. Default is tls_algs
  -w <filter_weight>, --weight=<filter_weight>
                     Weight of the z-score filter parameter. The
                     default is 2. filter_weight=0 means no data is
                     filtered
  -c, --cipher       Analyse performance of cipher algorithms
  -m, --md           Analyse performance of message digest algorithms
  -k, --ke           Analyse performance of key exchange algorithms
  -s <initial_lvl>,<final_lvl>, --sec_lvl=<initial_lvl>,<final_lvl>
                     Range of security levels to be considered. From
                     to 4, where 0 is considered insecure and 4 is
```

maximum security

-m <initial_size>,<final_size>,

--message_size=<initial_size>,<final_size>

Range of message sizes to be considered, in bytes. From 32 to 16384 (16KB)

-n <n_tests>, --n_tests=<n_tests>

Number of iterations

-d <data_directory>, --data_path=<data_directory>

Name of the directory where the data will be stored and used. Root directory is docs/