Title of the MsC Thesis

Name of author
author.name@ist.utl.pt

Instituto Superior Técnico, Lisboa, Portugal

December 2011

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 system that will allow its users to select proper TLS configurations, enabling their devices to execute TLS and optimize it. This system 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 system will allow its users to add cost metrics and use various tools to perform the measurements. This system will implement basic metrics, such as execution time and the number of CPU cycles. This system will allow the use of alternate algorithms implementations to further optimize the TLS protocol. In the end, the capabilities of the system 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, Security Services

1. Introduction

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. They can also be embedded in various environments, such as mobile devices, industrial equipment, environmental sensors, or medical devices. IoT devices also play an important role in the concept and execution of smart home technologies.

Since IoT devices are connected to a network, they are susceptible to various attacks. 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 communication. 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 between having weaker security or having worse performance. 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 configuring the TLS session.

With this idea in mind, this work will create a system that can provide the desired trade-off between security and performance and enable the use of safer and/or more efficient TLS sessions, by se-

lecting the best available TLS configurations. The system 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.

This system will allow its users to get a detailed analysis of the TLS protocol. The analysis consists of a breakdown of the performance of the algorithms used during the session, as well as the security services provided in it. The system provides a dynamic performance analysis as it implements an interface that allows new metrics to be easily integrated and also allows choosing which metrics will be used.

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.

The TLS protocol is comprised of the handshake protocol and the record protocol. Figure 1 shows the establishment of a TLS session.

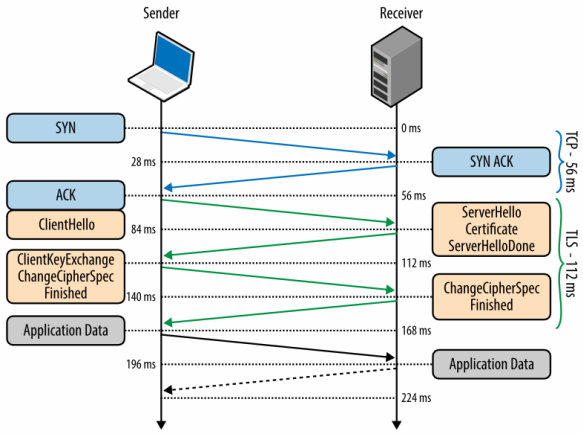


Figure 1: TLS protocol, with the TCP handshake in blue, the handshake protocol in orange and the record protocol in gray

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 agree on a session key. In the record protocol, the parties will start securely exchanging messages, using the settings defined in the handshake.

This work demonstrates the capabilities of the developed system 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.

2. Background

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 [2]. To better group these purposes, cryptography can be divided into security services. 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.

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.

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.

2.2. TLS Protocol

The TLS protocol is composed of the handshake protocol and the record protocol.

The handshake protocol is responsible for making the entities agree on the specifications that will be used to protect the communication [6]. These specifications include the algorithms that will be used.

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: a key exchange algorithm, a cipher algorithm, and a MAC algorithm.

The key exchange algorithm defines the session keys that will be used for cryptographic operations, while the cipher and MAC algorithms define how the exchanged data will be protected and validated. A ciphersuite can also include a signature and an authentication algorithm. Each cipersuite has a unique name that indicates its algorithmic contents.

The record protocol starts as soon as the handshake protocol ends. Only now do the client and

server start exchanging application data between themselves [6].

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. The records have a content type field that is used to differentiate its purpose and structure, although all record types use a similar structure [6].

By default, the records use a MAC-then-encrypt technique to provide confidentiality and integrity and the size of their content cannot be bigger than 16KB.

2.3. Computing Platforms

A computing platform is an environment in which a piece of software is executed. Computing platforms are important tools for software development since they can either constrain or assist in the performance of the software.

Intel implemented the Advanced Encryption Standard New Instructions (AES-NI) instruction set. AES-NI improves the speed and security of applications that use the AES algorithm to perform encryption or decryption operations.

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 [3]. The new instructions are AESENC, AESENCLAST, AESDEC, AESDECLAST, AESKEYGENASSIST and AESIMC.

These new instructions combine various steps of the algorithm in a single instruction. This not only improves the performance of the algorithm, but it also prevents recently discovered side-channel attacks on the algorithm.

2.4. Mbed TLS

Mbed TLS 2.16.5 supports TLS 1.0 to 1.2, DTLS 1.0 and 1.2, and SSL 3.0. It 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 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.

By using vtables, it is possible to implement the functions necessary to perform each algorithm separately and save the pointers to each function in a wrapper structure. That structure represents the implemented algorithm and is then added to a list of wrappers of the same algorithm type.

3. Proposed Solution

3.1. System Architecture

This system will mainly consider the configurations available to TLS 1.2. In the TLS context, the configuration refers to the ciphersuite that will be used during a session.

The ciphersuite specifies which security services will be guaranteed in the session since the services are achieved through the use of algorithms. Table 1 shows all security services and the respective algorithms, implemented in Mbed TLS, that provide them. The table does not include the use of AEAD modes for symmetric cipher algorithms.

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

The system is composed of two major modules, the data acquisition module and the data analysis module. After generating the data, the system will produce plots and statistics from it. The analysis of the data can be done using two approaches: focusing on individual algorithms or focusing on security services.

3.2. Data Acquisition Module

This module is composed of three components: the measurement component, the TLS component and the communication 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.

This component must allow the users to use multiple measurement tools at the same time, to measure all the enabled metrics and to allow a measurement tool to measure different metrics, if possible.

In order to meet all of these requirements, it was decided that this component needed to use a structure similar to the one present in Mbed TLS.

All the metric modules follow a specific structure, consisting of a data structure where two measured values will be stored and all the relevant functions needed to enable the measuring process.

The main module of this component contains all the functions needed to perform the measurements and save the acquired values in a file. It also contains a list of generic pointers that can be converted

into the metrics data structure.

It is necessary to perform two measurements to get the values of the metrics. One before a function call and another after the call. The metric value is then calculated by subtracting both values. The calculated values are all saved in CSV files.

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.

The TLS component is the one responsible for implementing the TLS protocol as well as all the cryptographic algorithms that it uses, through the use of the Mbed TLS library.

After analysing the workflow of the library, the SSL/TLS communication module was instrumented with measurement functions from the measurement component. 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.

A premaster secret is formed during the handshake and each key exchange algorithm uses different cryptographic materials to form it. The messages exchanged by the communication peers also vary depending on the chosen key exchange algorithm [6, 7, 5, 4].

Cipher and hashing algorithms are only evaluated during the record protocol. The SHA-2 algorithms are also evaluated in the handshake protocol, as they are used to generate the master secret and to derive the keys that will be used in the record protocol.

The public key module implements two types of algorithms: signing algorithms and key exchange algorithms. The former makes of use wrapper structures, while the latter does not follow a specific structure.

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 turn, the key exchange algorithms and the SHA-2 function calls were found by analysing the stack of functions calls.

This work also aims to enable the use of hardware accelerators to improve the execution of the TLS protocol. The method chosen for this task was utilizing the alternative implementation mechanism of the Mbed TLS library.

The communication component is the one responsible for implementing a server-client architecture and for creating all the keys and certificates necessary to perform the protocol.

The server-client architecture was implemented using the TLS component and makes use of a configuration file to select which features will be used. The communication endpoints enable the handshake and record protocols to be executed multiple times.

The user needs to send some parameters to these programs for them to work. 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.

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 used.

Table 2 shows the security levels considered in this work, along with their security strength, measured in bits[1], and corresponding key sizes for all the algorithms used in the handshake protocol. The current minimum required security level is 1.

		Key size (in bits)		
Security Level	Security Strength (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 2: Security levels with their corresponding security strength and key sizes for all algorithms used during the handshake.

These endpoints use self-signed certificates as this work does not focus on analysing the impacts of certificate validation.

3.3. Data Analysis Module

The data analysis can be done using two methodologies. By grouping the data into the security services provided by each algorithm or ciphersuite, or by grouping the data into the algorithm type. Each algorithm can provide one or more security services.

With this in mind, each individual function was assigned to a single security service. All functions in each Mbed TLS algorithm module were analysed, regarding their purpose in implementing the algorithm and assigned to a single security service. Table 1 shows all the security services that each algorithm provides. Table 3 shows all the algorithms and their respective algorithm type.

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

Some tools that make use of these grouping

methodologies were developed. These tools will generate all the relevant statistics and plots relative to a ciphersuite. Tools that allow the automation of the data acquisition process were also developed.

All tools use a basic data grouping procedure, depending on the grouping methodology they use. The idea of this procedure is to group the data by security service or algorithm type, as well as 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 type is "key exchange". The operation is the algorithm operation if the service is provided during the record protocol, i.e "confidentiality" and "integrity", or the algorithm type is "cipher" or "MAC".

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.

A graphical user interface (GUI) was also developed to make the use of the tools more intuitive to the users.

4. Results

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

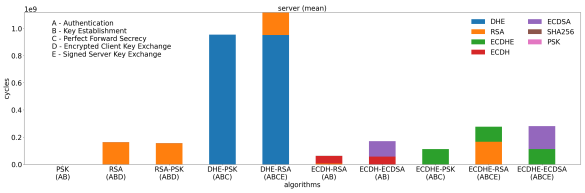
For the first scenario, it was decided to use the developed system to analyse the performance of each security service provided during the handshake protocol. The services that can be provided during the handshake are authentication, key establishment and perfect forward secrecy.

For this scenario, each key exchange algorithm that can be used in a ciphersuite will be tested. Table 3 contains all the key exchange algorithms that were tested. The server and client authenticated themselves mutually, when possible, and used the same symmetric encryption and MAC algorithms.

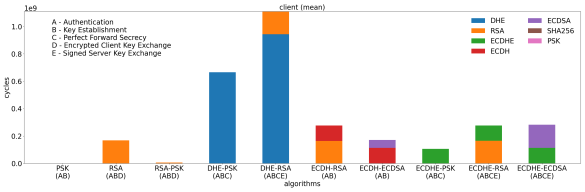
Additionally, each key exchange algorithm was tested using keys that provide security levels from 1 to 3. Table 2 contains the respective key sizes for each algorithm. The services profiler tool was used to acquire the data. This implies that no data discrepancies were created by compiler optimizations as the code was only compiled once.

The metric used was the number of cycles per algorithm execution, which was obtained using the PAPI library. The tests were executed using the VirtualBox software to virtualize a pre-built Ubuntu (32-bits) virtual machine image provided by SEED Labs. The host device uses an Intel(R) Core(TM) i7-4720HQ processor and the virtual environment made use of all 4 CPU cores.

After generating the data, the services analyser and comparator tools were used to generate the plots. Figures 2, 3 and 4 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 bar has an extra label that indicates which security services are provided by that algorithm.

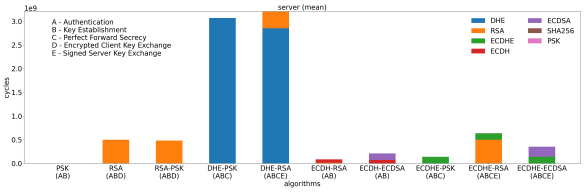


(a) Server-side

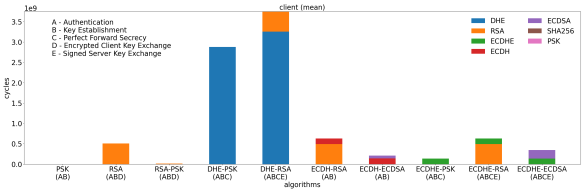


(b) Client-side

Figure 2: Performance of each key exchange algorithm for security strength of 112 bits, in number of CPU cycles.



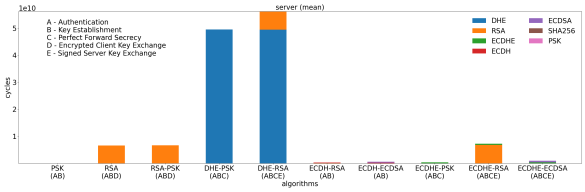
(a) Server-side



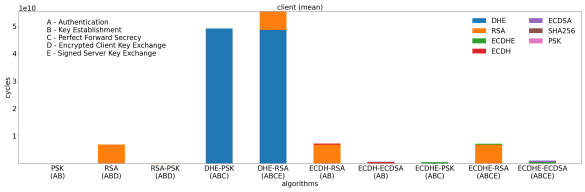
(b) Client-side

Figure 3: Performance of each key exchange algorithm for security strength of 128 bits, in number of CPU cycles.

As can be seen from all those plots, the most taxing key exchange algorithm is DHE-RSA followed by DHE-PSK. This is due to the use of the extremely large keys by DHE and, in the case of DHE-RSA, RSA. 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



(a) Server-side

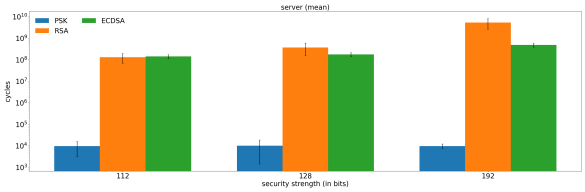


(b) Client-side

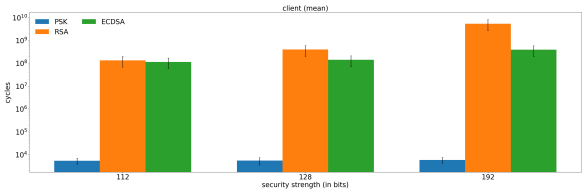
Figure 4: Performance of each key exchange algorithm for security strength of 192 bits, in number of CPU cycles.

of security.

Figures 5, 6 and 7 show the performance of each algorithm used to provide the authentication, key establishment and perfect forward secrecy services, respectively. The plots use a logarithmic scale and group the bars by security strength. The bars also contain an error bar corresponding to the standard deviation of the data sample.



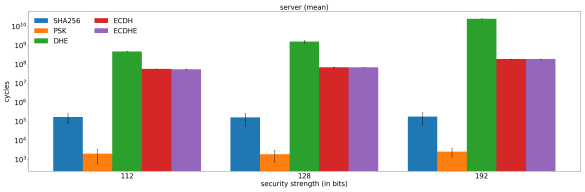
(a) Server-side



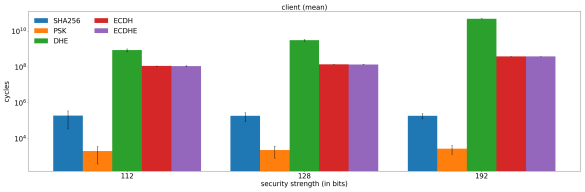
(b) Client-side

Figure 5: Performance of each algorithm that provides the authentication security service for all security strengths, in number of CPU cycles.

As can be seen in figures 5 and 6, PSK is the least taxing algorithm 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 7, the effect of the difference between DHE and ECDHE key sizes can be seen again.

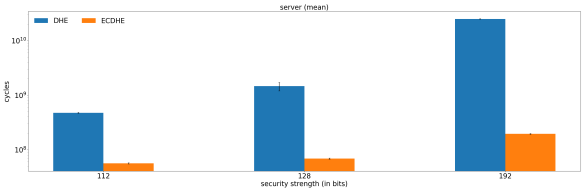


(a) Server-side

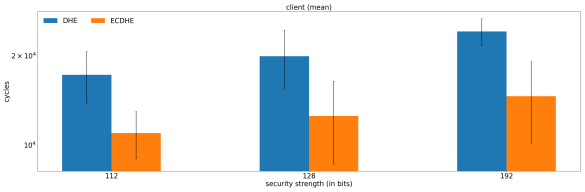


(b) Client-side

Figure 6: 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 7: Performance of each algorithm that provides the perfect forward secrecy security service for all security strengths, in number of CPU cycles.

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 2 and 3 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 7.

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 2, 3 and 4 and the reasonably sized error bar in Figure 5 exists due to the fact that ECDHE-ECDSA performs the signing of the ServerKeyExchange message, while ECDH-ECDSA does not.

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 server private key. 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.

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 2, 3 and 4. The analysis also explains the error bars in Figure 5.

The PSK and SHA256 have such little impact, relative to the other algorithms that they can not be seen in Figures 2, 3 and 4. In Figures 5 and 6 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, likely caused by the same reason as the DHE data inconsistencies.

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 is the ECDHE-ECDSA. This key exchange algorithm provides all of the mentioned security services 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 than RSA or DHE algorithms, since they can use much smaller keys to provide equivalent levels of security.

4.2. 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.

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. The second AES implementation was created using the example code found in a white paper [3] that makes use of the AES-NI instruction set created by Intel. The code from both algorithms was adapted to be used in the Mbed TLS library.

The adaptations were made using the alternate implementation method described in section "3.2". These implementations will be mentioned as alternate implementations for the duration of this section.

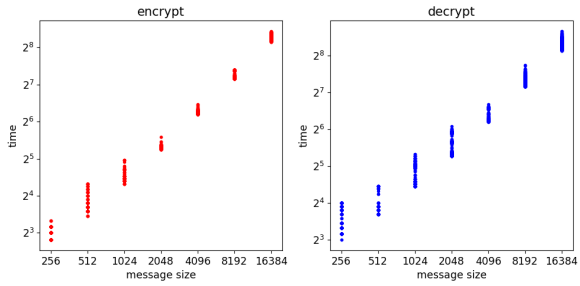
The algorithms profiler tool was used to acquire the data for this scenario. To generate the data, only the TLS-PSK-WITH-AES-256-CBC-SHA384 ciphersuite was tested. The tool needed to be used twice, once for the native implementations and another use for the alternate implementations.

To get a better understanding of the performance of both implementations, various message sizes were tested. The tested message sizes range from 256B to 16KB.

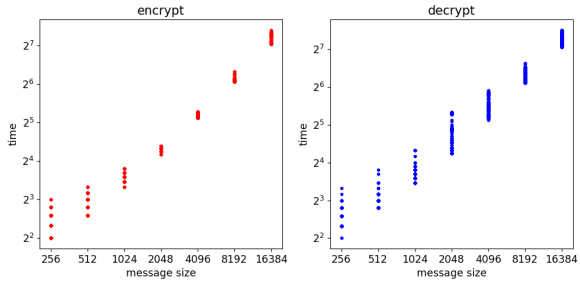
The metric used for this scenario was CPU time, in microseconds, using the time.h standard library from C. The tests were performed using the host device mentioned in the previous scenario. After getting the data, the algorithms plotter and comparator tools were used to generate plots.

Figures 8, 9 and 10 show the plots relative to the implementations of the AES algorithm, while Figures 11, 12 and 13 show the plots relative to the implementations of the SHA-2 algorithm. The data coloured in red represents the encrypt or hash operations, whereas the blue coloured data represents

the decrypt or verify operations. 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.

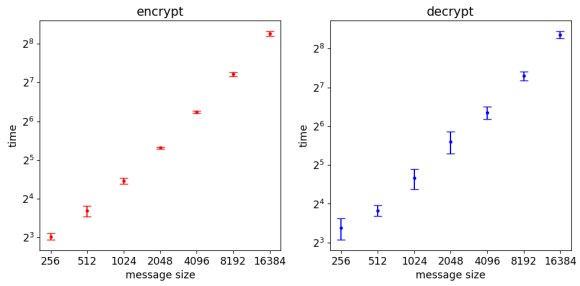


(a) Native Implementation

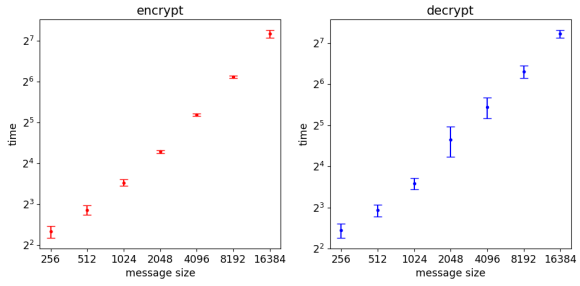


(b) Alternate Implementation

Figure 8: Data distribution of the AES operations for all message sizes, in microseconds.

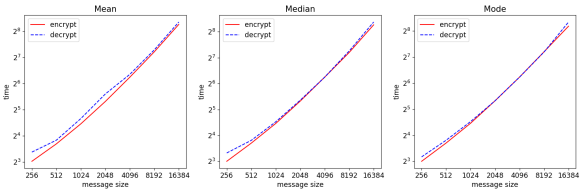


(a) Native Implementation

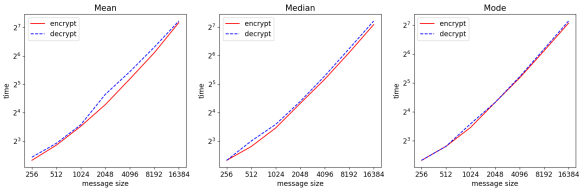


(b) Alternate Implementation

Figure 9: Mean and standard deviation of the AES operations for all message sizes, in microseconds.

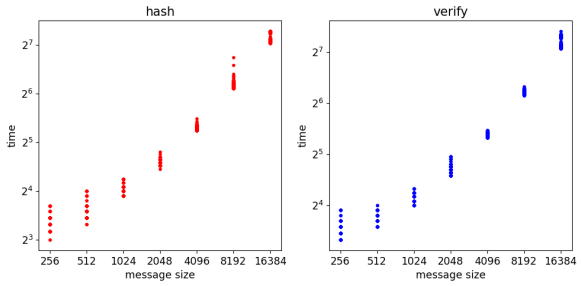


(a) Native Implementation

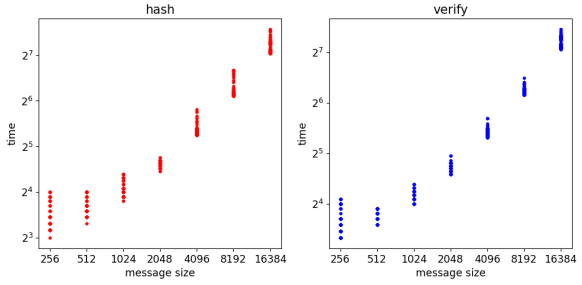


(b) Alternate Implementation

Figure 10: Mean, median and mode of the AES operations for all message sizes, in microseconds.



(a) Native Implementation

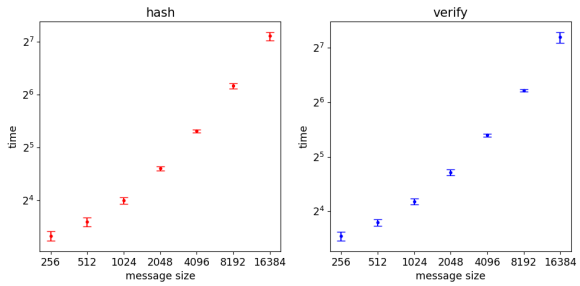


(b) Alternate Implementation

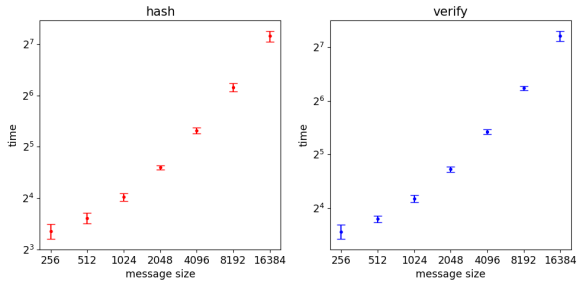
Figure 11: Data distribution of the SHA-2 operations for all message sizes, in microseconds.

As can be seen in Figures 10 and 13, the in operations are slightly slower than the out operations. For the decrypt operations only, this might be because the data set has high standard deviations and the data points are also more dispersed than the ones from the encrypt operations, as can be seen in Figures 8 and 9.

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

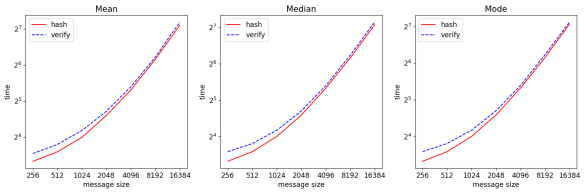


(a) Native Implementation

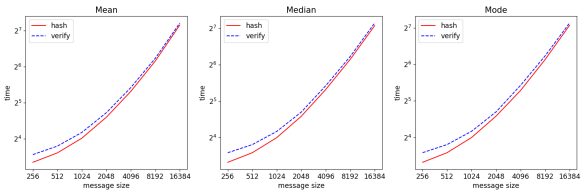


(b) Alternate Implementation

Figure 12: Mean and standard deviation of the SHA-2 operations for all message sizes, in microseconds.



(a) Native Implementation



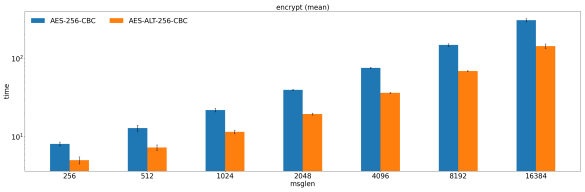
(b) Alternate Implementation

Figure 13: Mean, median and mode of the SHA-2 operations for all message sizes, in microseconds.

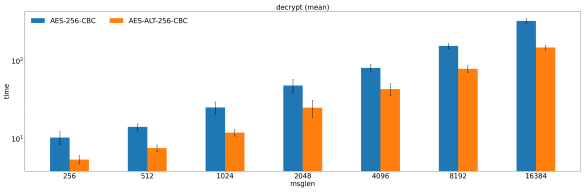
this behaviour is persistent no matter the number of that were run when acquiring the data. Apart from this particularity, all the generated plots show reasonable results.

Figures 14 and 15 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.

As can be seen in Figure 14 the alternate AES

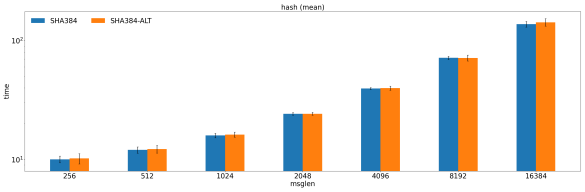


(a) Encryption Operation

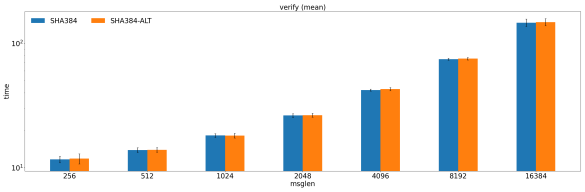


(b) Decryption Operation

Figure 14: Comparison of the performance of both AES implementations for all message sizes, in microseconds.



(a) Hash Operation



(b) Verify Operation

Figure 15: Comparison of the performance of both SHA-2 implementations for all message sizes, in microseconds.

implementation is a lot less taxing on the device, as it takes around half the time to perform the operations than the native implementation. This can also be seen when comparing the scale on of the plots in Figure 10. 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.

As can be seen in Figure 15, 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 on optimizing 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 or AMD processors.

5. Conclusions

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

The system 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 system 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 system, devices, particularly IoT ones, can secure their communications by properly configuring a TLS session. Other devices, that have access to more resources, can also use this system to further increase the performance and/or robustness of their TLS sessions.

This work also provides the most extensive analysis of the Mbed TLS 2.16.5 library, to date. The analysis includes a detailed explanation of the mechanisms used by the library, as well as its structure and how its modules are connected.

This work also demonstrates the capabilities of the developed system 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 different algorithm implementations, with the particularity of using the AES-NI instruction set developed by Intel.

6. Future Work

For future work, it would be interesting to extend the profiling capability of the system by including other relevant metrics. As it stands, the system 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 chose a good TLS configuration.

The most interesting metrics to include would be power consumption and memory usage. Both of these metrics are relevant because 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 are more relevant in TLS 1.3, they can still be used in version 1.2 and are even supported by Mbed TLS. 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 system 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 system 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.

Acknowledgements

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.

References

[1] 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.

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

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

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

[5] 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.

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

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