# TLS for Internet of Things

Illya Gerasymchuk
illya@iluxonchik.me

Instituto Superior Técnico, Lisboa, Portugal

May 2019

**Abstract**

Transport Layer Security (TLS) is one of the most used communication security protocols in the world. Its main goal is to provide a secure communication channel with the security services of confidentiality, integrity, authentication, and Perfect Forward Secrecy (PFS). Each security service can be implemented by one of the multiple available algorithms. TLS was not designed for the constrained environment and is too computationally demanding for many Internet Of Things (IoT) devices. However, it is a malleable protocol and individual security services can be enabled and disabled on a per-connection basis. Foregoing a security service or using a cheaper algorithm to implement it reduces the utilized computational resources. The security properties of a connection are defined by a TLS configuration. Some of those configurations can be used with the resource-constrained IoT devices. Existing work focuses on Datagram TLS (DTLS) and is either tied to a specific protocol or requires the usage of a third-party entity. For this reason, it cannot be easily integrated with existing deployments. In this work, we perform a thorough evaluation of the TLS protocol and its security services. We present a framework that can be used by software developers and security professionals to select the cheapest TLS configuration for their environment's needs and limitations. We evaluated the TLS implementation of the *mbedTLS* library using two cost metrics: the estimated number of CPU cycles, obtained with *valgrind*, and execution time, obtained with *PAPI*. In the end, we will show that the estimated values are close to the real ones.
**Keywords:** TLS, DTLS, SSL, IoT, Embedded Systems

## 1. Introduction

In recent years there has been a sharp increase in the number of IoT devices and this trend is expected to continue. The IoT is a network of interconnected devices, which exchange data with one another over the internet. In fact, it can be any object that has an assigned IP address and is provided with the ability to transfer data over a network. While there are many types of IoT devices, all of them are restricted: they have limited memory, processing power and available energy. This doest not mean, however, that such devices are only capable of running the least demanding algorithms. Various devices, with different hardware characteristics fall under the definition of IoT. While for some of them symmetric cryptography is the only viable option, others have resources that allow them to use public key cryptography. Examples of IoT devices include temperature sensors, smart light bulbs and physical activity trackers.

While inter-device communication has numerous benefits, it is important to ensure the security of that communication. For example, when you log in to your online banking account, you do not want others to be able to see your password, as this may lead to the compromise of your account. Having your account compromised means that a malicious entity might take a hold of your money. Despite of all of the benefits that the IoT technology brings, communication security is often an afterthought and is frequently ignored.

TLS is one of the most used protocols for communication security. It powers numerous technologies, such as Hypertext Transfer Protocol Secure (HTTPS). TLS offers the security services of authentication, confidentiality, privacy, integrity, replay protection and perfect forward secrecy. It is not a requirement to use all of those services for every TLS connection. The protocol is similar to a framework, in the sense that you can enable individual security services on a per-connection basis. Foregoing unnecessary services will lead to a smaller resource usage, which in turn leads to smaller execution time and power usage. This is especially important in the context of IoT, due to the constrained nature of the devices. For example, while confidentiality, integrity and authentication are important when the device communicates with an external ser-

vice, the first security property is not crucial when the device is downloading a firmware update. In the latter case, integrity and authentication would be enough.

While TLS was not designed for the constrained environment of IoT, it is a malleable protocol and can be configured to one's needs. For each security service that the protocol offers, there is an array of algorithms that can be used to implement it. If those algorithms are chosen properly, it is possible to use TLS in the context of IoT.

Existing work either focuses on Datagram TLS (DTLS) optimization and not all of it can be applied to TLS, or requires the changes to the core TLS protocol, such as by introducing third-party entities. Herein we want to further explore TLS optimization. There is clear a need for that, especially with Constrained Application Protocol (CoAP) over TCP and TLS standard[6] being currently developed. The aforementioned standard does not explore any TLS optimizations, and since any IoT device using it in the future would benefit from them, this is an important area to explore.

The objective of this work is to provide a means of assisting application developers who wish to include secure communications in their applications to make security/resource usage trade-offs, according to the environment's needs and limitations. We aim to provide a general overview of of the costs of the TLS protocol as a whole and of its individual parts. This is will allow to answer questions such as *How much will we save if we use algorithm X instead of Y for authentication?*. Thus, performing evaluations on specific IoT hardware or analyzing hardware-specific optimizations is outside of the scope of this paper.

In order to achieve our goals, a detailed cost evaluation of TLS is needed. With this information, the programer will be able to choose a configuration that meets his security requirements and device constraints. If the limitations of the device's hardware do not allow to meet the requirements, the programer may decide on an alternative configuration, possibly with a loss of some security services and a lower security level, or forgo using (D)TLS altogether. Thus, this work is targeted towards developers and InfoSec professionals who wish to add communication security to applications in the IoT environment.

In our work, we performed a thorough cost evaluation of the TLS 1.2 implementation in *mbedTLS 2.7.0*. *mbedTLS* is among the most popular TLS implementation libraries for embedded systems. We evaluated costs in terms of the estimated number of CPU cycles and time taken. The time values were read directly from the processor's registers. In our analysis we will show that the estimates do reflect real values, by comparing them to time measurements obtained directly from the CPU registers. We evaluated every single one of the 161 TLS configurations available in *mbedTLS 2.7.0*, at 4 different security levels.

A TLS connection consists of two main parts: first, the peers establish a secure communication channel in the Handshake phase, followed by the data exchange using that channel in the Record phase. We focused on the Handshake part of the protocol for two main reasons. First, it is the part with the most variability in terms of cost, due to the complex combinations of different possible algorithms. Second, it is part which has been the least studied by existing work. The Record phase mainly consists in the use symmetric encryption algorithms and hash functions. Their costs has already been thoroughly studied by existing work.

Although our focus was on the Handshake, we also profiled the costs of the symmetric encryption algorithms and hash functions. We measured how much data needs to be exchanged between the peers in order for the costs of the Record phase to equate the costs of the Handshake phase. We concluded that for the most commonly used configurations on the internet, that number is between 560KB and 1.62MB for the client, and between 830KB and 1.27MB for the server. Thus, considering that the device will perform a Handshake for each new connection, it only makes sense to focus on Handshake cost optimization if the amount of exchanged data is small.

In the process of the work on this dissertation, we have made several contributions to the TLS 1.3 specification, and were formally recognized as contributors[1]. The name of the author of this dissertation can be found in the document specifying TLS 1.3[9]. Although to the lesser extent, we have also contributed to DTLS 1.3 specification[2]. We have found a security vulnerability and a nonconformity to the standard in the TLS implementation of the *mbedTLS* library. We reported it and it has been assigned a Common Vulnerabilities and Exposures (CVE) with the id *CVE-2018-1000520*[3]. It is a vulnerability in the authentication part of the TLS protocol, where certificates signed with an incorrect algorithm were accepted in some cases. More specifically, *ECDH(E)-RSA* ciphersuites allowed Elliptic Curve Digital Signature Algorithm (ECDSA)-signed certificates, when only Rivest-Shamir-Adleman (RSA)-signed ones should have been. We also found a bug in *mbedTLS*'s test suite related to the use of deprecated *SHA-1*-signed certificates and submitted a code fix to it[4][5]. We have also developed an extensive set of tooling which can be used to further study the costs of TLS, by allowing to automate metric col-

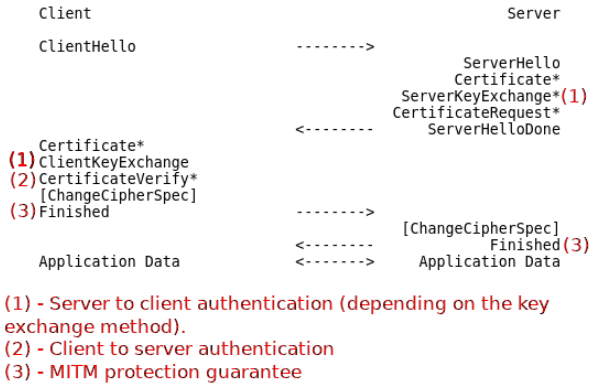


Figure 1: TLS 1.2 message flow for a full handshake

lection and analysis on different hardware and environments.

## 2. The TLS Protocol

TLS is a **client-server** protocol that runs on top a **connection-oriented and reliable transport protocol**, such as **TCP**. Its main goal is to provide **confidentiality** and **integrity** between the two communicating peers. Confidentiality implies that a third party will not be able to read the data, while integrity means that a third party will not be able to alter the data.

In the TCP/IP Protocol Stack, TLS is placed between the **Transport** and **Application** layers. It is designed to simplify the establishment and use of secure communications from the application developer's standpoint. The developer's task is reduced to creating a "secure" connection (*i.e.* socket), instead of a "normal" one.

A secure communication established using TLS has two phases. In the first phase, the communicating peers authenticate one to another and negotiate the parameters, such as the secret keys and the encryption algorithm. In the second phase, they exchange cryptographically protected data under the previously negotiated parameters. The first phase is done under the Handshake Protocol and the second under the Record Protocol. In order to achieve its goals, during the Handshake Protocol the client and the server exchange various messages. This message flow is depicted in Figure 1. * indicates situation-dependent messages that are not always sent.

TLS provides the following **security services**:

- **authentication** - both, **peer entity** and **data origin** (or **integrity**) authentication.

  **peer entity authentication** - a peer has a guarantee that it is talking to certain entity, for example, *www.google.com*. This is achieved thought the use of Asymmetrical Cryptography (AC), also known as Public Key Cryptography (PKC), (*e.g.* *RSA* and *DSA*) or **symmetric key cryptography**, using a Pre-Shared Key (PSK).

- **confidentiality** - the data transmitted between the communicating entities (the client and the server) is encrypted. Symmetric cryptography is used for data encryption (*e.g.*, *AES*).

- **integrity** (also called **data origin authentication**) - a peer can be sure that the data was not modified or forged, *i.e.*, there is a guarantee that the received data is coming from the expected entity. For example, a peer can be sure that the *index.html* file that was sent to when it connected to *www.google.com* did, in fact, come from *www.google.com* and it was not tampered with by an attacker (**data integrity**). This is achieved either through the use of a keyed Message Authentication Code (MAC) or an Authenticated Encryption With Associated Data (AEAD) cipher.

- **replay protection** (also known as **freshness**) - a peer can be sure that a message has not been replayed. This is achieved through the use of sequence numbers. Each TLS record has a different sequence number, which is incremented. If a non-AEAD cipher is used, the sequence number is a direct input of the MAC function. If an AEAD cipher is used, a nonce derived from the sequence number is used as input to that cipher.

- **perfect forward secrecy (PFS)** - the confidentiality of the past interactions is preserved even if the long-term secret is compromised.

It is not a requirement to use all of the security services every situation. In this sense, TLS is like a framework that allows to select which security services should be used for a communication session. As an example, certificate validation might be skipped, which means that the **authentication** guarantee is not provided.

The set of security services offered by a connection depend on the TLS configuration in use. A TLS configuration defines the key exchange method and the symmetric algorithm/hash function pair. The key exchange method defines the security services that will be used in the connection, as well as which algorithms will be used to offer that security service. In this text, we use the terms TLS configuration and *ciphersuite* interchangeably. For example, if the *TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384* ciphersuite is used, the connection will be established with the *ECDHE-ECDSA* key exchange, and the combination of *AES* in *CBC* mode with

256 bit keys and the *SHA-384* hash function will be used. *ECDHE-ECDSA* key exchange implies that the Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) algorithm will be used to provide PFS and ECDSA algorithm to provide authentication.

## 3. Results and Data Analysis

This section presents the results of the cost analysis of TLS and its security services. We will begin with an analysis of the estimated number of CPU cycles obtained with *vallgrind*/*callgrind*, after which we will present the time metrics obtained with Performance Apppicaiton Programming Interface (PAPI). A comparison of both set of results will show that the estimates reflect the real values.

### 3.1. Methodology

In our work, we used two cost metrics: the estimated number of CPU cycles and the time taken. The estimated number of CPU cycles was obtained with the combination of of *valgrind* and *callgrind*. In order to compute the time values, we used PAPI, which obtains the time values directly from the processor's registers. We used *kcachegrind*'s formula to compute the estimated number of CPU cycles ($CEst$): $CEst = Ir + 10 * Bm + 10 * L1m + 100 * L2m$, where $Ir$ is the number of instruction fetches, $Bm$ is the number of mis-predicted branches, and $L1$ and $L2$ are the total number of *L1* and *L2* cache misses, respectively.

With PAPI, in order to keep the metrics consistent and approximate the conditions to the ones of an IoT environment, we disabled Intel Turbo Boost and fixed the processor's speed to the lowest available frequency of $800Mhz$. We profiled the *virtual time* elapsed, which is the actual CPU time used in executing the process and does not include time slices used by other processes or the time the process spends blocked (*e.g.* waiting for $I/O$).

We performed the evaluations at 4 different security levels: **low**, **normal**, **high** and **very high**. At the **low** security level we used 1024 bit RSA/Diffie-Hellman (DH)/Digital Signature Algorithm (DSA) keys and 256 bit Elliptic Curve Cryptography (ECC) keys, at the **normal** those key were of 2048 bit and 224 bit, at the **high** security level 4096 bit and 384 bit, and at the **very high** security level 8192 bit and 512 bit, respectively. We based the **normal** security level on the most used configuration on the internet at the time this paper was written. In all of the cases, only client authenticated itself to the server and the server's certificate was signed with either a 2048 bit RSA key or a 256 bit ECC key, depending on the ciphersuite.

We developed tooling to automate the processes of metric collection and analysis. During the metric collection phase, we executed a client-server connection with each one of the TLS configurations. The resulting metrics from the Handshake and the Record phase were stored to disk for posterior analysis with our second set of tooling.

All of the evaluations were performed on the *Intel(R) Core(TM) i7-4700HQ CPU @ 2.40GHz* processor. We did not collect any measures on typical IoT processors. Despite that, the presented metrics are are still relevant. If collected on an IoT processor, the metrics would maintain a similar proportion, thus the conclusions and analysis presented would still hold true. Moreover, it is possible to run *callgrind* on an IoT processor (either manually or using our automated tooling) and use those metrics for more accurate and device-specific CPU cycle estimation. The same is true for PAPI.

### 3.2. Authentication Costs In TLS

In TLS there are two ways of doing authentication: either by using a PSK or by using asymmetric cryptography. If asymmetric cryptography is used, there are two choices for the algorithm: RSA or ECDSA. With PSK authentication, both of the peers already posses the secret that they use to authenticate one to another. This secret is then used as an input to the Pseudo-Random Function (PRF) when generating the keying material. Mutual authentication is achieved if the integrity check of the *Finished* message is successful. This is only possible if both of the peers generated the same keying material, which can only happen if they used the same PSK as an input to the PRF. Thus, the client and the server authenticate one to another, without any explicit authentication step. For this reason, we can consider PSK authentication to have a cost of 0.

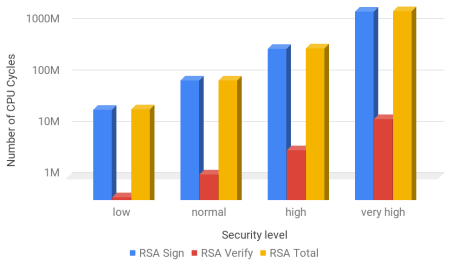


Figure 2: RSA operations costs for all security levels in estimated number of CPU cycles (logarithmic scale)

RSA and ECDSA exhibit different properties and their costs depend on the security level in use. Figure 2 presents the costs in of RSA operations for all security levels, in logarithmic scale. Figure 3 presents the same information for ECDSA. The costs are presented in the estimated number of CPU cycles, with the symbol $M$ meaning *millions*. An analysis of these two graphs reveals numerous differences between the two algorithms. First, while

RSA's cost increase is exponential, the cost increase of ECDSA is logarithmic. This is consequence of the mathematical operations that are at the base of each algorithm. While for RSA this operation is modular exponentiation, for ECDSA it is multiplication of a scalar by a point on the elliptic curve. The second difference is the fact that, across all security levels, RSA is less costly for signature verification, and ECDSA is less costly for signature creation. Finally, the third difference is that while the *total* cost (*i.e.* the sum of signature creation and verification) is lower for RSA at the **low** and **normal** security levels, it is lower for ECDSA at the **high** and **very high** security levels.

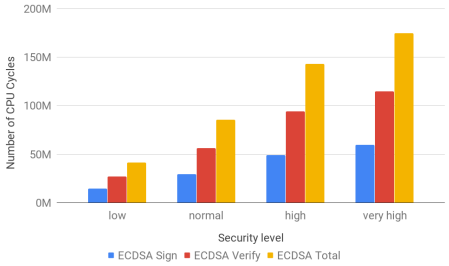


Figure 3: ECDSA operations costs for all security levels in estimated number of CPU cycles

The answer to the question of which one of the algorithms should be used is not straightforward and will depend on the environment. For example, if the scenario is a constrained client and a non-constrained server, RSA would be the least costly choice. If, on the other hand, the server is the constrained node, ECDSA would be the least costly algorithm. If both of the nodes are constrained minimizing *Total* cost is the goal, thus RSA would be the least costly choice for the *low* and *normal* security levels, and ECDSA for the remaining ones. If the objective is to distribute the costs among the peers as evenly as possible, ECDSA is the algorithm to use.

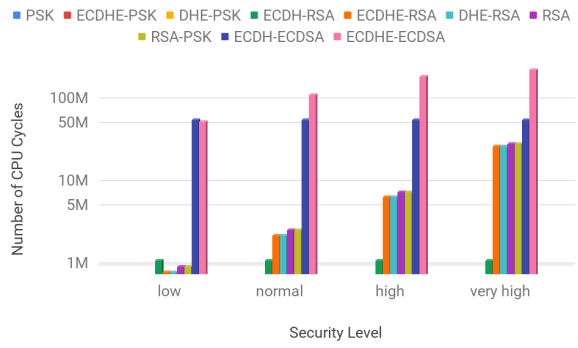


Figure 4: Client authentication costs in estimated number of CPU cycles

Having analyzed the costs of the algorithms that can be used for authentication, we will now analyze the cost of this security service for each one of the key exchange methods for the client and the server. Figures 4 and 5 show the costs of authentication for each one of the key exchange methods and security levels, for the client and the server, respectively. As we have already seen, PSK ciphersuites have an authentication cost of 0 for both of the peers. ECDSA-based ciphersuites use the ECDSA algorithm to create and verify signatures. Similarly, RSA-based ciphersuites use RSA for those purposes. In TLS it hard to talk about the cost of authentication without talking about PFS. If a PFS-enabled ciphersuite is used, an additional piece of information is authenticated in all non-PSK ciphersuites: the *ServerKeyExchange* message. This message contains a signature over the hash of the public *(EC)DH parameters*. This has an implication on the signature creation cost for the server and signature verification cost for the client. All non-PSK key exchange methods which begin with either *ECDHE* or *DHE* incur in that extra cost. This explains why *ECDHE* ciphersuites have a higher cost than *ECDH* ones on the client side.

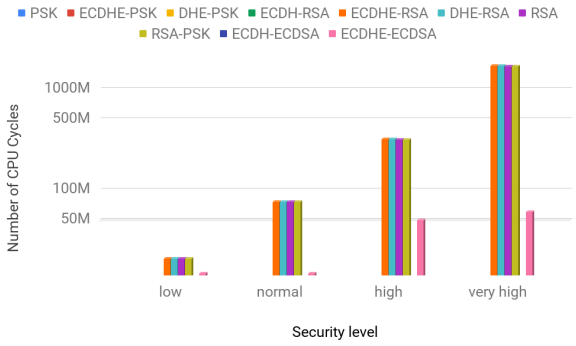


Figure 5: Server authentication costs in estimated number of CPU cycles

By looking at the graphs, it becomes evident that the some key exchange methods can be grouped together by authentication cost. For the client, those groups are: 1 - *PSK*, *ECDHE-PSK*, *DHE-PSK*, 2 - *ECDH-RSA*, 3 - *ECDHE-RSA*, *DHE-RSA*, 4 - *RSA*, *RSA-PSK*, 6 - *ECHD-ECDSA* and 7 - *ECHDE-ECDSA*. For the server, those groups are: 1 - *PSK*, *ECDHE-PSK*, *DHE-PSK*, *ECDH-RSA*, *ECDH-ECDSA*, 2 - *ECHDE-ECDSA*, 3 - *ECDHE-RSA*, *DHE-RSA* and 4 - *RSA*, *RSA-PSK*. For each security level, inside every group, the authentication cost is identical. Group numbers are ordered in ascending cost order, with Group 1 being the least costly one, Group 2 the second least costly one, and so on. All key exchange methods in the same group share a common set of operations that

are performed to provide authentication.

### 3.3. PFS Costs In TLS

In TLS there are two ways of achieving PFS: either by using the DH algorithm or its ECC counterpart Elliptic Curve Diffie-Hellman (ECDH). In both algorithms, the same basic operations are performed by each peer in sequence: generate a public/private keypair, exchange the public values and derive the shared secret.

In both ECDH and DH, two basic operations are performed by each peer: first, a public/private ECC keypair is generated, followed by generation of the shared secret. In ECDH, the resulting shared secret will be a 2D $(x, y)$ coordinate on the curve. In TLS the $y$ value is discarded and $x$ is used as the premaster secret. In DH the resulting shared secret will be a scalar, which is used as the premaster secret.

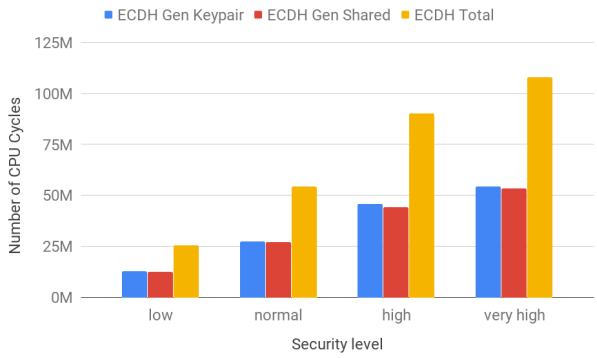


Figure 6: ECDH costs in estimated number of CPU cycles

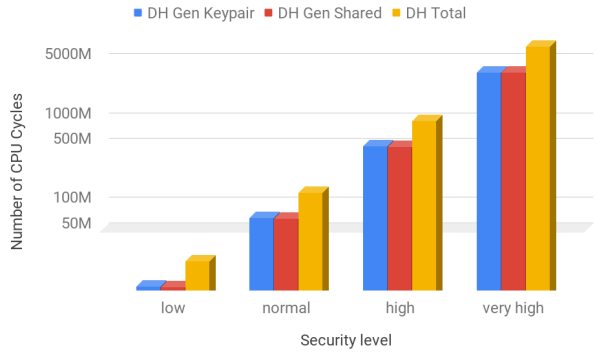


Figure 7: DH costs in estimated number of CPU cycles (logarithmic scale)

Figures 6 and 7 show the cost of ECDH and DH operations, respectively. In both algorithms, computing the private key is cheap, since it is just a randomly generated number. The costly part is the computation of the public key and the shared secret, since for both it involves multiplications of a scalar by a point on the elliptic curve. For both, ECDH and DH, the cost of computing the public key and the shared secret is very similar.

One key difference between ECDH and DH is that while in the first the cost increase is logarithmic, in the second it is exponential. This is a consequence of the underlying mathematical operations of each algorithm. Unlike in our comparison of RSA and ECDSA in Section 3.2, the answer to which one of the algorithms is less costly, is straightforward. In RSA and ECDSA the cost the signature creation and verification operations is different, so the choice of the least costly option depended not only on the security level, but also on whether we were optimizing for the signature creation or verification. In ECDH and DH, the total cost is almost evenly divided between the keypair and the shared secret generation. Thus, we can make our decision simply by comparing the *Total* cost from figures 6 and 7. If the **low** security level is being used, DH is the least costly choice, if the **normal** or any security level above is being used, ECDH is. The logarithmic and exponential properties of ECDH and DH, respectively, are also visible by equations and shapes of the trendlines.

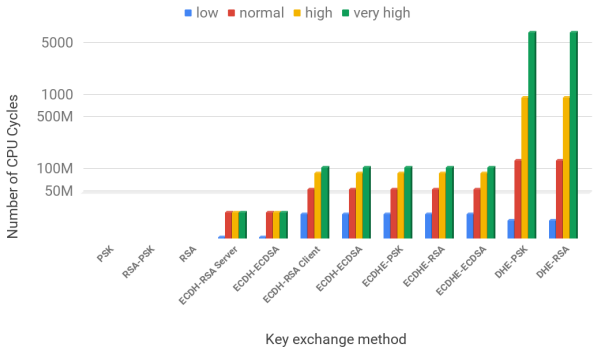


Figure 8: PFS and ECDH ciphersuite costs in estimated number of CPU cycles (logarithmic scale)

Having analyzed the costs of the algorithms that can be used for PFS, we will now analyze for the cost of this security service, for each one of the ciphersuites for the client and the server. Figure 8 shows the PFS and *ECDH* ciphersuites costs, for both, the client and the server. Even though the *ECDH* key exchange does not offer PFS, it is still closely related to the *ECDHE* one, since both use the ECDH algorithm. The additional costs for *ECDH* ciphersuites are significant, and in our evaluated scenario where only the server authenticates to the client, there is no distinction in costs between *ECDHE* and *ECDH* ciphersuites for the client. Instead of presenting multiple figures, we decided to presented the costs of PFS and the *ECDH* ciphersuites in a single one.

An analysis of the figure, shows that we can group the costs into 4 groups, presented in increasing cost order: 1 - non-*ECDH(E)/DHE* ciphersuites, which have a total cost of 0; 2 - *ECDH* server-side ciphersuites; 3 - *ECDH* client-side and *ECDHE* ciphersuites and 4 - *DHE* ciphersuites.

### 3.4. TLS Handshake Costs

Having analyzed the costs of the authentication and PFS security services in TLS, we will now discuss the total cost of the Handshake. Figure 9 shows the Handshake costs for the client, for each one of the key exchange methods and security levels. Figure 10 shows the same information for the server.

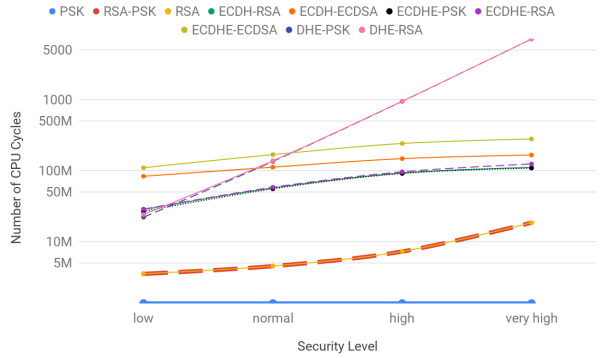


Figure 9: Client Handshake costs in estimated number of CPU cycles (logarithmic scale)

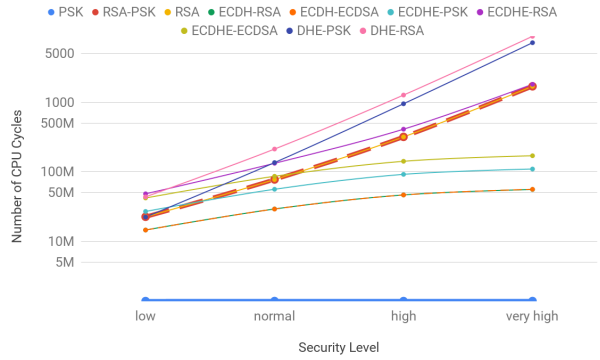


Figure 10: Server Handshake costs in estimated number of CPU cycles (logarithmic scale)

An analysis of the graphs shows, that that while there are 10 unique key exchange methods, the costs of some of them are identical. If two or more key exchange methods have similar costs, we say that they belong to the same *cost group*. A *cost group* can also contain a single key exchange method, if its costs are significantly different from all others. In figures 9 and 10 two or more key exchange methods belong to the same cost group if their lines overlap throughout all security levels. We can find 6 cost groups at the client-side: 1 - *PSK*; 2 - *RSA*, *RSA-PSK*; 3 - *ECDHE-PSK*, *ECDHE-RSA*, *ECDH-RSA*, 4 - *ECDH-ECDSA*; 5 - *ECHDE-ECDSA*; 6 - *DHE-PSK*, *DHE-RSA*. For the server, 7 groups can be identified: 1 -*PSK*; 2 - *ECDH-RSA*, *ECDH-ECDSA*;3 - *ECHDE-PSK*, *ECDHE-ECDSA*; 4 - *RSA*, *RSA-PSK*; 5 - *ECDHE-RSA*; 6 -*DHE-PSK*; 7 - *DHE-RSA*.

The groups are presented in ascending cost order. For both, the client and the server, the *PSK* key exchange is, by far, the cheapest option. The most expensive key exchange method depends on the peer and the security level. If we had to choose one option for the title of the most expensive one, *DHE-RSA* would be the answer, since this is true starting from **normal** security level for the client and **low** security level for the server. *DHE-PSK* follow a similar trend, especially for the client. Once again we see the advantage of ECC, with the cost increase of key exchanges that use *ECDH(E)* and/or *ECDSA* being much lower than of the ones that use *DHE* and *RSA* instead. This is a consequence of logarithmic (for ECC) vs exponential (for non-ECC) cost increase.

As a result of our analysis, we derived a formula that decomposes the costs of the TLS Handshake into individual parts: $HandshakeCost = TLSOverhead + AuthCost + PFSCost + AdditionalCosts$. The *TLS Overhead* is the cost of the *PSK* key exchange. *Auth Cost* is the cost of authentication. *PFS* cost is the cost of PFS. *Additional Costs* are the extra costs in which the peers incur when creating and parsing TLS messages not present in PSK Handshake. The majority of those costs come from the *Certificate* message, present in all RSA and ECDSA ciphersuites. The server has to write the *Certificate* message to the Record layer and the client has to parse the *der*-encoded certificate into internal fields, while performing some checks along the way, such as the *Not Valid Before/After* fields.

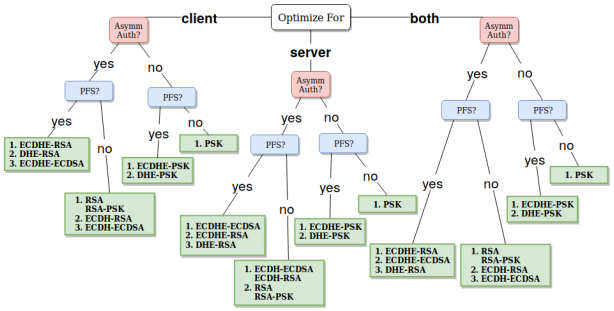


Figure 11: **normal** security level decision tree for the cheapest key exchange

Selecting the cheapest TLS configuration for one's needs and limitations is not a straightforward

task. For this reason, we developed a set of decision trees that simplify this process. Figure 11 shows the decision tree for the **normal** security level. At the terminal nodes, the key exchange methods are presented in order of the cheapest to the most expensive one. If two exchange methods belong to one of the cost groups presented above, they are are presented as such in the figures too. The cost differences between the ciphersuites within the same cost group are very small and can be ignored. If the choice is to optimize for the client/server, ciphersuites are ordered to minimize client/server costs. If the choice is choice is to optimize for both, the ciphersuite are ordered to minimize the total costs, *i.e.* the sum of the costs for the client and the server.

### 3.5. Confidentiality and Integrity Costs In TLS

Having analyzed the cost of authentication, PFS and the Handshake, we will now analyze the cost of the confidentiality and integrity services. In TLS, it does not make sense to analyze these services separately, since they're offered as a whole as a part of the ciphersuite. While the establishment of a secure communication channel between two peers is known as the Handshake phase, the posterior use of that channel to exchange data under the guarantees of confidentiality and integrity (if applicable to the ciphersuite) is known as the Record phase.

There are 26 unique symmetric encryption algorithm/hash function pairs available in *mbedTLS 2.7.0*. A total of 4 hash functions is available: $MD5$, $SHA-1$, $SHA-256$ and $SHA-384$. We profiled the costs of each one of 26 unique pairs, in terms of estimated number of CPU cycles. The results are presented in Figure 12. As expected, their cost grows linearly with the amount of encrypted bytes. The analysis of the obtained data shows that *3DES* with *SHA* is the most costly combination of an encryption algorithm with a MAC function, while AEAD encryption algorithms (*AES* with *GCM* and *CCM* modes) are the least costly block cipher algorithms. *CAMELLIA* algorithms are more costly than their *AES* counterparts.

An analysis of the hash function costs, showed that *SHA-256* is the most costly hash function, while *MD5* is the least costly one. However, it is important to consider that *MD5* and *SHA-1* are nowadays considered insecure and vulnerable to numerous attacks. For this reason, one should preferably choose between *SHA-256* and *SHA-384* and the latter is the least expensive one. For both of the cases, AEAD algorithms and non-AEAD algorithms combined with a hash function, the majority of the cost comes from data encryption/decryption.

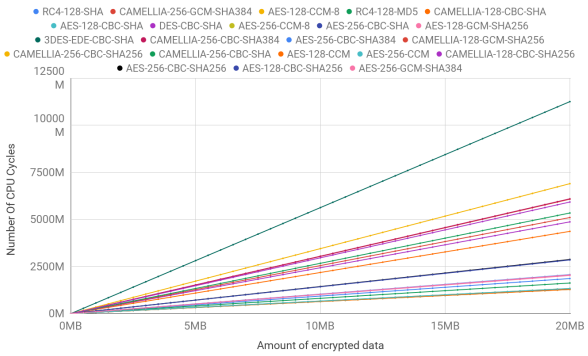


Figure 12: Confidentiality and integrity cost with different algorithms in estimated number of CPU cycles

It only makes sense to heavily optimize handshake if the amount of transmitted data is small. But what exactly is a *small* amount of data? In order to answer that question we profiled the costs of encrypting data with the *AES-128-GCM* (**low** and **normal** security levels) and *AES-256-GCM* (**high** and **very high** security levels) more thoroughly. We selected those algorithms because they were among the cheapest ones to provide the required security level and are preferred by browsers, such as *Google Chrome 67*. The cost of encryption and decryption for those algorithms are similar. Our analysis yielded the following formulas for the cost of encryption with *AES-128-GCM* and *AES-256-GCM*, respectively: $NumCC = 104 * NumBytes + 22680$ and $NumCC = 105 * NumBytes + 22740$ ($R^2 = 1$ for both). $NumCC$ is the number of CPU Cycles and $NumBytes$ is the number of bytes encrypted. As the formulas show, the cost of *AES-256-GCM* is slightly larger than of *AES-128-GCM*. This is expected due to the larger key size of the first one.

The derived formulas can be used to answer the question of when the costs spent on data encryption equate the costs spent on performing the handshake. In 85% of cases for the client, less than 2 MB of data need to be exchanged for the Record phase costs to equate the Handshake costs. For the server, that percentage is 72.5%. Currently, *ECDHE-ECDSA* and *ECDHE-RSA* at the normal security level are the most used TLS configurations. For those ciphersuites, the client only needs to send about 1.62MB for the first key exchange and 560KB for the second one. For the server, those numbers are 830KB, and 1, 27MB, respectively.

### 3.6. PAPI Time Measurements and Comparisons With Estimates

The previous analysis was done with the estimated number of CPU cycles obtained from *valgrind/callgrind*. We wanted to know how accurate those estimations were. For this reason, we used

*papi* to obtain time metrics. *The iron law of processor performance* states that the time taken by a program execution is proportional to the number of instructions ($I$), the average number of cycles per instruction ($CPI$) and the amount of time per each processor cycle ($CT$): $CPUTime = I * CPI * CT$. The number of CPU cycles can be approximated as follows: $CPUCycles = I * CPI$. This means that the program execution time can be expressed as a product of the number of CPU cycles and a constant: $CPUTime = CPUCycles * CT$. Thus, we expect the graphs with time in the $y$ axis to look similar, but have smaller values. The results obtained with PAPI matched our expectations.

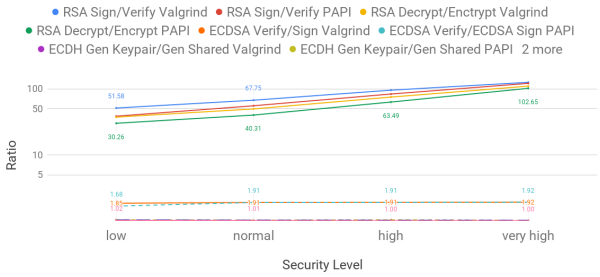


Figure 13: Ratio of time taken between a set of related operations for PAPI and *valgrind*/*callgrind* (logarithmic scale)

Not only the graphs resulting from PAPI analysis for the Handshake, authentication and PFS look very similar to the ones resulting from *valgrind* estimates (differing on the $y$ axis values, of course), but the ratio between each related set of operations is also similar. Figure 13 depicts the ratio between RSA signature creation and verification, RSA encryption and decryption, ECDSA signature verification and creation, ECDH keypair and shared secret generation and DH keypair and shared secret generation for both, *valgrind* and PAPI. We purposefully divided the more costly operation by the least costly one, so that all of the ratios are positive. The results are presented in logarithmic scale. An analysis of the figure shows that both, the *valgrind* and the PAPI ratios are very similar. This is specially true for the ECDH, DH and ECDSA. For DH there is no difference, for ECDH, *valgrind*'s ratio is 0.96% larger at the *high* and **very high** security levels, and for ECDSA the ratio differs only for the **low** security level, where it's 9.2% larger in *valgrind* than in PAPI. The largest difference in ratio is observed for opeartions that use RSA, but this difference gets smaller as the security level increases. In fact, this is the general trend for all of the cases. *Valgrind*'s RSA's sign/verify ratio is 24.87% larger than PAPI's for the **low** security level, but only 3.6% larger for the **very high** security level. Following this trend, *callgrind*'s RSA's encrypt/decrypt ratio is 19.41% higher than PAPI's for the low security level, and 7.3% higher for the **high** security level.

## 4. Related Work

The majority of the work done in the area proposes a solution that is either tied to a specific protocol, such as CoAP, or requires an introduction of a third-party entity, such as the trust anchor in the case of the S3K system[8] or even both. This has two main issues. First, a protocol-specific solution cannot be easily used in an environment where (D)TLS is not used with that protocol. Second, the requirement of a third-party introduces additional cost and complexity, which will be a big resistance factor in adopting the technology. This is specially true for developers working on personal projects or projects for small businesses, leaving the communications insecure in the worse case scenario.

The work that is neither tied to a specific protocol, nor requires an introduction of a third-party is focused on DTLS. With the standards such as *CoAP over TCP and TLS*, being in active development[6], it is important to study TLS optimization. CoAP[7] is often referred to as the "HTTP protocol for constrained devices". *CoAP over TCP and TLS* does not explore any TLS optimizations, and since any IoT device using it in the future would benefit from them, this is an important area to explore.

## 5. Conclusions

This dissertation presented the most complete and detailed analysis of the costs of the TLS protocol that exists to date. The herein presented results can be used by software engineers and security professionals to make informed decisions about the security/cost trade-offs, specific to the environment.

We analyzed the costs of TLS at the low, middle and high levels. At the low level, we studied and compared, the costs of each one of the algorithms that enable the security services of TLS. We concluded that the choice of the cheapest algorithm, depended not only on the key size, but also on the peer whose costs we wanted to minimize. At the middle level, we analyzed the costs of the security services of authentication and PFS. We answered the question of how much each security service costs in terms of the number of CPU cycles and time. At the high level, we analyzed the cost of the Handshake as a whole. Its cost was dissected in light of the costs of the security services. As a result, we derived a formula that decomposes the costs of the TLS Handshake into its individual parts.

While the focus thorough-out this work as on the costs of the Handshake, we also evaluated the costs of the security services of confidentiality and

integrity. The asymmetric encryption algorithms were profiled in order to answer the question of when the costs of the Handshake equate the costs of confidentiality and integrity. Our analysis showed that, in a typical configuration that is used on the internet, less than 1.7MB of data needs to be exchanged between the peers in order for that to happen. For this reason, we concluded that it only made sense to heavily optimize the Handshake if the amount of exchanged data is small.

In the process of the work on this thesis, we not only evaluated TLS at the level that was never done before, but also contributed to the global security community, by:

1. Contributing to the specification of the TLS protocol version 1.3, and to the lesser extent, of DTLS protocol version 1.3
2. Finding and reporting a security vulnerability in *mbedTLS*, which was assigned a CVE with id *CVE-2018-1000520*

For the evaluation, we used the TLS implementation of the *mbedTLS* library, which is one of the most popular TLS implementation libraries for embedded systems. We used two cost metrics First, we did a thorough analysis of TLS, by analyzing the number of estimated CPU cycles obtained with *callgrind*. After that, we showed that the estimates are close to the real values, by comparing them to the time metrics obtained directly from the processor's registers. The results presented here were obtained on a powerful, modern-day computer. Despite that, they are still relevant when considering the costs on constrained IoT devices. While on a different device, the absolute cost numbers will be different, they would still maintain a similar proportion one to another and follow a similar trend. Moreover, the developed tooling can be used to obtain profiling results on any machine, thus giving device-specific cost information.

## 6. Future Work

In our work we obtained and analyzed a large number of metrics obtained with *callgrind*. While *callgrind* provides only an estimates of the CPU cycles used, we later showed that they reflect real values by comparing them with the time results obtained with PAPI. However, it is important to remember that those metrics were obtained on a general-purpose computer. While we fixed the CPU frequency and disabled some hardware optimizations, the environment on an IoT device is still expected to be very different, due to factors such as a lower clock frequency, memory and cache size. Thus, it would be interesting to obtain metrics on an IoT device.

Another characteristic of numerous IoT devices is limited power (*e.g.* using battery as a power source). Thus, it would be interesting analyze the cost of TLS in terms of power usage. This would also allow to reach conclusions, such as: *Using the TLS configuration X would reduce the device's battery life by Y days.*

**Acknowledgements**

I would like to thank my supervisors Ricardo Chaves and Aleksandar Ilic for their guidance and motivation throughout this dissertation. I would also like to express my gratitude to my mother, grandmother and girlfriend for their help and support during this work.

## References

[1] Addition to tls 1.3 contributors list. `https://github.com/tlswg/tls13-spec/commit/43461876882a60251ecf24fb097f0ce2d7be4745`. (Accessed on 01/11/2018).

[2] Commits by iluxonchik tlswg/dtls13-spec. `https://github.com/tlswg/dtls13-spec/commits?author=iluxonchik`. (Accessed on 11/18/2018).

[3] Nvd - cve-2018-1000520. `https://nvd.nist.gov/vuln/detail/CVE-2018-1000520`. (Accessed on 10/10/2018).

[4] ssl_server.c and ssl_client1.c are using an sha-1 signed certificate. `https://github.com/ARMmbed/mbedtls/issues/1519`. (Accessed on 10/15/2018).

[5] update test rsa certificate to use sha-256 instead of sha-1 by iluxonchik. `https://github.com/ARMmbed/mbedtls/pull/1520`. (Accessed on 10/15/2018).

[6] C. Bormann, S. Lemay, H. Tschofenig, K. Hartke, B. Silverajan, and B. Raymor. CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets. Internet-Draft draft-ietf-core-coap-tcp-tls-11, IETF Secretariat, December 2017. `http://www.ietf.org/internet-drafts/draft-ietf-core-coap-tcp-tls-11.txt`.

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

[8] S. Raza, L. Seitz, D. Sitenkov, and G. Selander. S3k: Scalable security with symmetric keys—dtls key establishment for the internet of things. *IEEE Transactions on Automation Science and Engineering*, 13(3):1270–1280, 2016.

[9] E. Rescorla. The transport layer security (tls) protocol version 1.3. RFC 8446, RFC Editor, August 2018.