Table of Contents

Transport Layer Security Protocol For Internet Of Things

Illya Gerasymchuk
illya@iluxonchik.me

Instituto Superior Tecnico
Supervisors: Ricardo Chaves, Aleksandar Ilic

Abstract. TLS is one of the most used communication security protocols in the world. It comes with many configurations. Each configuration offers a set o security services, which has an implication on the security level and computational cost. Not all of those configurations can be used with the resource constrained Internet Of Things (IoT) devices, due to the high computational and memory demands. Most of the existing work focuses on DTLS and cannot be easily integrated with existing deployments. Existing work fails to evaluate the cost of various TLS configurations and its security services. This work focuses on cost analysis of the security services of the TLS protocol. We evaluate the number of CPU cycles used and time taken by each TLS configuration and security service. Software developers can use this information to make security/cost trade-offs based on the environment needs and limitations.

Keywords: TLS, DTLS, SSL, IoT, cryptography, protocol, lightweight cryptography

1 Introduction

In recent years there has been a sharp increase in the number of IoT devices and this trend is expected to continue[1]. 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. Examples of IoT devices include temperature sensors, smart light bulbs and physical activity trackers. Even a salt shaker[2] can now be part of the global network.

The IoT technology provides many benefits, from personal comfort to transforming entire industries, mainly due to increased connectivity and new sources for data analysis. The technological development, however, tends to focus on innovative design rather than on security. IoT devices frequently connect to networks using inadequate security and are hard to update when vulnerabilities are found.

This lack of security in the IoT ecosystem has been exploited by the the *Mirai* botnet[3] when it overwhelmed several high-profile targets with massive Distributed Denial-Of-Service (DDoS) attacks. This is the most devastating attack involving IoT devices done to date. However, the *Reaper* botnet[4] could be even worse if it is ever put to malicious use. Similar attacks will inadvertently come in the future.

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[5]. Our name can be found in the document specifying TLS 1.3[6]. Although to the lesser extent, we have also contributed to DTLS 1.3 specification[7]. We have found a security issue within the TLS implementation of the *mbedTLS* library. We reported it and it has been assigned a *CVE* with the id *CVE-2018-1000520*[8]. It is an authentication problem, 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[9][10].

1.1 Motivation

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 steal your money. Similarly, when you are transferring funds via online banking, you want the contents of that operation to be invisible to an observer, for privacy reasons. It is also desirable that no party is able to tamper with the data en transit, as it may lead to undesired consequences, such as the transfer of a larger amount than intended. Proper communication security allows those goals to be achieved.

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. For example, when you are downloading software updates, while data confidentiality is probably not a concern, data authenticity and integrity, are. In TLS, it is possible for a connection to only offer authenticity and integrity, without offering confidentiality. 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.

Existing work does not explore the computational costs of the security services available in TLS. Examples of such costs are the number of CPU cycles executed, time taken and power used. Thus, developers wishing to deploy the TLS protocol in constrained environments do not have a resource that would help them in choosing a TLS configuration appropriate to the environment's needs and limitations.

TLS is designed to run on top of a reliable, connection-oriented protocol, such as TCP. DTLS is the version of TLS that runs on top of an unreliable transport protocol, such as UDP. Most IoT devices have very limited processing power, storage and energy. Moreover, the performance of TCP is known to be inefficient in wireless networks, due to its congestion control algorithm. This situation is worsened with the use of low-power radios and lossy links found in sensor networks. Therefore, in many cases the use of TCP with IoT is not the best option. For this reason, DTLS, which runs on top of UDP, is used more frequently in such devices. While the work of this dissertation will be focused on TLS, the majority of it can also be applied to DTLS. This is a consequence of DTLS being just an adaption of TLS over unreliable transport protocols, without changes to the core protocol.

There are numerous IoT devices, each one with different hardware capabilities and security requirements. For example, some IoT devices have the resources to use public key cryptography, while for others symmetric cryptography is the only option. In some cases, the communicating devices require data authenticity, confidentiality and integrity (e.g. when logging in into a device), while in others data authenticity and integrity is enough (e.g. when transferring updates).

TLS was not designed for the constrained environment of IoT. Despite that, it is a malleable protocol and can be configured to one's needs. In essence, it is a combination of various security algorithms that together form a protocol for communication security. If configured properly, it is possible to use it in the context of IoT.

The majority of existing work on (D)TLS optimization proposes a solution that is either tied to a specific protocol, such as Constrained Application Protocol (CoAP), or requires an introduction of a third-party entity, such as the trust anchor in the case of the S3K system[11] 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 especially true for developers working on personal projects or projects for small businesses, leaving the communications insecure in the worse case scenario. Therefore a solution that is protocol independent and fully compatible with the (D)TLS standard and existing infrastructure is desired.

Another issue with the existing literature is that it almost exclusively focuses on DTLS optimization and not all of it can be applied to TLS. Herein we want to further explore TLS optimization. There is clearly a need for that, especially with CoAP over TCP and TLS standard[12] being currently developed. The mentioned 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.

1.2 Objectives

(D)TLS is a complex protocol with numerous possible configurations. Each configuration provides different set security services and a different security level. This has a direct impact on the resource usage. Thus, the cost of a (D)TLS connection can be lowered, by using an appropriate configuration. Typically, this involves making security/cost trade-offs. Optimizing the connection cost by selecting one of the numerous configurations available in (D)TLS meets our goals of being protocol independent, fully compatible with existing infrastructure and targeting TLS optimization specifically.

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 will give a general overview of of the costs of the TLS protocol as a whole and of its individual parts. This is will allow us to answer questions such as *How much will we save if we use protocol X instead of Y for authentication?*. Thus, performing evaluations on specific IoT hardware or analyzing TLS-specific optimizations on it is outside of the scope of this paper.

In order to achieve our goal, the cost of each individual security service will be evaluated. 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 evaluated the *mbedTLS 2.7.0*'s[13] implementation of the TLS protocol version 1.2. *mbedTLS* is among the most popular libraries with a TLS implementation for embedded systems. TLS protocol version 1.2 is currently the most used version of TLS on the internet [14]. The work on the dissertation started before TLS protocol's version 1.3 specification was finished and there were no embedded device libraries which implemented it. For this reason we did not evaluate TLS 1.3. Despite that, the results obtained in this work apply to it as well, since the core functionality of the security services remained mostly unchanged.

We used two cost metrics: the estimated number of CPU cycles and the time taken. The time values were read direcly from the processor's registers. The profiled instructions are CPU-bound, thus the number of CPU cycles will be proportional to time, as will later show in our analysis. Later in the text we will show that our esitmates do reflect real values, by comparing them to time measurments obtained directly from the CPU registers. Section 5.1 contains a detailed description of the evaluated costs and their limitations.

1.3 Results

In summary, the results of this work are enumerated as follows:

1. Evaluate the costs of the security services of confidentiality, integrity, Perfect Forward Secrecy (PFS) and authentication in TLS
2. Evaluate and compare the costs of various alternative algorithms which can be used to provide each one of the security services
3. Evaluate and compare the costs of all of the possible TLS configurations present in *mbedTLS 2.7.0*
4. Contribute to the TLS protocol's version 1.3 specification
5. Contribute to the DTLS protocol's version 1.3 specification
6. Find and report a security vulnerability present in *mbedTLS 2.7.0*
7. Find, report and submit a patch to fix a bug present in *mbedTLS 2.7.0*

1.4 Structure of The Document

The document is organized as follows: Section 2 describes the background. It introduces some of the concepts that will be used throughout the document. Section 3 describes the TLS and DTLS protocol versions 1.2 and 1.3, with a focus on the version 1.2 since it is the latest and the most used version of the protocol at the time we started this work (version 1.3 was still in draft mode). Section 4 describes all of the related work done in the area and the current state of the art. Section 5 describes the objectives of the work and evaluation are described in more detail. Section 6 covers the evaluation's methodology and limitations. In Section 7 we evaluate the costs of the various TLS configurations and their individual parts. Finally, the conclusion of the work is done in Section 8.

2 Background

TLS is a complex protocol that relies on various concepts to provide security. The most relevant ones will be described here.

In a typical scenario, TLS uses Asymmetrical Cryptography (AC) for peer authentication and Symmetrical Cryptography (SC) for bulk data encryption and integrity protection, for this reason this topic will be covered in Section 2.1. Section 2.2 covers the most common way of peer authentication: public key certificates. Authenticated Encryption With Additional Data (AEAD) ciphers offer various advantages in the context of IoT, particularly less computational and spacial overhead. Furthermore, they are the only type of ciphers that can be used in TLS 1.3. For those reasons, they're covered in Section 2.3. When compared to other public key cryptography approaches, ECC offers shorter keys, lower processing requirements and lower memory usage for equivalent security strength, being heavily used in TLS. An overview of ECC in presented in Section 2.4.

2.1 Symmetric vs Asymmetric Cryptography

AC is more expensive than SC in terms of performance. There are two main reasons for this. First, larger key sizes are required for an AC system to achieve the same level of security as in a SC system. Second, `CPU`s are slower at performing the underlying mathematical operations involved in AC, namely exponentiation requires $O(log e)$ multiplications for an exponent e. The 2016 `NIST` report [15] suggests that an AC algorithm would need to use a secret key with size of `15360 bits` to have equivalent security to a `256-bit` secret key for a SC algorithm. This situation is ameliorated by ECC, which requires keys of `512 bits`, but it is still slower than using SC. The 2017 `BSI` report [16] (from the German federal office for information security) suggests similar numbers.

Another argument for avoiding the use of AC algorithms as much as possible, is that they require additional storage space. This can be a problem for many IoT devices, like `class 1` devices according to the terminology of constrained-code networks[17] which have approximately `10KB` of RAM and `100KB` of persistent memory. We measured and compared the resulting size of the *mbedTLS 2.7.0* library[13] binary when it was compiled with and without the RSA module (located in the `rsa.c` file). The conclusion is that that using the `rsa.c` module adds an overhead of about `32KB`.

2.2 Public Certificates and Certificate Chains

A public key certificate, also known as a digital certificate, is an electronic document used to prove the ownership of a public key. This allows other parties to rely upon assertions made by the private key that corresponds to the public key that is certified. In the context of (D)TLS, certificates serve as a guarantee that the communication is done with the claimed entity and not someone impersonating it.

A Certification Authority (CA) is an entity that issues digital certificates. There are two types of CAs: the **root CAs** and the **intermediate CAs**. An intermediate CA is provided with a certificate with signing capabilities signed by one of the root CAs. A **certificate chain** is a list of certificates from the root certificate to the end-user certificate, including any intermediate certificates along the way. In order for a certificate to be trusted by a device, it must be directly or indirectly issued by a CA trusted by the device.

In (D)TLS, the certificates are in the `X.509` format, defined in `RFC 5280`[18].

2.3 AEAD Ciphers

Authenticated Encryption (AE) and AEAD are forms of encryption which simultaneously provide confidentiality, integrity and authenticity guarantees on the data. An AE cipher takes as input a `key`, a `nonce` and a `plaintext` and outputs the pair `(ciphertext, MAC)`, if it is encrypting and does the inverse process, while also performing the Message Authentication Code (MAC) check if it is decrypting.

AEAD is nothing more than a variant of AE, which comes with an extra input parameter that is additional data, that is **only authenticated, but not encrypted**. Some AEAD ciphers have shorter authentication tags (*i.e.* shorter MACs), which makes then more suitable for low-bandwidth networks, since the messages to be sent are smaller in size.

2.4 ECC

public key cryptography is based on the use of one-way math functions. Such functions make it easy to compute the answer given an input, but hard to compute the input given the answer. For example, RSA uses factoring as the one one way function: it is easy to multiply large numbers, but it is hard to factor them.

ECC is based on elliptic curves, which are set of points (x, y) that are solutions to the equation $y^2 = x^3 + ax + b$, where $4a^3 + 27b^2 \neq 0$. Depending on the value of a and b, elliptic curves assume different shapes on the plane.

The security of ECC is based on the elliptic curve discrete logarithm problem. It states that scalar multiplication is a one way function. To exemplify, given a curve $E(\mathbb{Z}/p\mathbb{Z})$ and points Q and P on that curve $Q, P \in E(\mathbb{Z}/p\mathbb{Z})$, where Q is a multiple of P, the elliptic curve discrete logarithm problem states that finding the integer k, such that $Q = kP$ is a very hard problem.

3 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 **privacy** and **integrity** between the two communicating peers. Privacy 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. The message flow is depicted in Figure 3 and described in more detail in Section 3.2.

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 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 MAC or an 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.

Despite using PKC, TLS does **not** provide **non-repudiation services**: neither **non-repudiation with proof of origin**, which addresses the peer denying the sending of a message, nor **non-repudiation with proof of delivery**, which addresses the peer denying the receipt of a message. This is due to the fact that instead of using **digital signatures**, either a keyed MAC or an AEAD cipher is used, both of which require a secret to be **shared** between the peers.

It is not required to use all of the tree 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. There are some differences regarding this claim between TLS 1.2[19] and TLS 1.3. For example, while in the first there is a `null` cipher (no authentication, no confidentiality, no integrity), in the latter this is not true, since it deprecated all non-AEAD ciphers in favor of AEAD ones.

The terms Secure Sockets Layer (SSL) and TLS are often used interchangeably, but one is a predecessor of another - SSL 3.0[20] served as the basis for TLS 1.0[20].

Section 3.1 will begin with a brief overview of the various sub-protocols that compose TLS. The TLS Record Layer will be described in sufficient detail for the TLS Handshake Protocol description that follows in Section 3.2. The way each record is processed when sending and receiving data is covered in Section 3.3. The symmetric keys involved in cryptographic operations that provide confidentiality and security are described in Section 3.4. Section 3.5 explains how those keys are generated in TLS 1.2. There are various methods that the client and the server can use to exchange keys, those will be covered in Section 3.6. The TLS Extension mechanism will be covered in Section 3.7. There are various differences from TLS 1.2 to 1.3 and those that were not covered in the previous sections will be in Section 3.8. This section ends with an outline of the main differences from DTLS to TLS in Section 3.9.

3.1 TLS (Sub)Protocols

TLS is composed of several protocols, which are illustrated in Figure 2 and briefly described below:

- **TLS Record Protocol** - the lowest layer in TLS. It takes messages to be transmitted, fragments the data into manageable blocks, optionally compresses them, encrypts them and transmits the result. When the data is received, the reverse process is done. The TLS Record Protocol is located directly on top of **TCP/IP** and it serves as an **encapsulation for the remaining sub-protocols** (`4` in case of TLS 1.2 and `3` in case of TLS 1.3). To the **Record Protocol**, the remaining sub-protocols are what `TCP/IP` is to `HTTP`. A TLS Record is comprised of 4 fields, with the first 3 comprising the TLS Record header. The first field is a 1-byte record `type` specifying the type of record that is encapsulated (ex: value `0x16` for the handshake protocol). The second is a 2-byte `TLS version` field. The third is a 2-byte `length` field specifying the length of the data in the record, excluding the header itself (this means that TLS has a maximum record size of `16384` bytes). The fourth is a `fragment` field, containing `length` bytes of data that is transparent to the Record layer and should be dealt by a higher-level protocol. That higher-level protocol is specified by the `type` field. This is illustrated in Figure 1.
- **TLS Handshake Protocol** - the core protocol of TLS. It allows the communicating peers to **authenticate** one to another and to negotiate the connection state. In TLS 1.2 a **cipher suite** and a **compression** method are negotiated. In TLS 1.3, a **cipher suite** and a **key exchange** algorithm are negotiated. The agreed upon **cipher suite** is used to provide the previously described security services. In TLS 1.2, a **cipher suite** consists of a **cipher spec**, a **key exchange** algorithm and a Pseudo-Random Function (PRF), which is used for key generation. In TLS 1.2, **cipher spec** defines the message encryption algorithm and the message authentication algorithm. In TLS 1.3, the term **cipher spec** is no longer present, since the **ChangeCipherSpec** protocol has been removed. The concept of **cipher suite** has been updated to define the pair consisting of an AEAD algorithm and a hash function to be used with HMAC-based Extract-and-Expand Key Derivation Function (HKDF). In TLS 1.3 the **key exchange** algorithm is negotiated via extensions.
- **TLS Alert Protocol** - allows the communicating peers to signal potential problems.
- **TLS Application Data Protocol** - used to transmit application data messages securely using the security parameters negotiated during the **Handshake Protocol**. The messages are treated as transparent data to the record layer.
- **TLS Change Cipher Spec Protocol** (removed in TLS 1.3) - used to activate the initial **cipher spec** or change it during the connection.

3.2 TLS 1.2 Handshake Protocol

The Handshake Protocol is responsible for negotiating a **session**, which will then be used in a **connection**. There is a difference between a TLS session and a TLS connection:

- **TLS session** - association between two communication peers that is created by the **TLS Handshake Protocol**, which defines a set of negotiated parameters (cryptographic and others, such as the compression algorithm, depending on the TLS version)

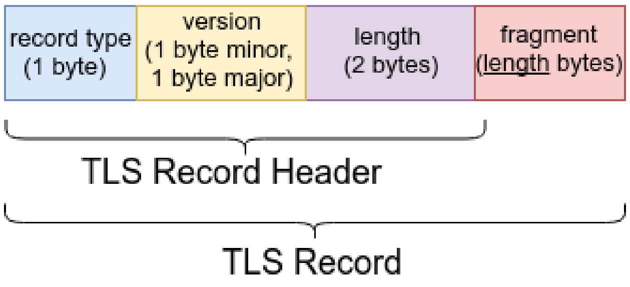


Fig. 1: TLS Record header

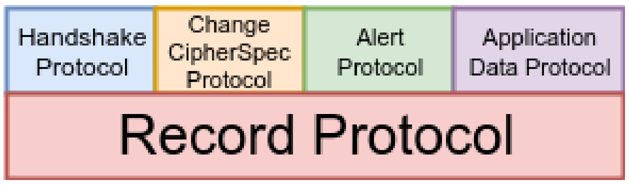


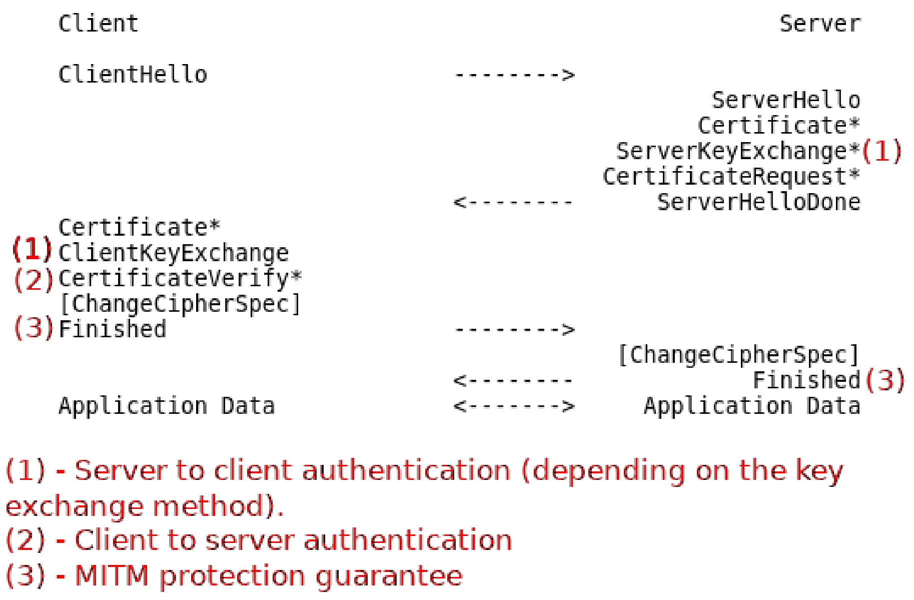
Fig. 2: TLS (Sub)protocols and Layers

that are used by the **TLS connections associated with that session**. A single **TLS session** can be shared among multiple **TLS connections** and its main purpose is to avoid the expensive negotiation of new parameters for each **TLS connection**. For example, let us say that a Hypertext Markup Language (HTML) page is being downloaded over the HTTPS and that page references some images from that same server using HTTPS links. Instead of the web browser negotiating a new TLS session for every single image again, it can re-use the the one it has established to download the HTML page, saving time and computational resources. Session resumption can be done using various approaches, such as **session identifiers**, described throughout `Section 7.4` of `RFC 5246`[19] and **session tickets**, defined in `RFC 5077`[21].

– **TLS connection** - used to actually transmit the cryptographically protected data. For the data to be cryptographically protected, some parameters, such as the secret keys used to encrypt and authenticate the transmitted data need to be established; this is done when a **TLS session** is created, during the **TLS Handshake Protocol**.

In the handshake phase the client and the server agree on which version of the TLS protocol to use, authenticate one to another and negotiate session state items like the cipher suite and the compression method. Figure 3 shows the message flow for the full TLS 1.2 handshake. `*` indicates situation-dependent messages that are not always sent. `ChangeCiperSpec` is a separate protocol, rather than a message type.

As already mentioned, every TLS handshake message is encapsulated within a TLS record. The actual handshake message is contained within the `fragment` of a TLS record. The record type for a handshake message is `0x16`. The handshake message has the following structure: a 1-byte `msg_type` field (specifies the Handshake message type), a 2-byte `length` field (specifies the length of the `body`) and a `body` field, which contains a structure depending on the `msg_type` (similar to `fragment` field in a TLS record).

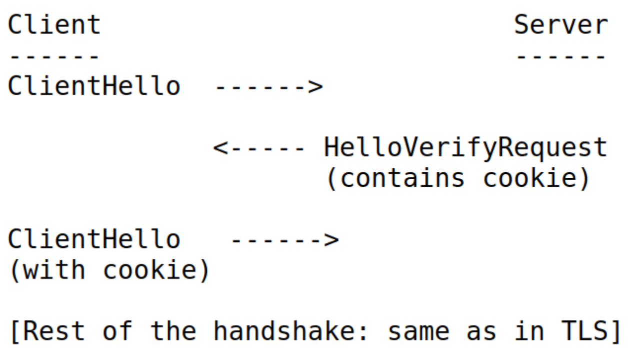


```
      Client                                               Server

      ClientHello                  -------->
                                                      ServerHello
                                                     Certificate*
                                               ServerKeyExchange*(1)
                                              CertificateRequest*
                                   <--------      ServerHelloDone
      Certificate*
   (1)ClientKeyExchange
   (2)CertificateVerify*
      [ChangeCipherSpec]
   (3)Finished                     -------->
                                               [ChangeCipherSpec]
                                   <--------             Finished(3)
      Application Data             <------->     Application Data

(1) - Server to client authentication (depending on the key
exchange method).
(2) - Client to server authentication
(3) - MITM protection guarantee
```

Fig. 3: TLS 1.2 message flow for a full handshake



```
Client                         Server
------                         ------
ClientHello  ------>

             <----- HelloVerifyRequest
                    (contains cookie)

ClientHello   ------>
(with cookie)

[Rest of the handshake: same as in TLS]
```

Fig. 4: DTLS handshake with HelloVerifyRequest containing the cookie

A typical handshake message flow will be described next, with only the most important fields of each message mentioned.

The TLS handshake starts with the client sending a `ClientHello`, containing `random`, `cipher_suites` and `compressison_methods`, among other fields.
`cipher_suites` contains a **list** of cipher suites and `compression_methods` contains a **list** of compression methods that the client supports, **ordered by preference**, with the most preferred one appearing first. The TLS record contains a 2-byte `version` field which indicates the highest version supported by the client.

The server responds to the `ClientHello` with a `ServeHello`. This message is similar, but contains the chosen `cipher_suite` and `compression_method` from the list sent by the client. Just like in the client's case, a `random` is present. The `version` field in the TLS record indicates the TLS version chosen by the server, which will be the one used for that connection.

TLS requires cryptographically secure pseudorandom numbers to be generated by both of the parties independently. Those random numbers (or *nonces*) are essential for freshness (protection against replay attacks) and session uniqueness. To provide those properties, both of the random values are required. Those two random values are inputs to the PRF when the master secret is generated, meaning that a new keying material will be obtained with every new session. If the output of the pseudorandom number generator can be predicted by the attacker, he can predict the keying material, as described in "A Systematic Analysis of the Juniper Dual EC Incident"[22]. The `32-byte` random value is composed by concatenating the `4-byte` GMT UNIX time with `28` cryptographically random bytes. Note that, in TLS 1.3, the random number structure has the same length, but is generated in a different manner: the client's `32 bytes` are all random, while the server's last `8 bytes` are fixed when negotiating TLS 1.2 or 1.3.

Next, the server sends a `Certificate` message, which contains a list of public key certificates: the server's certificate, every intermediate certificate and the root certificate, *i.e*, a certificate chain. The certificate's contents will depend on the negotiated cipher suite and extensions. The same message type occurs later in the handshake, if the server requests the client's certificate with the `CertificateRequest` message. In a typical scenario, the server will not request client authentication.

The `ServerKeyExchange` message follows, containing additional information needed by the client to compute the premaster secret. This message is only sent in some key exchange methods, namely `DHE_DSS`, `DHE_RSA` and `DH_anon`. For non-anonymous key exchanges, this is the message that authenticates the server to the client, since the server sends a digital signature over the client and server randoms, as well as the server's key exchange parameters. Note that this is not the only place where the server can authenticate itself to the client. For example, if `RSA` key exchange is used, the server authentication is done indirectly when the client sends the premaster secret encrypted with the public RSA key provided in the server certificate. Since only the server knows the corresponding private key, if both of the sides generate the same keying material, then the server must be who it claims to be. In TLS 1.3 this message is non-existent and a similar functionality is taken by the `key_exchange` extension.

The `ServerHelloDone` is sent to indicate the end of `ServerHello` and associated messages. Upon the receipt of this message, the client should check if the server provided a valid certificate. This message is not present in TLS 1.3.

With the ClientKeyExchange message the premaster secret is set. This is done either by direct transmission of the secret generated by the client and encrypted with the server's public RSA key (thus, authenticating the server to the client) or by the transmission of Diffie-Hellman (DH) parameters that will allow each side to generate the same premaster secret independently. In TLS 1.3 this message is non-existent and a similar functionality is taken by the key_exchange extension.

The CertificateVerify message is sent by the client to verify its certificate. This message is only sent if client authentication is used and if the client's certificate has signing capability (*i.e.* all certificates except for the ones containing fixed DH parameters).

The ChangeCipherSpec is its own protocol, rather than a type of handshake message. It is sent by both parties to notify the receiver that subsequent records will be protected under the newly negotiated cipher spec and keys. This message is not present in TLS 1.3.

The Finished message is an essential part of the protocol. It is the first message protected with the newly negotiated algorithms, keys and secrets. Only after both parties have sent and verified the contents of this message they can be sure that the Handshake has not been tampered with by a Man In The Middle (MITM) and begin to receive and send application data. Essentially, this message contains a keyed hash with the master secret over the hash of all the data from all of the handshake messages not including any HelloRequest messages and up to, but not including, this message. The other party must perform the same computation on its side and make sure that the result is identical to the contents of the other party's Finished message. If at some point a MITM has tampered with the handshake, there will be a mismatch between the computed and the received contents of the Finished message.

At any time after a session has been negotiated, the server may send a HelloRequest message, to which the client should respond with a ClientHello, thus beginning the negotiation process anew.

At any point in the handshake, the Alert protocol may be used by any of the peers to signal any problems or even abort the process through the use of an appropriate message type.

Besides the full handshake, TLS 1.2 also defines an abbreviated handshake mechanism, which can be used to either resume a previous session, or duplicate one, instead of negotiating new security parameters. This requires state to be maintained by both peers. The advantage of this mechanism is that the handshake is reduced to 1 RTT, instead of the usual 2 RTT, as it is the case in the full handshake.

In order to perform an abbreviated handshake, the client and the server must have established a session previously, by the means of a full handshake. In its ServerHello phase, the server generates and sends a session_id, which will be associated with the newly negotiated session.

To resume a session, in its ClientHello phase the client includes the session_id of the session it wants to resume. It is up to the server to decide if it will resume that session. In the positive case, the server responds with a ServerHello containing the same session_id value as the one sent by the client. In the negative case, the ServerHello will contain a different session_id value, thus triggering a new session negotiation process.

The keying material, such as the bulk data symmetric encryption keys and the MAC keys are formed by hashing the new client and server random values with the master secret. Therefore, provided that the master secret has not been compromised and that the secure hash operations are, in fact, secure, the new connection will be secure and independent from previous ones. The TLS 1.2 spec, suggests and upper limit of 24 hours for `session ID` lifetimes, since an attacker which obtains the master secret may be able to impersonate the compromised party until the corresponding `session ID` is retired.

3.3 TLS Record Processing

A TLS record must go through some processing before it can be sent over the network. This processing is done by the **TLS Record Protocol** and involves the following steps (`1-4` for TLS 1.2 and `1, 3-4` for TLS 1.3):

1. **Fragmentation** - the **TLS Record Layer** takes arbitrary-length data and **fragments** it into manageable pieces: each one of the resulting fragments is called a `TLSPlaintext`. Client message boundaries are not preserved, which means that multiple messages of the same type may be placed into the same fragment or a single message may be fragmented across several records.
2. **Compression** (removed in TLS 1.3) - the **TLS Record Layer** compresses the `TLSPlaintext` structure according to the negotiated compression method, outputting `TLSCompressed`. Compression is optional. If the negotiated compression method is `null`, `TLSCompressed` is identical to `TLSPlaintext`.
3. **Cryptographic Protection** - in TLS 1.2, either an AEAD cipher or a separate encryption and MAC functions transform a `TLSCompressed` fragment into a `TLSCipherText` fragment. In the case of TLS 1.3, the `TLSPlaintext` fragment is transformed into a `TLSCipherText` by applying an AEAD cipher, since all non-AEAD ciphers have been removed.
4. Append the `TLS Record Header` - encapsulate `TLSCipherText` in a `TLS Record`.

The process described above, as well as the structure names are depicted in Figure 5. The compression step is not present in TLS 1.3. The structure names are exactly as the appear in the TLS specifications.

3.4 TLS Keying Material

In TLS, the confidentiality and integrity guarantees are achieved through the use of SC. Consequently, the communicating peers need to **share** a **set of keys**. In TLS they are derived independently by the client and the server, during the TLS Handshake Protocol.

The keys appear with different names in TLS 1.2 and 1.3 specs, but they serve the same purpose. Additionally, more more keys can be found in TLS 1.3, for reasons that will be covered in Section 3.8. In TLS 1.2, the peers agree on the following set of keys:

- `client write key` - used by the client to encrypt the data to be sent
- `client read_key` - used by the client to decrypt the incoming data from the server
- `server write key` - used by the server to encrypt the data to be sent
- `server read key` - used by the server to decrypt the incoming data from the server
- `client write IV` - used by the client for implicit nonce techniques with AEAD ciphers
- `server_write_IV` - used by the server for implicit nonce techniques with AEAD ciphers

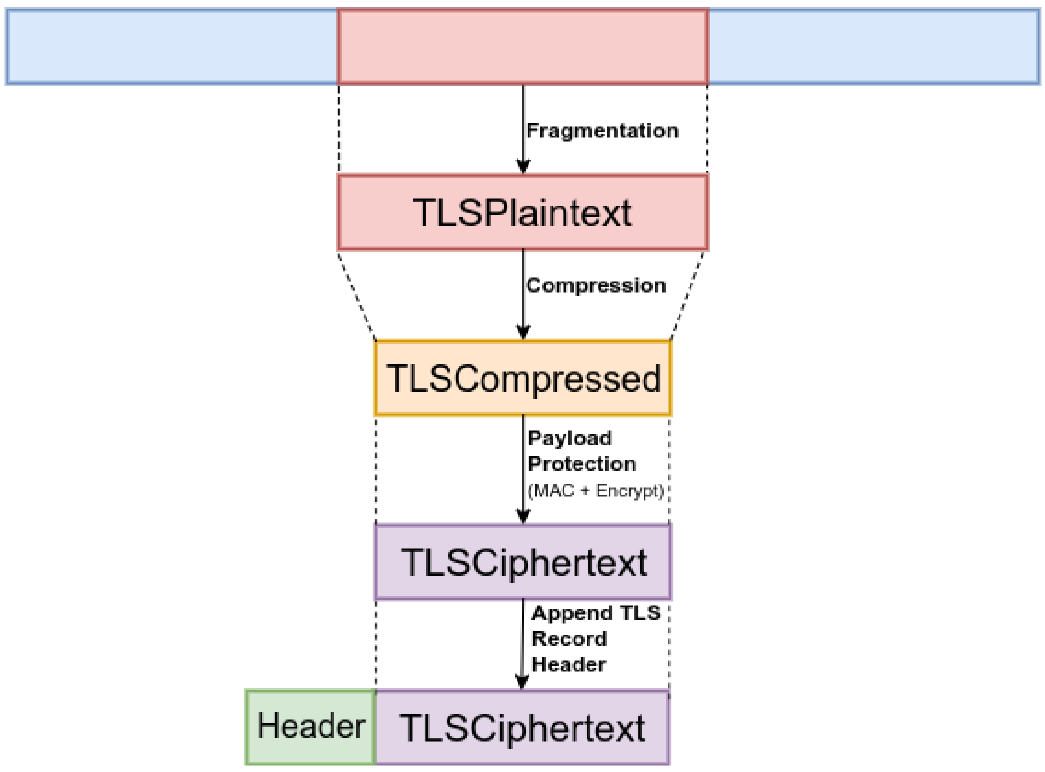


Fig. 5: TLS 1.2 Record Processing

- `client write MAC key` (TLS 1.2 only) - used by the client to authenticate the data to be sent
- `client write MAC key` (TLS 1.2 only) - used by the client to authenticate the data to be sent

When communicating with one another, the client uses one key to encrypt the data that it sends to the server and another key, different from the first one, to decrypt the data that it receives from the server, and vice-versa. This implies that the following relationships must hold: `client write key == server read key` and `server write key == client read key`.

3.5 TLS 1.2 Keying Material Generation

The generation of secret keys, used for various cryptographic operations involves the following steps, in order:

1. Generate the **premaster secret**.
2. From the **premaster secret** generate the **master secret**.
3. From the **master secret** generate the various secret keys, which will be used in the cryptographic operations.

The derivation of the keying material needed for a connection is done using the TLS PRF. It is defined as `PRF(secret, label, seed) = P_hash(secret, label + seed)`. The `P_hash(secret, seed)` function is an auxiliary data expansion function which uses a single cryptographic hash function to expand a `secret` and a `seed` into an arbitrary quantity of output. Therefore, it can be used to generate anywhere from 1 to an infinite number of bits of output. `PRF(secret, label, seed)` is used to generate as many bits of output as needed. When generating the master secret, the `secret` input is the `premaster secret`. When generating the key block, from which the final keys will be obtained, the `secret` input is the `master secret`.

The cryptographic hash function used in `P_hash(secret, label, seed)` is the hash function that is implicitly defined by the cipher suite in use. All of the cipher suites defined in the TLS 1.2 base spec use `SHA-256` and any new cipher suites must explicitly specify a the same hash function or a stronger one.

3.6 TLS 1.2 Key Exchange Methods

The way the peremaster secret is generated depends on the key exchange method used. This is the only phase of the keying material generation phase that is variable for a fixed cipher suite, since a cipher suite defines the PRF function that will be employed. Neither the derivation of the shared keys are impacted by the key exchange method.

There are many key exchange methods to choose from. Some of them are defined in the base spec (`RFC5246`[19]), while others in separate Request For Comment (RFC)s. For example, the ECC based key exchange, specified in `RFC4492` [23]).

The base spec specifies four key exchange methods, one using RSA and three using DH:

- static RSA (RSA; removed in TLS 1.3) - the client generates the premaster secret, encrypts it with the server's public key (which it obtained from the server's X.509 certificate) and sends it to the server. The server then decrypts it using the corresponding private key and uses it as its premaster secret. PFS is a property that preserves the confidentiality of past interactions even if the long-term secret is compromised. This key exchange method offers authenticity, but does not offer PFS.
- anonymous DH (DH_annon; removed in TLS 1.3) - each run of the protocol, uses different pubic DH parameters, which are generated dynamically. This results in a different, **ephemeral** key being generated every time. Since the exchanged DH parameters are **not authenticated**, the resulting key exchange vulnerable to MITM attacks. TLS 1.2 spec states that cipher suites using DH_annon **must not** be used, unless the application layer explicitly requests so. This key exchange offers PFS, but does not offer authenticity.
- fixed/static DH (DH; removed in TLS 1.3) - the server's/client's public DH parameter is embedded in its certificate. This key exchange method offers authenticity, but does not offer PFS.
- ephemeral DH (DHE) - the DH protocol is used, identically to DH_annon, but the public parameters are digitally signed in some way, usually using the sender's private RSA (DHE_RSA) or Digital Signature Algorithm (DSA) (DHE_DSS) key. This key exchange offers both, authenticity and PFS.

When either of the DH variants is used, the value obtained from the exchange is used as the premaster secret. Usually, only the server's authenticity is desired, but client's can also be achieved if it supplies the server with its certificate. Whenever the server is authenticated, it is secure against MITM attacks. Table 1 summarizes the security properties offered by each key exchange method.

Table 1: Key exchange methods and security properties

Key Exch Meth	Authentication	PFS
RSA	X	
DH_anon		X
DH	X	
DHE	X	X

In TLS 1.3, static RSA and DH ciphersuites have been removed, meaning that all public key exchange mechanisms now provide PFS. Even though anonymous DH key exchange has been removed, unauthenticated connections are still possible, by either using raw public keys[24] or not verifying the certificate chain and any of its contents.

The use of ECC-based key exchange (Elliptic Curve Diffie-Hellman (ECDH) and Elliptic Curve Diffie-Hellman Ephemeral (ECDHE)) and authentication (ECDSA) algorithms with TLS is described in RFC4492[23]. The document introduces five new ECC-based key exchange algorithms, all of which use ECC to compute the premaster secret, differing only in whether the negotiated keys are ephemeral (ECDH) or long-term (ECDHE), as well as the mechanism (if any) used to authenticate them. Three new ECDSA **client authentication** mechanisms are also defined, differing in the algorithms that the certificate must be signed with, as well as the key exchange algorithms that they can be used with. Those features are negotiated through TLS extensions.

3.7 TLS Extensions

TLS extensions were originally defined in `RFC 4366`[25] and later merged into the TLS 1.2 base spec. Each extension consists of an extension type, which identifies the particular extension type, and extension data, which contains information specific to a particular extension.

The extension mechanism can be used by TLS clients and servers; it is backwards compatible, which means that the communication is possible between a TLS client that supports a particular extension and a server that does not support it, and vice versa. A client may request the use of extensions by sending an extended `ClientHello` message, which is just a normal `ClientHello` with an additional block of data that contains a list of extensions. The backwards compatibility is achieved based on the TLS requirement that the servers that are not extensions-aware must ignore the data added to the `ClientHello`s that they do not understand (section `7.4.1.2` of `RFC 2246`[26]). Consequently, even servers running older TLS versions that do not support extensions, will not break.

The presence of extensions can be determined by checking if there are bytes following the `compression_methods` field in the `ClientHello`. If the server understands an extension, it sends back an extended `ServerHello`, instead of a regular one. An extended `ServerHello` is a regular `ServerHello` with an additional block of data following the `compression_method`, containing a list of extensions.

An extended `ServerHello` message can only be sent in a response to an extended `ClientHello` message. This prevents the possibility that an extended `ServerHello` message could cause a malfunction of older TLS clients that do not support extensions. An extension type must not appear in the extended `ServerHello`, unless the same extension type appeared in the corresponding extended `CleintHello`, and if this happens, the client must abort the handshake.

3.8 TLS 1.3

Due to limited space, TLS 1.3[27] will not be described in detail. The focus was on TLS 1.2 instead, because at the time the work on the thesis started, TLS 1.3 is still in draft mode and 1.2 was the latest and the recommended to use version.

Numerous differences from TLS 1.3 to 1.2 have been mentioned throughout the document. Various characteristics found in TLS 1.3 make it more suitable for the context of IoT than TLS 1.2. Some of them were already mentioned previously, and in this section a additional ones will be outlined.

The first important difference is that the use of extensions is required in TLS 1.3. This can be explained by the fact that some of the functionality has been moved into extensions, in order to preserve backwards-compatibility with the `ClientHello`s of the previous versions. The way a server distinguishes if a client is requesting TLS 1.3 is by checking the presence of the `supported_versions` extension in the extended `ClientHello`.

In TLS 1.3 more data is encrypted and the encryption begins earlier. For example, at the server-side there is a notion of ”encrypted extensions”. The `EncryptedExtensions` message, as the name suggests, contains a list of extensions that are encrypted under a symmetric key. It contains any extensions that are not needed for the establishment of the cryptographic context.

One of the main problems with using TLS in IoT is that while IoT traffic needs to be quick and lightweight, TLS 1.2 adds two additional round trips (`2 RTT`) to the start of every session. TLS 1.3 handshake has a lower latency, and this is extremely important in the context of IoT. The full TLS 1.3 handshake is only `1 RTT`. TLS 1.3 even allows clients to send data on the first flight (known as **early client data**), when the clients and servers share a PSK (either obtained externally or via a previous handshake). This means that in TLS 1.3 `0-RTT` data is possible, by encrypting it with a key derived from a PSK. Session resumption via identifiers and tickets has been obsoleted in TLS 1.3, and both methods have been replaced by a PSK mode. This PSK is established in a previous connection after the handshake is completed and can be presented by the client on the next visit.

Keying material generation is more complex in TLS 1.3 than in TLS 1.2, since different keys are used to encrypt data throughout the Handshake protocol. This can be explained by the fact that in TLS 1.3 the encryption begins earlier. Other Handshake messages besides `Finished` are encrypted. As a result, multiple encryption keys are generated and used to encrypt different data throughout the handshake.

The way the keying material is derived is also different. The PRF construction described above has been replaced. In TLS 1.3, key derivation uses the HKDF function [28] and its two components: `HKDF-Extract` and `HKDF-Expand`. This new design allows easier analysis by cryptographers due to improved key separation properties.

3.9 DTLS

As already mentioned, DTLS is an adaption of TLS that runs on top of an unreliable transport protocol, such as UDP. The design of DTLS is deliberately very similar to TLS, in fact, its specification is written in terms of differences from TLS. This similarity allows to both, minimize new security invention, and maximize the amount of code and infrastructure reuse. The changes are mostly done at the lower level and don not affect the core of the protocol. Even extensions defined before DTLS existed can be used with it. The latest version of DTLS is 1.2 and it is defined in `RFC 6347`[29]. There is a draft of DTLS 1.3 [30] that is currently under active development.

Since DTLS operates on top of an unreliable transport protocol, such as UDP, it must explicitly deal with the absence of reliable and ordered assumptions that are made by TLS. The main differences from DTLS 1.2 to TLS 1.2 are:

– two new fields are added to the record layer: an explicit `2 byte` sequence number and a `6 byte` epoch. The DTLS MAC is the same as in TLS, however, rather than using the implicit sequence number, the `8 byte` value formed by concatenation of the epoch number and the sequence number is used.
– stream ciphers must not be used with DTLS.
– a stateless cookie exchange mechanism has been added to the handshake protocol in order to prevent Denial-of-Service (DoS) attacks. To accomplish this, a new handshake message, the `HelloVerifyRequest` has been added. After the `ClientHello`, the server responds with a `HelloVerifyRequest` containing a cookie, which is returned back to the server in another `ClientHello` that follows it. After this, the handshake proceeds

as in TLS. This is depicted in Figure 4. Although optional for the server, this mechanism highly recommended, and the client must be prepared to respond to it. DTLS 1.3 follows the same idea, but does it differently, namely, the `HelloVerifyRequest` message has been removed, and the cookie is conveyed to the client via an extension in a `HelloRetryRequest` message.
- the handshake message format has been extended to deal with message reordering, fragmentation and loss by addition of three new fields: a message sequence field, a fragment offset field and a fragment length field.

4 Related Work

Lightweight cryptography is an important topic in the context of IoT security, due to the resource-limited nature of the devices. This section will begin with the description of the work done in this area.

Biryukov *et al*[31] explore the topic of lightweight symmetric cryptography, providing a summary of the lightweight symmetric primitives from the academic community, the government agencies and even proprietary algorithms which have been either reverse-engineered or leaked. All of those algorithms are listed in the paper, alongside relevant metrics. The list will not be included herein due to the lack of space. The authors also proposed to split the field into two areas: ultra-lightweight and IoT cryptography.

The paper systematizes the knowledge in the area oe lightweight cryptography in order to define "lightweightness" more precisely. The authors observed that the design of lightweight cryptography algorithms varies greatly, the only unifying thread between them being the low computing power of the devices that they are designed for.

The most frequently optimized metrics are the memory consumption, the implementation size and the speed or the throughput of the primitive. The specifics depend on whether the hardware or the software implementations of the primitives are considered.

If the primitive is implemented in hardware, the memory consumption and the implementation size are lumped together into its gate area, which is measured in Gate Equivalents (GE), a metric quantifying how physically large a circuit implementing the primitive is. The throughout is measured in *bytes/sec* and it corresponds to the amount of plaintext processed per time unit. If a primitive is implemented in software (typically for use in micro-controllers), the relevant metrics are the RAM consumption, the code size and the throughput of the primitive, measured in *bytes/CPU cycle*.

To accommodate the limitations of the constrained devices, most lightweight algorithms are designed to use smaller internal states with smaller key sizes. After analysis, the authors concluded that even though at least `128 bit` block and key sizes were required from the AES candidates, most of the lightweight block ciphers used only `64-bit` blocks, which leads to a smaller memory footprint in both, software and hardware, while also making the algorithm better suited for processing of smaller messages.

Even though algorithms can be optimized in implementation: whether it is a software or a hardware, dedicated lightweight algorithms are still needed. This comes down mainly to two factors: there are limitations to the the extent of the optimizations that can be done and the hardware-accelerated encryption is frequently vulnerable to various Side-Channel Attack (SCA)s. An example of such an attack is the one done on the Phillips light bulbs [32], where the authors were able to recover a secret key used to authenticate updates.

It is more difficult to implement a lightweight hash function than a lightweight block cipher, since standard hash functions need large amounts of memory to store both: their internal states, for example, `1600 bits` in case of SHA-3, and the block they are operating on, for example, `512 bits` in the case of SHA-2. The required internal state is acceptable for a desktop computer, but not for a constrained device. Taking this into consideration, the most common approach taken by the designers is to use a sponge construction with a very small bitrate. A sponge function is an algorithm with an internal state that takes as an input a bit stream of any length and outputs a bit stream of any desired length. Sponge functions are used to implement many cryptographic primitives, such as cryptographic hashes. The bitrate decides how fast the plain text is processed and how fast the final digest is produced. A smaller bitrate means that the output will take longer to be produced, which means that a smaller capacity (the security level) can be used, which minimizes the memory footprint at the cost of slower data processing. A capacity of `128 bits` and a bitrate of `8 bits` are common values for lightweight hash functions.

Another trend in the lightweight algorithms noticed by the authors is the preference for *ARX*-based and *bitsliced-S-Box* based designs, as well as simple key schedules.

Finally, a separation of the "lightweight algorithm" definition into two distinct fields has been proposed:

– **Ultra-Lightweight Crypto** - algorithms running on very cheap devices **not connected to the internet**, which are easily replaceable and have a limited life-time. Examples: *RFID* tags, smart cards and remote car keys.
– **IoT Crypto** - algorithms running on a low-power device, **connected to a global network**, such as the internet. Examples: security cameras, smart light bulbs and smart watches.

Considering the two definitions above, this the work of this dissertation focuses on **IoT Crypto** devices. A summary of differences between the both categories is summarized in table 2.

Table 2: A summary of the differences between ultra-lightweight and IoT crypto

	Ultra-Lightweight	**IoT**
Block Size	64 bits	$\geq$ 128 bits
Security Level	$\geq$ 80 bits	$\geq$ 128 bits
Relevant Attacks	low data/time complexity	same as "regular" crypto
Intended Platform	dedicated circuit (ASIC, RFID...)	micro-controllers, low-end CPUs
SCA Resilience	important	important
Functionality	one per device, e.g. authentication	encryption, authentication, hashing...
Connection	temporary, only to a given hub	permanent, to a global network

While there is a high demand for lightweight public key primitives, the required resources for them are much higher than for symmetric ones. As a paper by Katagi *et al*[33] concluded, there are no promising primitives that have enough lightweight and security properties, compared to the conventional ones, such as RSA and ECC. Further research on this topic, as part of the work on this dissertation, lead to the same conclusion.

Lightweight cryptography is an important topic this work and there are papers detailing various algorithms. In order to provide a good overview of it while staying succinct, recent papers that provide a summary of the area, rather than focusing on specific implementations, were chosen. The remainder of this section will focus on the work done on the (D)TLS protocol in the context of IoT.

The "Scalable Security With Symmetric Keys"[11] paper proposes a key management architecture for resource-constrained devices, which allows devices that have no previous, direct security relation to use (D)TLS using one of two approaches: shared symmetric keys or raw public keys. The resource-constrained device is a server that offers one or more resources, such as temperature readings. The idea in both approaches is to introduce a third-party `trust anchor (TA)` that both, the client and the server use to establish trust relationships between them.

The first approach is similar to Kerberos[34], and it does not require any changes to the original protocol. A client can request a PSK `Kc` from the `TA`, which will generate it and send it back to the client via a secure channel, alongside a `psk_identity` which has the same meaning and use as in `RFC 4279`[35]. When connecting to the server, the client will send to the server the `psk_identity` that it received in a previous handshake. Upon its receipt, the server will derive the `Kc`, using the `P_hash()` function defined in `RFC 5246`[19].

The second approach consists in requesting an Authorized Public Key (APK) from the `TA`. The client includes his Raw Public Key (RPK) in its request, which is used for authorization. The TA creates an authorization certificate, protects it with a MAC and sends it to the client alongside the server's public key. The client then sends this APK (instead of the RPK) when connecting to the server, which verifies it (to authorize the client) and proceeds with the handshake in the RPK mode, as defined in `RFC 4279` [35]. To achieve this, a new certificate structure is defined, alongside a new `certificate_type`. The new certificate structure is just the `RFC7250` [24] structure, with an additional MAC.

The hash function used for key derivation is SHA256. The authors evaluated the performance of their solution with and without SHA2 hardware acceleration and concluded that while it had significant impact on key derivation, it had little impact on the total handshake time (`711.11 ms` instead of `775.05 ms`), since most of the time was spent in sending data over the network and other parts of the handshake, the longest one being the `ChangeCipherSpec` message which required a processing time of `17.79ms`.

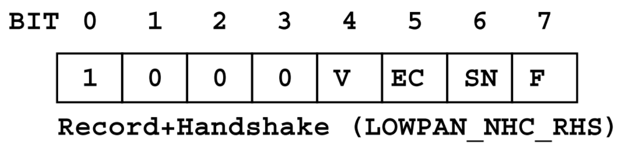
6LoWPAN[36] is a protocol that allows devices with limited processing ability and power to transmit information wirelessly using the `IPv6` protocol. The protocol defines IP Header Compression (IPHC) for the IP header, as well as, Next Header Compression (NHC) for the IP extension headers and the UDP header in `RFC 6282`[37]. The compression relies on the shared context between the communicating peers.

The work proposed in [38] uses this same idea, but with the goal of compressing DTLS headers. 6LoWPAN does not provide ways to compress the UDP payload and layers above. A proposed standard[39] for generic header compression for 6LoWPANs that can be used to compress the UDP payload, does exit, however. The authors propose a way to compress DTLS headers and messages using this mechanism.

Their work defines how the DTLS Record header, the DTLS Handshake header, the `ClientHello` and the `ServerHello` messages can be compressed, but notes that the same compression techniques can be used to compress the remaining handshake messages. They explore two cases for the header compression: compressing both, the Record header and the Handshake header and compressing the Record header only, which is useful after the handshake has completed and the fragment field of the Record layer contains application data, instead of a handshake message.

```
+-------------+------------+------------+----------------+--------
| LOWPAN_IPHC | In-line    | LOWPAN_NHC | In-line Next   | Payload
|   Encoding  |  IP Fields |  Encoding  |  Header Fields |
+-------------+------------+------------+----------------+--------
```
itself

Fig. 6: IPv6 Next Header Compression



e itself

Fig. 7: LOWPAN_NHC_RHS structure

Each DTLS fragment is carried over as a UDP payload. In this case, the UDP payload carries a header-like payload (the DTLS record header). Figure 6 shows the way IPv6 next header compression is done. The authors use the same value for the `LOWPAN_NHC Encoding` field (defined in `RFC 6282`[37]) as in `RFC7400` and define the format of the `In-line Next Header Fields` (also defined in [37]), which is the compressed DTLS content. The `LOWPAN_IPHC Encoding` and `In-Line IP Fields` fields are used in the IPv6 header compression and are not in the scope of the paper.

All of the cases follow the same basic idea, for this reason only one of them will be exemplified: the case where both, the Record and the Handshake headers are compressed. In this case `LOWPAN_NHC Encoding` will contain the `LOWPAN_NHC_RHS` structure (depicted in figure 7), which is the compressed form of the Record and Handshake headers. The parts that are not compressed will be contained in the `Payload` part. The first four bits represent the ID field and in this case they are fixed to `1000`, that way, the decompressor knows what is being compressed (*i.e* how to interpret the structure that follows the ID bits). If the `F` field of the `LOWPAN_NHC_RHS` structure contains the bit `0`, it means that the handshake message is not fragmented, so the `fragment_offset` and `fragment_length` fields are elided from the Handshake header (common case when a handshake message is not bigger than the maximum header size), meaning that they are not going to be sent at all (*i.e.* they are not going to be present in the `Payload` part). If the `F` bit has the value `1`, the `fragment_offset` and `fragment_length` fields are carried inline (*i.e.* they are present in the `Payload` part). The remaining two fields define similar behavior for other header fields (some of them assume that some default value is present, when a field is elided). The `length` field in the Record and Handshake headers are always elided, since they can be inferred from the lower layers.

The evaluation showed that the compression can save a significant number of bits: the Record header, that is included in all messages can be compressed by `64 bits` (*i.e.* by 62%).

There is also a proposal for TCP header compression for 6LoWPAN[40], which if adopted, in many cases can compress the mandatory `20 bytes` TCP header into `6 bytes`. This means that the same ideas can be applied to TCP and TLS as well.

Later, in 2013, Raza *et al.* proposed a security scheme called Lithe[41], which is a lightweight security solution for CoAP that uses the same DTLS header compression technique as in [38] with the goal of implementing it as a security support for CoAP. CoAP[42] is a specialized *REST*ful Internet Application Protocol for constrained devices. it is designed to easily translate to HTTP, in order to simplify its integration with the web, while also meeting requirements such as multicast support and low overhead. CoAP is like "HTTP for constrained devices". It can run on most devices that support UDP or a UDP-like protocol. CoAP mandates the use of DTLS as the underlying security protocol for authenticated and confidential communication. There is also a CoAP specification running on top of TCP, which uses TLS as its underlying security protocol currently being developed[12].

The authors evaluated their system in a simulated environment in *Contiki OS*[43], which is an open-source operating system for the IoT. They obtained significant gains in terms of packet size (similar numbers to the ones observed in [38]), energy consumption (on average 15% less energy is used to transmit and receive compressed packets), processing time (the compression and decompression time of DTLS headers is almost negligible) and network-wide response times (up to 50% smaller RTT). The gains in the mentioned measures are the largest when the compression avoids fragmentation (in the paper, for payload size of `48 bytes`).

Angelo *et al.* [44] proposed to integrate the DTLS protocol inside CoAP, while also exploiting ECC optimizations and minimizing ROM occupancy. They implemented their solution in an off-the-shelf mote platform and evaluated its performance. DTLS was designed to protect web application communication, as a result, it has a big overhead in IoT scenarios. Besides that, it runs over UDP, so additional mechanisms are needed to provide the reliability and ordering guarantee. With this in mind, the authors wanted to design a version of DTLS that both: minimizes the code size and the number of exchanged messages, resulting in an optimized Handshake protocol.

In order to minimize the code size occupied by the DTLS implementation, they decided to delegate the tasks of **reliability** and **fragmentation** to CoAP. This means that the code responsible for those functionalities, can be removed altogether from the DTLS implementation, thus reducing ROM occupancy. This part of their work was based on an informational RFC draft[45], in which the authors profiled DTLS for CoAP-based IoT applications and proposed the use of a *RESTful* DTLS handshake which relies on CoAP block-wise transfer to address the fragmentation issue.

To achieve this they proposed the use of a *RESTful* DTLS connection as a CoAP resource, which is created when a new secure session is requested. The authors exploit the the CoAPs capability to provide connection-oriented communication offered by its message layer. In particular, each `Confirmable` CoAP message requires an `Acknowledgement` message (page 8 of `RFC 7252` [46]), which acknowledges that a specific `Confirmable` message has arrived, thus providing reliable retransmission.

Instead of leaving the fragmentation function to DTLS, it was delegated to the block-wise transfer feature of CoAP[42], which was developed to support transmission of large payloads. This approach has two advantages: first, the code in the DTLS layer responsible for this function can be removed, thus reducing ROM occupancy, and second, the fragmentation/reassembly process burdens the lower layers with state that is better managed in the application layer.

The authors also optimized the implementation of basic operations on which many security protocols, such as ECDH and ECDSA rely upon. The first optimization had to do with modular arithmetic on large integers. A set of optimized assembly routines based on [47] allow the improved use of registers, reducing the number of memory operations needed to perform tasks such as multiplications and square roots on devices with `8-bit` registers.

Scalar multiplication is often the most expensive operation in Elliptic Curve (EC)-based cryptography, therefore optimizing it is of high interest. The authors used a technique called *IBPV* described in [48], which is based on pre-computation. of a set of discrete log pairs. The mathematical details have been purposefully omitted, since they are not relevant for this description. The *IBPV* technique was used to improve the performance of the ECDSA signature and extended to the ECDH protocol. In order to reduce the time taken to perform an ECDSA signature verification, the *Shamir Trick* was used, which allows to perform the sum of two scalar multiplications (frequent operation in EC cryptography) faster than performing two independent scalar multiplications.

The results showed that the ECC optimizations outperform the scalar multiplication in the state of the art class 1 device platforms, while also improving the the network lifetime by a factor of up to 6.5 with respect to a standard, non-optimized implementation. Leaving reliability and fragmentation tasks to CoAP, reduces the DTLS implementation code size by approximately 23%.

`RFC 7925`[49] describes a TLS and DTLS 1.2 profile for IoT devices that offer communication security services for IoT applications. In this context, "profile" means available configuration options (ex: which cipher suites to use) and protocol extensions that are best suited for IoT devices. The document is rather lengthy, only its fundamental parts will be summarized. A number of relevant RFCs will also be described.

`RFC 7925` explores both cases: constrained clients and constrained servers, specifying a profile for each one and describing the main challenges faced in each scenario. The profile specifications for constrained clients and servers are very similar. Code reuse in order to minimize the implementation size is recommended. For example, an IoT device using a network access solution based on TLS, such as EAP-TLS[50] can reuse most parts of the code for (D)DTLS at the application layer.

For the credential types the profile considers 3 cases:

– PSK - authentication based on PSKs is described in `RFC 4249`[35]. When using PSKs, the client indicates which key it wants to use by including a PSK identity in its `ClientKeyExchange` message. A server can have different PSK identities shared with different clients. An identity can have any size, up to a maximum of `128 bytes`. The profile recommends the use of shorter PSK identities and specifies `TLS_PSK_WITH_AES_128_CCM_8` as the only mandatory-to-implement cipher suite to be used with PSKs, just like CoAP does. If a PFS cipher suite is used, ephemeral DH keys should not be reused over multiple protocol exchanges.

- RPK - the use of RPKs in (D)TLS is described in `RFC 7250`[24]. With RPKs, only a subset of the information that is found in typical certificates is used: namely the `SubjectPublicKeyInfo` structure, which contains the necessary parameters to describe the public key (the algorithm identifier and the public key itself). Other PKIX certificate[51] parameters are omitted, making the resulted RPK smaller in size, when compared to the original certificate and the code to process the keys simpler. In order for the peers to negotiate a RPK, two new extensions have been defined: one for the client indicate which certificate types it can provide to the server, and one to indicate which certificate types it can process from the server. To further reduce the size of the implementation, the profile recommends the use of the TLS Cached Information extension[52], which enables TLS peers to exchange just the fingerprint (a shorter sequence of bytes used to identify a public key) of the public key. Identical to CoAP, the only mandatory-to-implement cipher suite to be used with RPKs is `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8`.
- certificate - conventional certificates can also be used. The support for the Cached Information extension[52] and the `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8` cipher suite is required. The profile restricts the use of named curves to the ones defined in `RFC 4492`[23]. For certificate revocation, neither the Online Certificate Status Protocol (OCSP)[53], nor the Certificate Revocation List (CRL)[51] mechanisms are used, instead this task is delegated to the software update functionality. The Cached Information extension does not provide any help with caching client certificates. For this reason, in cases where client-side certificates are used and the server is not constrained, the support for client certificate URLs is required. The client certificates URL extension[25] allows the clients to point the server to a URL from which it can obtain its certificate, which allows constrained clients to save memory and amount of transmitted data. The Trusted CA Indication[54] extension allows the clients to indicate which trust anchors they support, which is useful for constrained clients that due to memory limitation posses only a small number of CA root keys, since it can avoid repeated handshake failures. If the clients interacts with dynamically discovered set of (D)TLS servers, the use of this extension is recommended, if that set is fixed, it is not.

The signature algorithms extension[19] allows the client to indicate to the server which signature/hash pairs it supports to be used with digital signatures. The client must send this extension to indicate the use of `SHA-256`, otherwise the defaults defined in [19] are used. This extension is not applicable when PSK-based cipher suites are used.

The profile mandates that constrained clients must implement session resumption to improve the performance of the handshake since this will lead to less exchanged messages, lower computational overhead (since only symmetric cryptography is used) and it requires less bandwidth. If server is constrained, but the client is not, the client must implement the Session Resumption Without Server-Side State mechanism[21], which is achieved through the use of tickets. The server encapsulates the state into a ticket and forwards it to the client, which can subsequently resume the session by sending back that ticket. If both, the client and the server are constrained, both of them should implement `RFC 5077`[21].

The use of compression is not recommended for two reasons. First, `RFC7525`[55] recommends disabling (D)TLS level compression, due to attacks such as `CRIME`[56]. `RFC7525` provides recommendations for improving the security of deployed services that use TLS and DTLS and was published as a response to the various attacks on (D)TLS that have emerged over the years. Second, for IoT applications, the (D)TLS compression is not needed, since application-layer protocols are highly optimized and compression at the (D)TLS layer increases the implementation's size and complexity.

RFC6520[57] defines a heartbeat mechanism to test whether the peer is still alive. The implementation of this extension is recommended for server initiated messages. Note that since the messages sent to the client will most likely get blocked by middleboxes, the initial connection setup is initiated by the client and then kept alive by the server.

Random numbers play an essential role in the overall security of the protocol. Many of the usual sources of entropy, such as the timing of keystrokes and the mouse movements, will not be available on many IoT devices, which means that either alternative ones need to be found or dedicated hardware must be added. IoT devices using (D)TLS must be able to find entropy sources adequate for the generation of quality random numbers, the guidelines and requirements for which can be found in RFC4086[58].

Implementations compliant with the profile must use AEAD ciphers, therefore encryption and MAC computation are no longer independent steps, which means that neither encrypt-then-MAC[59], nor the truncated MAC[54] extensions are applicable to this specification and must not be used.

The Server Name Indication (SNI) extension[54] defines a mechanism for a client to tell a (D)TLS server the name of the server that it is contacting. This is crucial in case when multiple websites are hosted under the same IP address. The implementation of this extension is required, unless the (D)TLS client does not interact with a server in a hosting environment.

The maximum fragment length extension[54] lowers the maximum fragment length support of the record layer from 2^14 to 2^9. This extension allows the client to indicate the server how much of the incoming data it is able to buffer, allowing the client implementations to lower their RAM requirements, since it does not not need to accept packets of large size, such as the 16K packets required by plain (D)TLS. For that reason, client implementations must support this extension.

The Session Hash Extended Master Secret Extension[60] defines an extension that binds the master secret to the log of the full handshake, thus preventing MITM attacks, such as the triple handshake[61]. Even though the cipher suites recommended by the profile are not vulnerable to this attack, the implementation of this extension is advised. In order to prevent the renegotiation attack[62], the profile requires the TLS renegotiation feature to be disabled.

With regards to the key size recommendations, the authors recommend symmetric keys of at least 112 bit, which corresponds to a 233-bit ECC key and to a 2048 DH key. Those recommendations are made conservatively under the assumption that IoT devices have a long expected lifetime (10+ years) and that those key recommendations refer to the long-term keys used for device authentication. Keys that are provisioned dynamically and used for protection of transactional data, such as the ones used in (D)TLS cipher suites, may be shorter, depending on the sensitivity of transmitted data.

Even though TLS defines a single stream cipher: *RC4*, its use is no longer recommended due to its cryptographic weaknesses described in RFC 7465[63].

RFC 7925[49] points out that designing a software update mechanism into an IoT system is crucial to ensure that potential vulnerabilities can be fixed and that the functionality can be enhanced. The software update mechanism is important to change configuration information, such as trust anchors and other secret-key related information. Although the profile refers to LM2M[64] as an example of protocol that comes with a suitable software update mechanism, there has been new work done in this area since the release of this profile. There is a document specifying an architecture for a firmware update mechanism for IoT devices[65] currently in Internet-Draft state.

5 Methodology

In (D)TLS the key the authentication algorithm, the encryption algorithm, the data integrity algorithm, as well as the associated key sizes for each are all defined in a *ciphersuite*. A ciphersuite defines the security properties of a (D)TLS connection. For this reason, the terms *ciphersuite* and *configuration* will be used interchangeably.

A (D)TLS connection consists of two main phases:

1. The peers authenticate one to another, agree on the data encryption and integrity algorithms that they will use and establish the shared keys. This part is known as the handshake protocol.
2. The peers exchange the data securely, using the algorithms and keys negotiated in the previous phase. This part is known as the record protocol.

The relative cost of each phase depends on the chosen algorithms, as well as the amount of data transferred. For this reason, it is important to evaluate the costs of both.

(D)TLS has numerous possible configurations. Each one of those configurations is defined in an RFC. Each ciphersuite is assigned a unique identification number. Internet Assigned Numbers Authority (IANA) is responsible for maintaining the full list of them. At the moment of this writing, there are over 300 ciphersuites defined for (D)TLS [66].

mbedTLS implements a subset of those ciphersuites. As of version 2.7.0, *mbedTLS* has a total of 161 ciphersuites [67]. Manual cost evaluation and data analysis would greatly limit the scope of obtained results, as it would be very time consuming and error-prone. For this reason, we developed tools that would automate the profiling and collection of results.

In our work, we evaluated the *mbedTLS's* implementation of the TLS protocol, thus obtained metrics reflect the algorithm's implementations used within the library. We focused our analysis on two metrics: an estimated numbers of CPU cycles executed and time taken. The obtained time metrics were not estimated, but rather values read directly from the processor's registers. As we will show in later sections, the estimates reflect the real values.

There are well-established, industry-standard tools for collecting and analyzing the estimated computational costs, such as the number of CPU cycles. Two of such tools are *valgrind* and *callgrind* and it's the ones that we used. Relying on such tooling allowed us to analyze the costs of TLS at both, the hiher and lower levels. First, *valgrind* allowed us to collect cost metrics for every part of TLS. Then, *callgrind* allowed us to visualize those costs and map the architecture of the codebase to relevant parts of TLS (*i.e.* which functions in the code correspond to which TLS operations).

Similarly, tools for reading performance related hardware counters directly from the processor exist. One of such tools is Performance Apppicaiton Programming Interface (PAPI) [68]. After analyzing the estimated costs, we instructed the relevant parts of the code with PAPI, in order to obtain cost metrics directy from the processor's registers. More specifically, we measured time. A comparison of the estimates and the real values showed that they are proportional one to another.

TLS. First we collected a wide array of metrics with *valgrind* and *callgrind*, which allowed to analyze the costs of TLS at both, high and low levels. After that, we studied the costs obtained by reading the relevant counters directly from the processor and showed that they are proportional to the estimated ones.

In the next section we will analyze the evaluated metrics in detail, as well as describe their limitations.

5.1 Evaluated Metrics and Limitations

In this section we will describe the metrics that we evaluted, the environment in which the evaluation was performed and the limitations of our approaches. We will present our descriptions in the same order that we performed them: first estimating performance measurments with *valgrind*, followed by collecting values directly from the processor's counters with *papi*.

Estimation with Valgrind and Callgrind In order to estimate the number of executed cycles, we used *valgrind*, more specifically its *callgrind* tool. *valgrind* runs the application on a synthetic CPU. While running the code in that synthetic environment, it is able to insert instructions to perform profiling and debugging.

In essence, *valgrind* is a virtual machine, using just-in-time (JIT) compilation techniques, such as dynamic recompilation. Dynamic recompilation is a feature where some part of the program is recompiled during execution.

The *valgrind* tool consists of two parts, the *valgrind core* and the *tool plugin*. The *valgrind core* transforms the machine code into a simpler form called Intermediate Representation (IR). The IR code is then passed to the *tool plugin*, which modifies the IR code as needed, typically by instrumenting it. This modified IR code is then passed back to the *valgrind core*, which transforms it back into machine code. That recompiled machine code will then run on the host CPU (the JIT compilation step). This process is illustrated in Figure 8.

In our case, the *tool plugin* is *callgrind*. Among other metrics, *callgrind* collects the number of executed instructions, L1/L2 caches misses (the caches are simulated), and branch prediction misses. The metrics collected by *callgrind* can then be loaded into *kcachegrind* to visualize and analyze the performance results. One of such results is the estimate of the number of executed CPU cycles. *callgrind* and *kcachegrind* are widely in conjunction for performance analysis and optimization of programs. In order to count the number of executed CPU cycles, we derived a formula from the one that is used by *kcachegrind*.

The number of executed cycles presented by *kcachegrind* is an estimate, which might not correspond to the real value. Although undocumented, we found the formula that estimates the number of executed CPU cycles in *kcachegrind*'s source code [69]. It uses the following formula: $CEst = Ir + 10 * Bm + 10 * L1m + 20 * Ge + 100 * L2m + 100 * LLm$, where

- $CEst$ - estimated CPU cycles

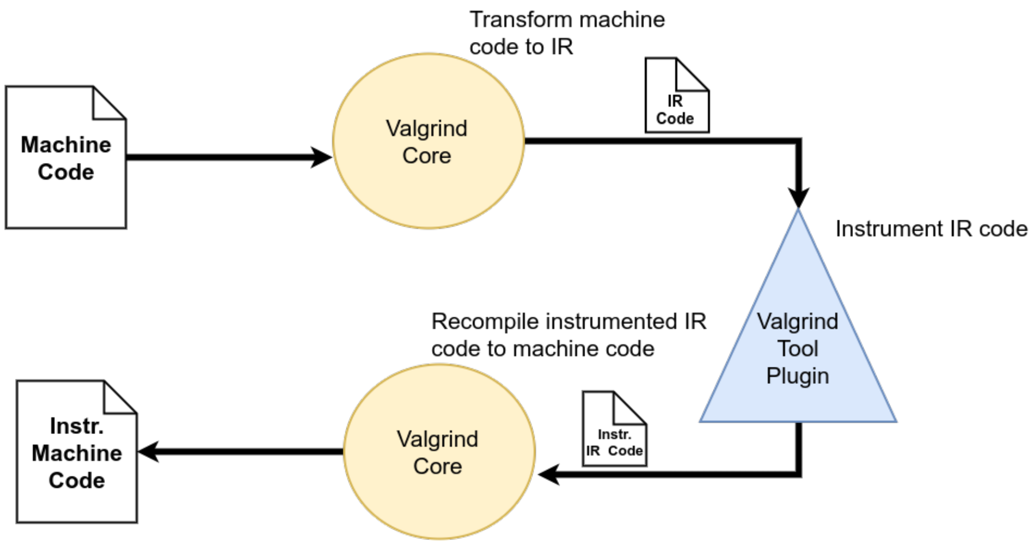


Fig. 8: How Valgrind works

- Ir - instruction fetches
- Bm - mis-predicted branches (direct and indirect)
- Ge - number of global bus events
- $L1m$ - total L1 cache misses (instruction fetch, data read and data write)
- $L2m$ - total L2 cache misses (instruction fetch, data read and data write)
- LLm - total last-level cache misses (instruction fetch, data read and data write)

Callgrind only simulates L1 and L2 caches, making $LLm = 0$, therefore the actual formula used by *KCachegrind* (when used with *Callgrind* output) is: $CEst = Ir + 10 * Bm + 10 * L1m + 20 * Ge + 100 * L2m$.

Ge is a useful metric when synchronization primitives are present, since it counts the number of atomic instructions executed. For example, on the x86 and x86_64 architectures, these are instructions using the *lock* prefix. In our evaluation, only single-threaded code was used, therefore reason we did not measure Ge. This further simplified the formula used by *kcachegrind* to $CEst = Ir + 10 * Bm + 10 * L1m + 100 * L2m$.

In order to estimate the number of CPU cycles, we developed tooling that parsed the metrics output by *callgrind* and input them into the *kcachegrind*'s formula. All of the evaluations were performed in our local machine, with its relevant characteristics shown in table 3.

Processor	Intel(R) Core(TM) i7-4700HQ CPU @ 2.40GHz
L1 Instruction Cache	32768 B, 64 B, 8-way associative
L1 Data Cache	32768 B, 64 B, 8-way associative
LL Cache	6291456 B, 64 B, 12-way associative

Table 3: Local machine's characteristics

Reading Values From Processor Counters with PAPI In order to obtain the actual processor time spent on various parts of TLS we used PAPI. Hardware designers added registers to the processors to measure various aspects of its function. Those registers act as performance counters that count specific signals related to processor's funtion, such as the number of instructions executed and total time elapsed. PAPI provides a standard interface for accessing those counters.

PAPI provides interfaces to measure both, virtual and processor time. *Real time*, also known as *wall clock time* is total elapsed time, including the time slices used by ohter processes, as well as the time spent while the process was blocked, while, for example, waiting for I/O operations to complete. *Virtual time*, also known as *user time*, is amount of time spent in user mode within the process (*i.e.* time spent in kernel mode is not included). This measure 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. For our purposes, only the *virtual time* was relevant and it was the one that we measured.

Just like with *valgrind*, all of our PAPI evaluations were perfomed on the machine described in table 3. However, in order to keep the metrics consistent, we disabled Intel Turbo Boost [70] and fixed the processor's speed to the lowest available frequency of $800Mhz$.

Limitations The metrics obtained with *valgrind/callgrind* are just estimated values, which do not necessarily reflect the real ones. For this reason, we complemented them with PAPI measures, which are read directly from the processor's registers.

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. In the next section we will describe which we tools developed in order to collect the metrics presented in this work. It is possible to use those tools in a local environment to automate metric collection.

5.2 Developed Tools

mbedTLS 2.7.0 provides numerous possible configurations for a TLS connection. It would be unfeasible to perform a thorough and precise evaluation of its TLS implementation manually. For this reason, we developed a set of tools to automate those tasks. In general, the developed tools served one of two purposes: metric collection and collected metric analysis, for both *valgrind* and PAPI. All of the developed tooling is open-source and available online.

In order to collect the relevant metrics, we developed automated tools that would run a client-server connection with a specified set of ciphersuites and save the collected metrics for later analysis. *callgrind* outputs files with the profiling information, so we used those files directly [71]. As for the PAPI results, we programmed a custom metrics collector that would group the metrics and save them in *JSON* format [72]. In order to isolate ciphersuites with unique encryption algirithms for the purposes of encryption algorithms cost analsys, we developed a command-line tool that performs that filtering [73].

After being collected, the metrics were ready to be analyzed. For *callgrind* we developed tools that would use the collected measurments to estimate the number of CPU cycles for the specified *mbedTLS* functions [71] [74]. *mbedTLS* functions either implement TLS security services (*e.g.* authentication) or primitives used by TLS them (*e.g.* RSA signature creation). After associating the function name(s) to each relevant security service and primitive, their costs could be studied. Our tools allow to analyze the collected metrics both, individually (*e.g.* compute the cost the handshake for a specific ciphersuite), and in conjunction (*e.g.* compute the average cost of the handshake for all of the ciphersuites that use a specific key exchange method). A separate set of tooling with equivalent functionality was developed in order to analyze the time results obtained with PAPI [72].

In order to collect the *valgrind* metrics, no modification of *mbedTLS*'s source code was necessary. To obtain the time measurments, the relevant parts of the *mbedTLS* code had to be instrumented with PAPI [75]. In both cases, dedicated client and server executables were developed [75]. Each executable accepts two arguments: the *id* of the ciphersuite to use and the number of bytes to transmit to the other peer. This set up allowed us to profile the costs of all security services for each one of the ciphersuites. [75]

6 Results and Data Analysis

The (D)TLS protocol consists of two sub-protocols: the Handshake protocol and the Record protocol. During the Handshake protocol, the peers authenticate one to another, agree on the data encryption and integrity algorithms and establish the shared keys. During the Record protocol the peers exchange the data securely, using the algorithms and keys negotiated during the Handshake protocol. Those algorithms support the security services provided by (D)TLS. For example, peer entity authentication is usually offered by algorithms, such as RSA or ECDSA. The cost of the Handshake phase depends on the security services used. The cost of the Record phase depends on the amount of data transmitted.

The core of the Record protocol is the use of a symmetric encryption algorithm (*e.g* AES), used to provide confidentiality and an Hash-Based Messaage Authentication Code (HMAC) algorithm, used to provide data integrity and data origin authentication. The HMAC algorithm is not used if the symmetric encryption algorithm is an AEAD cipher, which already provides data integrity guarantees. Each encryption algorithm has a few possible varieties (*e.g.* the CBC, GCM and CCM modes in AES) and key sizes (*e.g.*, 128 and 256 bit for AES). The same is true for HMAC algorithms, which have a well defined MAC function, key size and output length. The performance of symmetric encryption algorithms and hash functions has been studied in detail by existing work [76] [77]. There is an approximately linear relation between the amount of data encrypted and the the cost, as we show in Section 6.7.

The previous paragraph does not apply to the Handshake protocol. In this part, there is more variety in the computational cost outcomes. There are a few reasons for that:

– Asymmetric cryptography algorithms are used to provide security services of authentication and PFS. For each algorithm, there are significantly more possible key sizes, when compared to symmetric encryption algorithms. Theoretically there is an infinite number of them, however practical limits exist[78].

- In (D)TLS, various algorithms, in various combinations can be used to provide authentication and PFS. Two types of public key algorithms than can be used: ECC-based and non-ECC-based. It is also possible to have authentication without asymmetric cryptography. PSK-based ciphersuites allow peers to authenticate one to another by the means of a shared secret. This shared secret is then input to the PRF in the premaster key generation, without the use of public key cryptography.
- The use of asymmetric cryptography leads to asymmetric costs (*i.e.* distinct costs) for the client and the server. The costs of asymmetric encryption algorithms vary greatly depending if the public or the private key is used. Also, while ECC algorithms tend to have a smaller computational footprint, this is not always the case. Some operations (*e.g* signature verification) is faster on the non-ECC counterparts. It is important to consider those factors, when for example, only one of the peers is constrained, since the costs for the client and the server will be different. In the Record phase we do not have that problem, since the costs of HMAC and encryption/decryption in symmetrical encryption algorithms is approximately the same.

Another cryptographic operation that is done in the Handshake phase is PRF keying material generation. This part, however, does not have a major influence on the computational costs and is the same for both peers, since it is essentially hashing operations.

It is clear that the Handshake protocol is significantly more complex, with more possible cost variations. Furthermore, existing work neither evaluated the costs of individual security services of TLS, nor their various combinations. For this reason, our work is concentrated around the Handshake protocol.

This section as structured as follows. Section 6.1 describes the various security levels used for evaluation. Section 6.2 presents an overview of the costs of authentication and PFS. Section 6.3 discusses the costs of symmetric (or PSK) authentication. In Section 6.4 we analyze the costs of asymmetric authentication. We begin by analyzing the cost of RSA's and ECDSA's operations and then put everything together and present the authentication cost for each key exchange method (thus, for each ciphersuite). Section 6.5 does a similar analysis, but his time for the cost of PFS. Here, once again, we begin by analyzing the costs of ECDH and DH, after which we use that information to arrive at the PFS cost for each key exchange method (thus, for each ciphersuite). Section 6.6 puts the information from the previous sections together and analyzes the cost of the TLS handshake as a whole. We do both, present actual handshake cost values obtained by profiling its cost directly, and show how to use the information from the previous sections to arrive to those cost values without the need of executing and profiling the handshake costs directly. We decompose TLS costs into a single formula. In Section 6.7 we discuss costs of confidentiality in TLS and compare the performance of all of the symmetric encryption algorithms present in *mbedTLS 2.7.0*. Having analyzed the estimates obtained with *valgrind*, we will turn our focus to time costs obtained with PAPI. Section 6.8 presents handshake time measurments collected with PAPI and compares them to the *valgrind* ones from Section 6.6. As we will show, they are similar. In Section 6.8 we go further, and compare PAPI's ratios of related algorithm opearations to the *valgrind* ones. In essence, we will compare PAPI's results to the valgrind one's analyzed in Sections 6.4 and 6.5, and once again show that they are similar. Finally, in section 6.9 we present some conclusions.

6.1 Security Levels

We performed evaluation under various security levels, which are presented in table 5. The defined security levels are based on modern day security practices. The equivalence between symmetric and asymmetric key sizes is based on table 4. It shows approximate comparable key sizes for symmetric and asymmetric key algorithms based on the best known algorithms for attacking them[23].

Unfortunately, mbedTLS does not have the exact ECC curves for the key sizes with the values specified in the the table. For this reason, for ECC we choose the closest larger value available.

Security Level	Symmetric	ECC	DH/DSA/RSA
S1	80	163	1024
S2	112	233	2048
S3	128	283	3072
S4	192	409	7680
S5	256	571	15360

Table 4: Comparable symmetric and asymmetric key sizes (in bits)

	low	**normal**	**high**	**very high**
Symmetric Key Size (bits)	128	128	192[1]	256
RSA/DH/DSA Key Size (bits)	1024	2048	4092	8192
ECC Key Size (bits)	163 [2]	233 [3]	317[4]	420[5]
HMAC	SHA-256	SHA-256	SHA-384	SHA-512 [6]

Table 5: Security levels used in evaluation

We based the **normal** security level on the current most used TLS configuration on the internet [14]. We could not find any typical values used in the sphere of IoT. In all of the security levels, a certificate chain with two certificates is used: the CA's certificate and the server's certificate. Only server-side authentication is used, since this is the most common scenario. We decided to use the smallest possible certificate chain, consisting of two certificates. This is not the norm on the internet, where the chain size is larger. However, adding more certificates to the chain would not provide additional information,

[1] The closest value available in mbedTLS 2.7.0 (rounded up) is 256 bit
[2] The closest value available in mbedTLS 2.7.0 (rounded up) is 224 bit
[3] The closest value available in mbedTLS 2.7.0 (rounded up)is 256 bit
[4] The closest value available in mbedTLS 2.7.0 (rounded up) is 384 bit
[5] The closest value available in mbedTLS 2.7.0 (rounded up) is 512 bit
[6] The strongest hash function available in mbedTLS is SHA-384

since the extra cost of additional certificates in the chain can be computed by adding the costs of making/verifying additional signatures and parsing the certificates.

For all security levels, the server certificates are either signed with a 2048 bit RSA with SHA-256 secure signing scheme [79], or with a 256 bit ECDSA with SHA-256 secure signing scheme (*secp256r1* curve) [80], depending on the ciphersuite. The chosen signature algorithm and key size is based on the usual CA practices. For example, Google's root certificates with common names *Google Internet Authority G2* and *Google Internet Authority G3* use this set up[81].

The server's certificates, however, contained public keys of different sizes, according to the certificate type. For this reason, it is possible to deduce the cost of the CA's signature using a different algorithm with a different key. For the **normal** security configuration, the certificates used by the server are as follows:

- if RSA authentication is used, the server's certificate will contain an 2048 bit RSA key;
- if ECDSA authentication is used, the server's certificate will contain a 256 bit ECC key (*secp256r1* curve);
- if DH is used for PFS, the negotiated key will be 2048 bit long;
- if ECDH is used fro PFS, the negotiated key will be 256 bit long and the *secp256r1* curve will be used. The choice of this curve was based on the preferred curve order of Google Chrome 67, the most used web browser in the world[82].

Table 6 contains a the certificate configuration and key size information for all of the security levels. The key sizes are presented in bits.

	low	**normal**	**high**	**very high**
RSA Key Size	1024	2048	4092	8192
ECDSA Key Size	192	256	384	521
DH Key Size	1024	2048	4092	8192
ECDH Key Size	192	256	384	521
ECC Curve	secp192k1	secp256r1	secp384r1	secp521r1

Table 6: Security levels configuration

6.2 Evaluation of Security Services in TLS

The TLS protocol offers various security services. The list of security services offered by a connection depends on the ciphersuite in use. Each ciphersuite consists of a set of algorithms, which have an implication on the performance and cost. It is important to evaluate the cost of each security service to assist the developer in making security/performance trade-offs. In this section we will make a first, high level analysis of the costs of security services in TLS.

As mentioned previously, the cost of the Handshake varies greatly, depending on the security level and algorithms used. In this section an overview of the Handshake costs for the various configurations will be presented. For this first evaluation, we will concentrate on the **normal** security level, which was defined in the text above. *mbedTLS 2.7.0* includes a total of 161 ciphersuites, with 10 unique key exchange methods, with 13 unique symmetric key encryption algorithms and 4 unique hash functions. Some of the ciphersuites are disabled by default, since they are considered weak. In the sphere of IoT, however, it might still make sense to use them in certain scenarios, if their cost is considerably lower. For this reason, we have enabled and evaluated all of the possible configurations.

Since in our evaluations, the server authenticates, but the client does not, a separate analysis for both peers will be made. We will begin with an overview of the costs for the various TLS configurations.

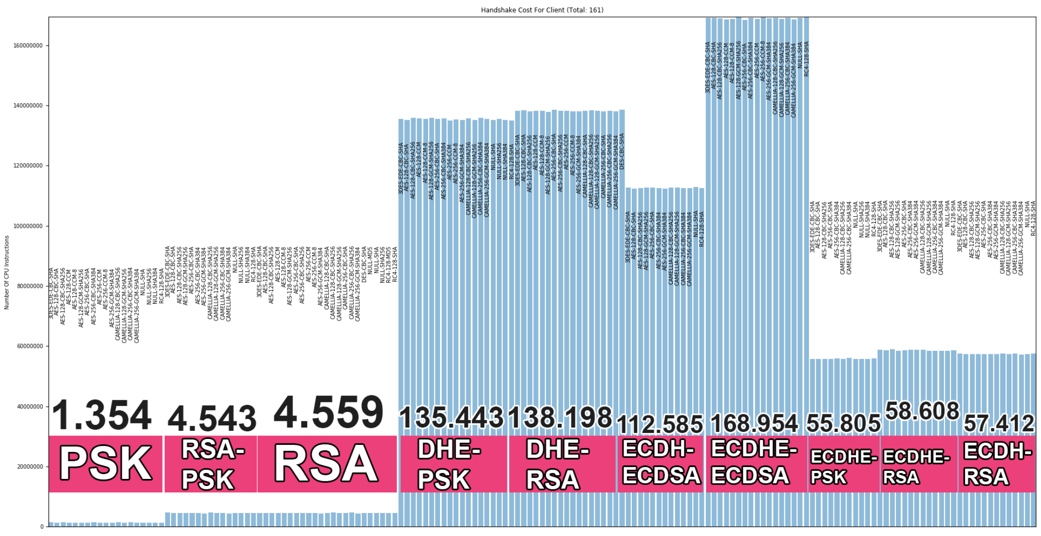


Fig. 9: Client handshake cost for all of the 161 ciphersuites

Figures 9 and 10 depict a graph with the Handshake cost of each one of the ciphersuites, for the client and the server, respectively. In both graphs, y axis represents the number of CPU cycles executed. The ciphersuites have been grouped by the key exchange method. Each bar within a key exchange method group represents a different combination of a symmetric encryption function and a hash function. The pink rectangles group the ciphersuites by the key exchange method. The numbers at the top of each rectangles indicate the average number of CPU cycles, in millions, used to perform the handshake for that key exchange method. An analysis of both figures shows that there is very little variation in costs within each key exchange method group. This is due to the fact that most of the resources are spent in authenticating and generating keying material for PFS.

From the analysis of Figures 9 and 10, it becomes obvious that some of the ciphersuites have much lower handshake costs than the others. For the client, the *PSK* based ciphersuites are the least and *ECHE-ECDSA* are the most expensive ones. For the server, the *PSK* based ciphersuites are the least and *DHE-RSA* are the most expensive ones. To make the presented information clearer, we took the average of the costs of all ciphersuites

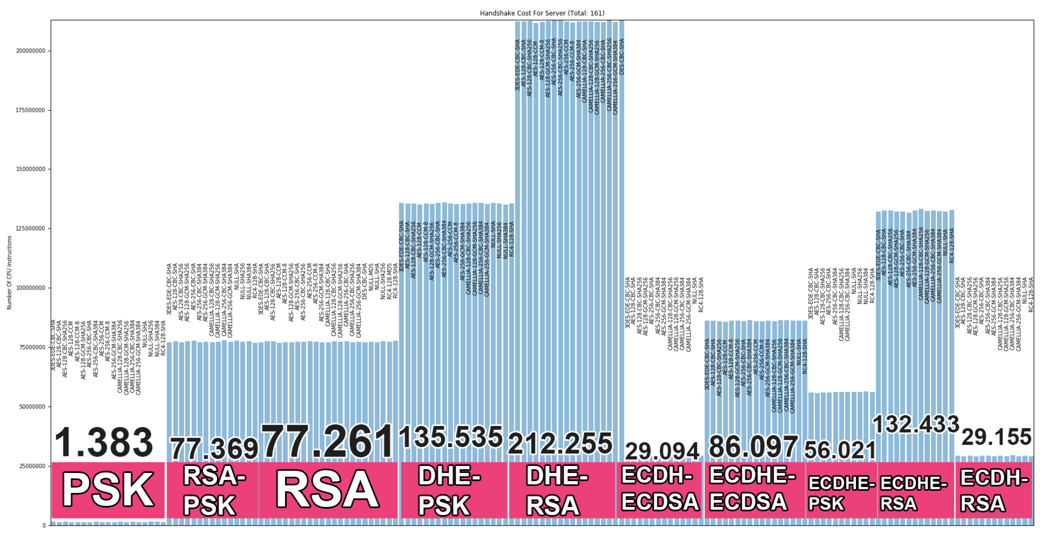


Fig. 10: Server handshake cost for all of the 161 ciphersuites

for each key exchange method. This information is presented in table 7 for the client and in table 8 for the server.

		PFS		No PFS		
		DHE	**ECDHE**	**ECDH**	**RSA**	**X**
Sym Auth	**PSK**	135.443	55.805		4.543	1.354
Asym Auth	**RSA**	138.198	58.608	57.412		4.559
	ECDSA		168.954	112.585		

Table 7: Average handshake cost for the client in millions CPU cycles

Each row and column intersection in the tables 7 and 8 forms a ciphersuite. The rows separate the various authentication algorithms that can be used. The columns separate the key exchange methods that offer PFS, from the ones that don't. Each entry of the table presents the average number of CPU cycles, in millions. In order to simplify this initial analysis, we are not presenting the standard deviation values for the averages and are rounding up the results to millions, with 3 decimal places. In Section 6.6 we will present a more detailed evaluation for all security levels.

Client By analyzing table 7 we can approximate the costs of the security services of authentication and PFS for the client. Even though those values are approximations, they are very close actual numbers, since they dominate the costs of the Handshake. All of the costs presented in the analysis that follows are taken directly from table 7 and will be in millions CPU cycles, rounded to 3 decimal places.

		PFS		No PFS		
		DHE	**ECDHE**	**ECDH**	**RSA**	**X**
Sym Auth	**PSK**	135.535	56.021		77.369	1.383
Asym Auth	**RSA**	212.255	132.433	29.155		77.261
	ECDSA		86.097	29.094		

Table 8: Average Handshake cost for the server in millions CPU cycles

If PFS is used, the cost of the Handshake varies from 55.805 to 135.443 million CPU instructions for PSK based authentication, and from 58.608 to 168.954 million CPU cycles for asymmetric authentication. If the security service of PFS is foregone, the cost of the Handshake varies from 1.354 to 4.543 for PSK based authentication, and from 4.559 to 112.585 for asymmetric authentication.

If we analyze the values in the *ECDHE* column in table 7, thus fixing the algorithm used to offer PFS we can compare the costs of authentication for various algorithms on the client-side. For example, we can see that *RSA* authentication costs 2.803 ($58.608 - 55.805$) more than PSK authentication and *ECDSA* authentication costs 110.346 ($168.954 - 58.608$) million CPU cycles more than *RSA* authentication. The total cost of using PSK for authentication can be estimated by looking at the value in the *PSK* row and *X* column, thus fixing authentication to *PSK* and not using any other algorithm for key agreement: 1.354. In fact, we can consider this value to be the overhead of TLS for the client, since this is the cost of establishing the most basic TLS connection available in *mbedTLS 2.7.0*. In the same manner, the cost of using *RSA* for authentication can be approximated by taking the value located in row *RSA* and column *X*, thus fixing authentication to RSA and not using any other algorithm for key agreement: 4.559. Even though we cannot estimate the cost of using *ECDSA* for authentication directly from the table, we have already seen that *ECDSA* authentication is 110.346 more costly than *RSA* authentication. Thus, the cost of using *ECDSA* for authentication is of 114.905($4.559 + 110.346$) million CPU cycles. When compared, authentication with RSA is 236.7% more expensive than with PSK and authentication with ECDSA is 2420.4% more expensive than with RSA.

Similarly, we can compare the costs of using DH vs ECDH to provide PFS, by looking at the PSK (or RSA) row, thus fixing the authentication algorithm. Using *DHE* is 76.638 ($135.443 - 55.805$) million CPU cycles more (+142.7%) expensive than using *ECDHE*. The total cost of PFS using the ECDH algorithm can be computed by fixing the *PSK* row (or RSA row) and subtracting the value in the *ECDHE* column from the value in the *X* column. By doing this, we are fixing the authentication algorithm to PSK and subtracting the cost of the Handshake when no PFS is used, from the cost of the handshake when *ECDHE* is used to provide PFS. Thus, the cost of using *ECDHE* to provide PFS is of 54.451 million CPU cycles ($55.805 - 1.354$). Since we already know that *DHE* costs 76.638 million CPU cycles more than *ECDHE*, we can compute the cost of using the *DHE* to provide PFS: 131.089($54.451 + 76.638$) million CPU cycles.

Server We will now perform the same analysis for the server by analyzing the table 8. Once again, the presented costs will be approximations in being millions CPU cycles rounded to 3 decimal places.

If PFS is used, the cost of the Handshake varies from 56.021 to 135.535 million CPU instructions for PSK based authentication, and from 86.097 to 212.255 million CPU cycles for asymmetric authentication. If the security service of PFS is foregone, the cost of the Handshake varies from 1.383 to 77.369 for PSK based authentication, and from 27.094 to 77.261 for asymmetric authentication.

If we analyze the values in the *ECDHE* column in table 8, thus fixing the algorithm used to offer PFS we can compare the costs of authentication for various algorithms on the server-side. For example, we can see that *RSA* authentication costs 76.412 $(132.433 - 56.021)$ more than PSK authentication and *ECDSA* authentication costs 46.336 $(132.433 - 86.097)$ million CPU cycles less than *RSA* authentication. The total cost of using PSK for authentication can be estimated by looking at the value in the *PSK* row and *X* column, thus fixing authentication to *PSK* and not using any other algorithm for key agreement: 1.383. In fact, we can consider this value to be the overhead of TLS for the client, since this is basically the cost of establishing the most basic TLS connection available in *mbedTLS 2.7.0*. In the same manner, the cost of using *RSA* for authentication can be approximated by taking the value located in row *RSA* and column *X*, thus fixing authentication to RSA and not using any other algorithm for key agreement: 77.261. Even though we cannot estimate the cost of using *ECDSA* for authentication directly from the table, we have already seen that *ECDSA* authentication is 46.336 less costly than *RSA* authentication. Thus, the cost of using *ECDSA* for authentication is of $30.925(77.261 - 46.336)$ million CPU cycles. When compared, authentication with ECDSA is 2136.1% more expensive than with PSK and authentication with RSA is 149.8% more expensive than with ECDSA.

Similarly, we can compare the costs of using DH vs ECDH to provide PFS, by looking at the PSK (or RSA) row, thus fixing the authentication algorithm. Using *DHE* is 79.514 $(135.535 - 56.021)$ million CPU cycles more expensive (+141.9%) than using *ECDHE*. The total cost of PFS using the ECDH algorithm can be computed by fixing the *PSK* row (or RSA row) and subtracting the value in the *ECDHE* column from the value in the *X* column. By doing this, we are fixing the authentication algorithm to PSK and subtracting the cost of the Handshake when no PFS is used, from the cost of the handshake when *ECDHE* is used to provide PFS. Thus, the cost of using *ECDHE* to provide PFS is of $54.638(56.021 - 1.383)$ million CPU cycles . Since we already know that *DHE* costs 79.514 million CPU cycles more than *ECDHE*, we can compute the cost of using the *DHE* to provide PFS: $134.152(54.638 + 79.514)$ million CPU cycles.

Discussion By analyzing at tables 7 and 8 it is clear that some ciphersuites are more costly than others. This dissimilarity can explained by the by fact that different ciphersuites use different security services and different algorithms to offer those security services. By analyzing the Handshake costs we were able to approximate the costs of the security services of authentication and PFS, as well as of the algorithms used to offer them. This analysis made it clear that the use or non-use of a security service can have a big impact on the cost of establishing a TLS connection.

Our analysis showed that *PSK* based ciphersuites are the most efficient ones overall, for both, the client and the server, thus their popularity in the IoT environment. Symmetric authentication is, by far, the least costly one for both peers. For the client, RSA is the second least costly authentication and ECDSA is the most costly one of them. For the server, this situation is reversed. The reasons for that will be explained in Section 6.4, when we analyze the Handshake costs in detail.

As for the PFS, in both cases *ECDHE* is about 1.5 less costly than *DHE*. This gives us a glimpse into the benefits of elliptic curve cryptography. The costs of PFS are identical for both peers. In the next section we will analyze both of the security services in more detail.

6.3 PSK Authentication Cost Analysis

In the previous section we compared the costs of different authentication methods by analyzing the total cost of the Handshake with different ciphersuites. In this section and section 6.4, we will analyze this security service in detail, including the underlying algorithms it.

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. We will begin with an analysis of PSK authentication cost in this section and analyze asymmetric authentication costs in Section 6.4.

In the previous section we approximated the cost of PSK authentication to be 1.354 million CPU cycles for the client and 1.383 million CPU cycles for the server. In reality, this cost is very close to zero, since both of the parties already have the authentication key. The approximated value is actually the cost of performing the smallest and least costly handshake in *mbedTLS*. This value can be considered as the TLS overhead, since it is in essence, the cost of establishing the simplest possible TLS connection, without using asymmetric authentication or PFS. The majority of those costs are spent in the PRF, when generating the shared keying material and in reading/writing the *Finished* message.

Table 9 shows the number of CPU cycles spent in the PRF and in computing the *Finished* message for all of the defined security levels. This number is the average of the runs with each one the 161 ciphersuites for both of the peers. Thus, it is an average of 322 samples. The standard deviation shown in parenthesis. An analysis of the table shows that the cost of PRF is almost the same for all security levels. This can be explained by the fact that it's essentially composed of hashing operations and even if the input size varies by a few hundred bytes, the total cost does not change by a significant amount. In *mbedTLS 2.7.0* there are 26 unique encryption/hash algorithm combinations, 4 of which use a *NULL* encryption algorithm. The *Finished* message is the first encrypted message in TLS. Different encryption algorithms have different performances. Due to this heterogeneity, the standard deviation is relatively higher for writing and parsing the *Finished* message.

Security Level / Opeation	**low**	**normal**	**high**	**very high**
PRF	982957 (14830)	986854 (19227)	992441 (29439)	1009178 (50185)
Finished Write	158717 (22451)	157912 (23253)	156648 (21950)	158306 (22561)
Finished Parse	164680 (23952)	164984 (26244)	163501 (24703)	164897 (25322)

Table 9: Cost of PRF and *Finished* message

The values in table 9 remain valid for all key exchange methods, and as we will see, become insignificant.

6.4 Asymmetric Algorithms Authentication Cost Analysis

If asymmetric cryptography is used for authentication, there are two choices of algorithms: RSA and ECDSA. Each one of them has advantages and disadvantages, depending on the scenario. We will analyze them now.

In *mbedTLS 2.7.0*, there are 31 ciphersuites that use RSA to authenticate ephemeral DH/ECDH parameters and 17 ciphersuites that use ECDSA to authenticate ephemeral DH/ECDH parameters. The metrics from those ciphersuites can be used to analyze the cost of public and private key operations for both algorithms. Thus, all of the presented costs for for RSA are an average computed from 31 runs, and for ECDSA an average computed from 17 runs, all from different ciphersuites.

RSA and ECDSA have two basic operations: sign and verify. The first one uses the private key, while the second one the public key. Figure 11 compares the performance of the algorithm's operations for the **normal** security level (*i.e.* 2048 bit RSA key and 256 bit ECDSA key). The *Total* cost is the sum of the *Sign* and *Verify* costs for the corresponding algorithm.

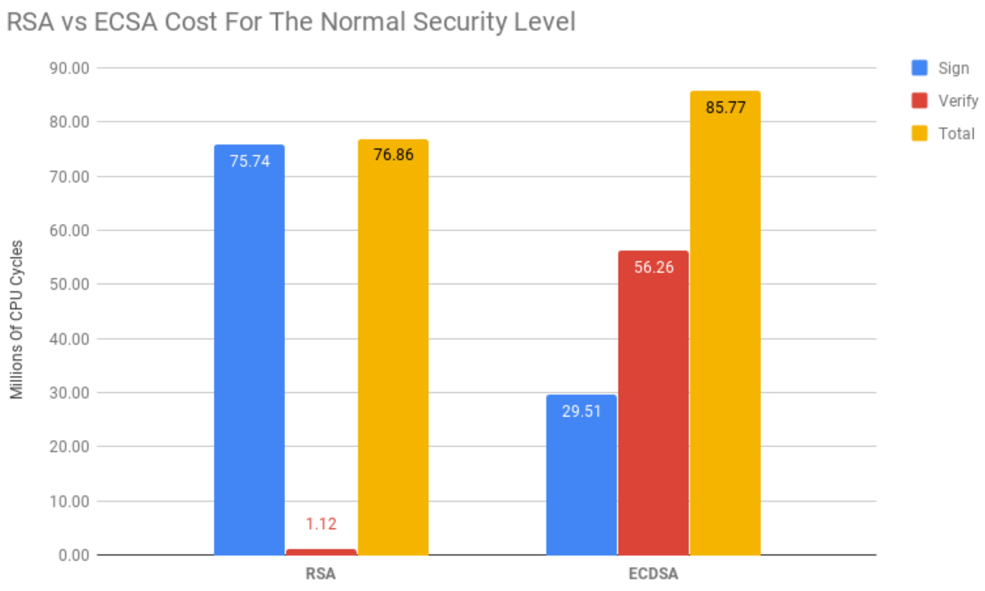


Fig. 11: RSA and ECDSA operations cost for normal security level

By analyzing the graph, we can can observe two things:

– RSA is more efficient at performing public key operations, *i.e.* verifying the signature

– ECDSA is more efficient at performing private key operations, *i.e.* making the signature

This holds true for other security levels too, as will be shown further down the text. Let us now analyze each one of the algorithms and their costs in more detail.

RSA Cost Analysis We have already seen the costs of RSA for the **normal** security level and now we will analyze the remaining ones. Table A.1 shows the number of CPU cycles used when signing a message with size 20, 32 or 48 bytes and verifying the resulting signature with RSA and ECDSA. As already mentioned, the values are an average of 31 runs for RSA and 17 for ECDSA. The numbers in parenthesis is the standard deviation. All of the values are rounded up to significant digits. The size of the signed message depends on the output size of the hash function used with the ciphersuite. The cost differences between them are minimal, and for this reason we decided not to separate them. Table A.2 presents the sum of the signing and verifying operation cost for RSA and ECDSA, which we call the *Total* cost. In out tests, both RSA and ECDSA signed certificates have signatures over a 32 byte value (output of *SHA-256*), so the values in Table A.1 apply to those costs too.

Figure 12 depicts the costs of RSA operations graphically. The data is presented in a logarithmic scale. An analysis of the data shows that the exponential trendline is the one that best fits the cost increase for both, signature creation and verification. This is a result of modular exponentiation being RSA's core operation. Moreover, for all security levels, the cost of private key operations is significantly higher than of the public key ones. The cost of the former also increases more, as the security level raises. For example, there is an increase of 2145% from the *normal* to *very high* security level for signature creation and of 1102% for signature verification.

This results in a big difference between the cost of creating and verifying signatures, which becomes larger as the security level increases. For example, at the *normal* security level signature verification is approximately 74.6 million CPU cycles more expensive, while at the *very high* security level, this value raises to over 1687 million, *i.e.* a 2161% increase.

This increase more modest for public key operations, as is shown in Figure 13. The percentages show the relative cost increase from the previous security level. For example, creating an RSA signature costs 268.4% more at *normal* than at *low* security level. For all operations, the cost increase is exponential. The consequences of the exponential cost increase are shown in table A.3, which presents this relative increase in terms of absolute values. The numbers are increasing, with the signature creation increase being significantly larger than the signature verification one. An analysis Figure 13 explains the difference between the costs of private and public key operations. As the security level increases, the relative cost increase is larger for signing than for verifying. On average, the signing operation cost increase from one security level to another is about 1.5 larger than the verifying one. This has a cumulative effect: the constantly larger value increases even more. This is shown graphically in Figure 14, which for RSA, presents the ratio between the cost of private and public key operations. As the security level increases, the value becomes larger. The fact that the signing operation dominates the total cost can also be seen graphically in Figure 13. The *Total* cost increase line is very close to the *RSA Sign* cost increase line, in fact, they're almost identical. On average, the *Total* cost increase is only 1.4% smaller than the signing operation cost increase.

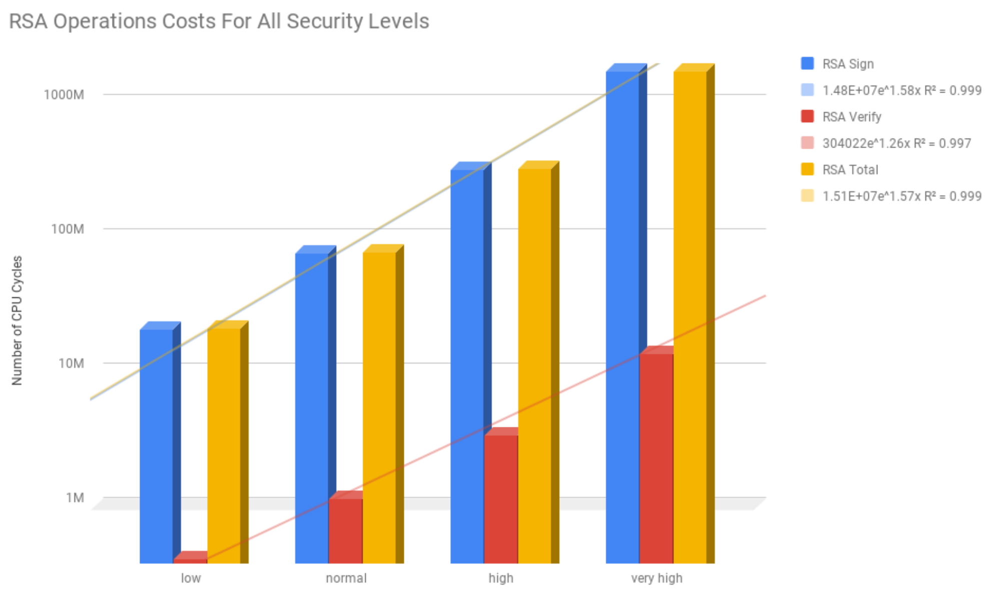


Fig. 12: RSA operations costs for all security levels (logarithmic scale)

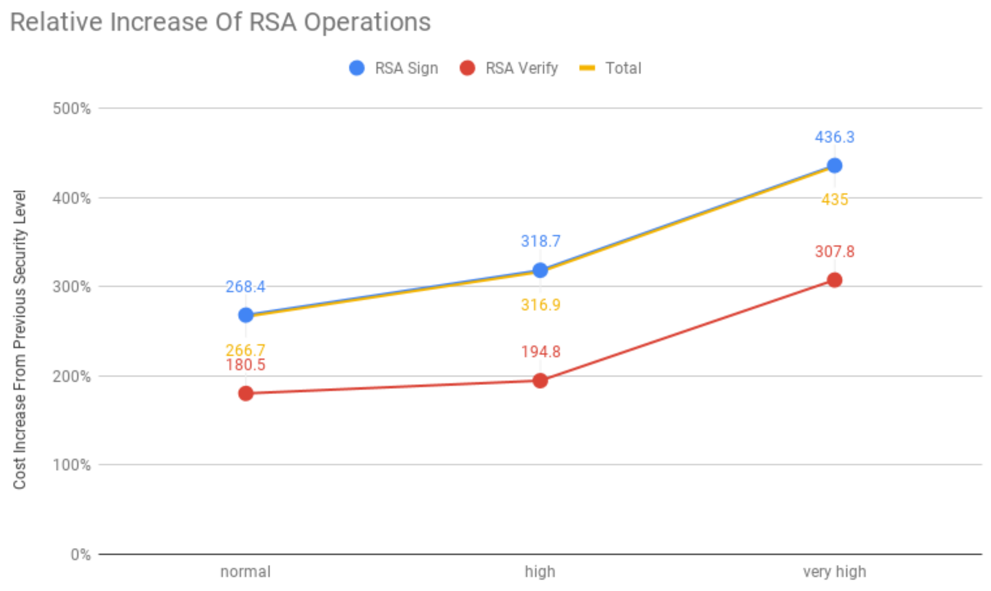


Fig. 13: Relative increase of RSA operation costs

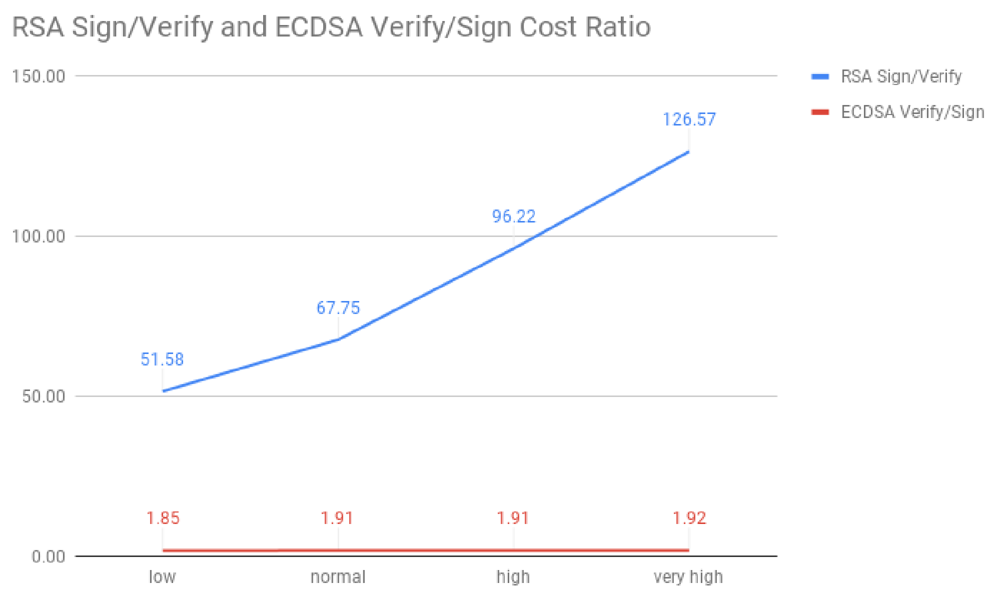


Fig. 14: Ratio between the most and the least costly operation for RSA and ECDSA

The reason for the discrepancy between the cost of public and private key operations has to do with an optimization in the choice of the keys. RSA's public keys are deliberately chosen to have a small public exponent, such as $e = 65537$. This is considered as a sensible compromise, since it is famously known to be prime, large enough to avoid the attacks to which small exponents make RSA vulnerable, and can be computed extremely quickly on binary computers [83][84]. A smaller exponent leads to less exponentiation operations, thus better performance. For this reason, public key operations with RSA are faster than private key ones.

ECDSA Cost Analysis After analyzing the costs of RSA, we will now perform a similar analysis for ECDSA. Figure 16 shows the costs of ECDSA operations graphically. The values are taken from Table A.1. Unlike in RSA, in ECDSA the private key operation is the least costly one. Figure 14 shows this ratio for both, RSA and ECDSA. Besides the ratio being a lot smaller for ECDSA, it varies very little. This means, that no matter the security level, ECDSA signature verification will always be about 2 times more costly than signature creation.

As it is shown in figure 16, a logarithmic trendline is the one that best fits the cost increase for both, signature creation and verification. Figure 15 shows the relative cost increase of of ECDSA's operations. The percentages are the relative cost increase from the previous security level. There are two major differences from RSA. First, the relative cost increase of signature creation and verification is very similar. Consequently, the same is true for the total cost increase. This can be confirmed graphically in Figure 15, where all of the lines are close one to another. As a result, even with the increase of security level, none of the operations will dominate in cost as much as it happens in RSA. Second, as the security level increases, the cost increase from the previous security level becomes smaller. In fact, after the **high** security level, the absolute cost increase starts decreasing as well. We can see this in table A.4. This is a consequence of the cost increase being logarithmic, which is a result of ECDSA's core mathematical operation being multiplication of a scalar by a point on the curve. Although not presented here, this trend continues for higher security levels. Those properties of ECDSA makes the security level increase more manageable. It's not as costly to increase the security level for ECDSA as it for RSA.

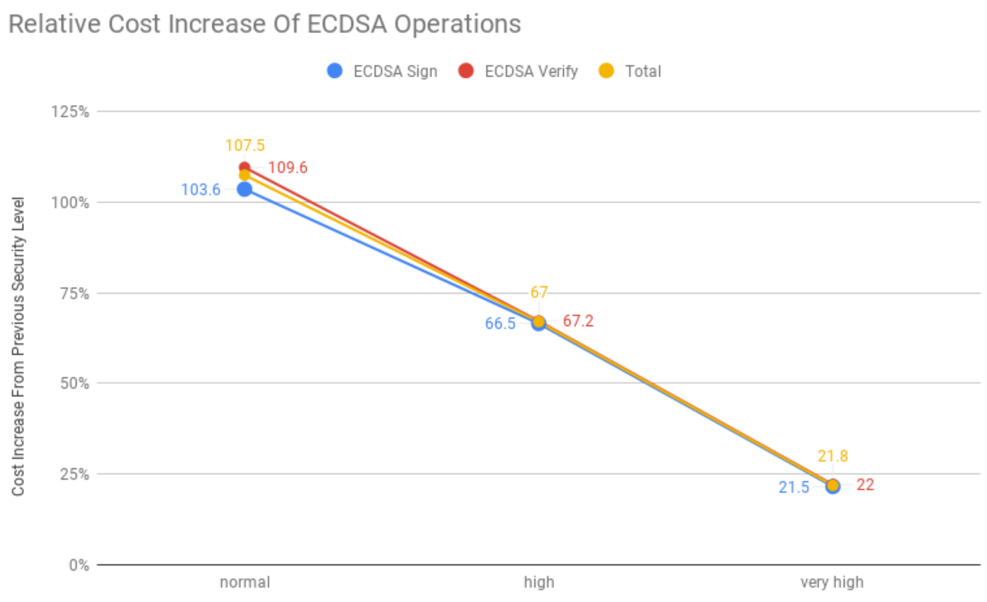


Fig. 15: Relative increase of ECDSA operations cost

In the previous subsection we have described an optimization in the choice of keys for RSA. There are no such optimizations for ECDSA. For this reason, it is expected that the cost of both operations will increase in similar proportion. The non-optimized key (private for RSA, private and public for ECDSA) operation cost increase is a lot smaller for ECDSA. This can be explained by the fact that smaller keys are used in ECDSA, as we have already seen in Table 4, and is a big advantage of ECC.

RSA vs ECDSA After having analyzed the costs of RSA and ECDSA, we will now compare both and draw some conclusions. The answer to the question of which one of algorithms is less costly will vary depending on the security level and the operation. Figure 17 shows the costs of both, RSA's and ECDSA's operations. The data is presented in logarithmic scale. Since for RSA most of the *Total* cost comes from the signature creation operation, those two lines overlap in the graph.

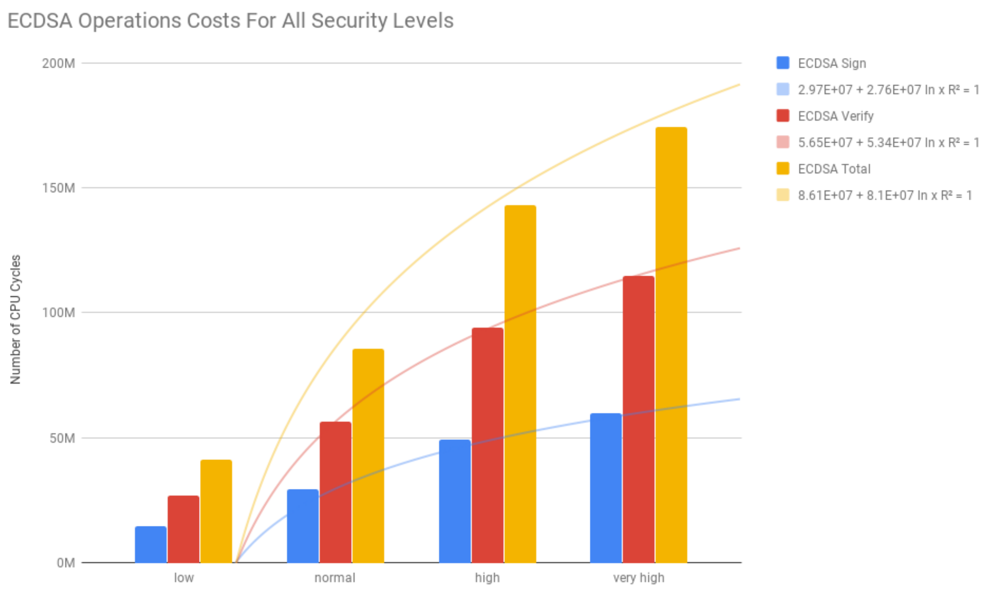


Fig. 16: ECDSA costs for all security levels

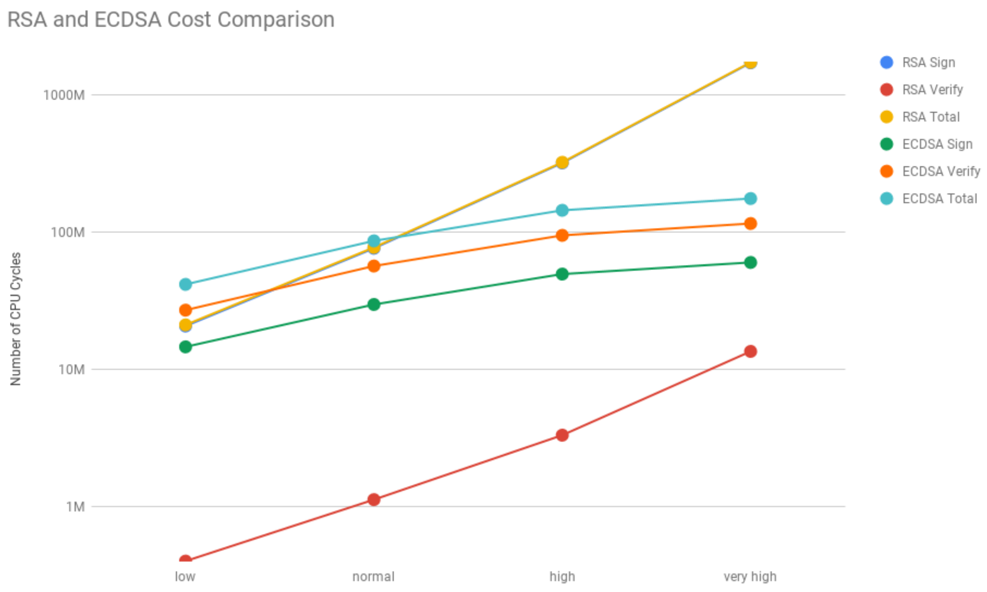


Fig. 17: RSA and ECDSA cost comparison

By analyzing Figure 17, we can answer the question of *Which algorithm is less costly?*:

- RSA is always less costly at signature verification
- ECDSA is always less costly at signature creation
- RSA's cost increase is exponential
- ECDSA's cost increase is logarithmic
- Total cost of RSA is smaller for the *low* and *normal* security levels
- Total cost of ECDSA is smaller for the *high* and *very high* security levels

By analyzing figures 13 and 15, and tables A.3 and A.4 which show the relative and absolute cost increases for RSA and ECDSA operations, we can see that as the security level increases, the cost increase of RSA operations from the previous security level always always increases, while after the **normal** security level, the cost increase of ECDSA operations from the previous security level starts decreasing. Although not presented here, our tests show that this trend holds true for higher security levels as well. This is a consequence of RSA having an exponential cost increase, while ECDSA a logarithmic one. For this reason, it is safe to say that for security levels higher than the ones that we defined, ECDSA would be the preferred choice. RSA's cost increases exponentially due to the mathematical operation at the algorithm's core: modular exponentiation. Similarly, the cost increase in ECDSA is logarithmic, due its ECC properties: the base mathematical operation is multiplication of a scalar by a point on the curve.

So which algorithm should be used for each security level? The answer to this question 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 have the costs for both peers as similar as possible, ECDSA is the algorithm to use.

This information can also be used to make certificate choices for mutual authentication scenarios, *i.e.* when both, the client and the server authenticate one to another. For example, if only one of the nodes is constrained, an RSA-singed certificate from the non-constrained node and an ECDSA-signed certificate from the constrained node would minimize the costs for the constrained node. If both of the nodes are constrained, then the choice of the least costly algorithm will be guided by the *Total* cost: RSA for the *low* and *normal* security levels and ECDSA for the *high* and *very high* security levels.

The Cost Of Authentication in TLS Having analyzed the costs of the algorithms that can be used for authentication, we will now describe the cost of this security service for each one of the ciphersuites for the client and the server. Table A.5 shows authentication costs for each ciphersuite for the client. Table A.6 shows the same information for the server. Each row specifies a key exchange method and each column the security level.

Figures 18 and 19 are a graphical representation of tables A.5 and A.6, respectively. In both of the figures, the data is presented in logarithmic scale. By looking at the graphs, it becomes evident that the some ciphersuites can be grouped together by authentication cost. For the client, those groups are:

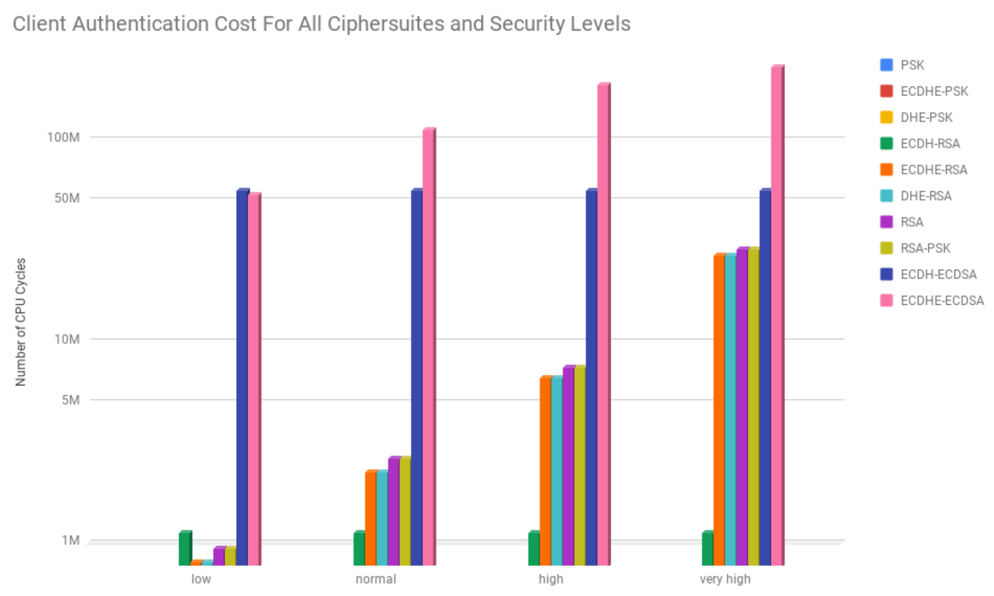


Fig. 18: Client authentication cost (logarithmic scale)

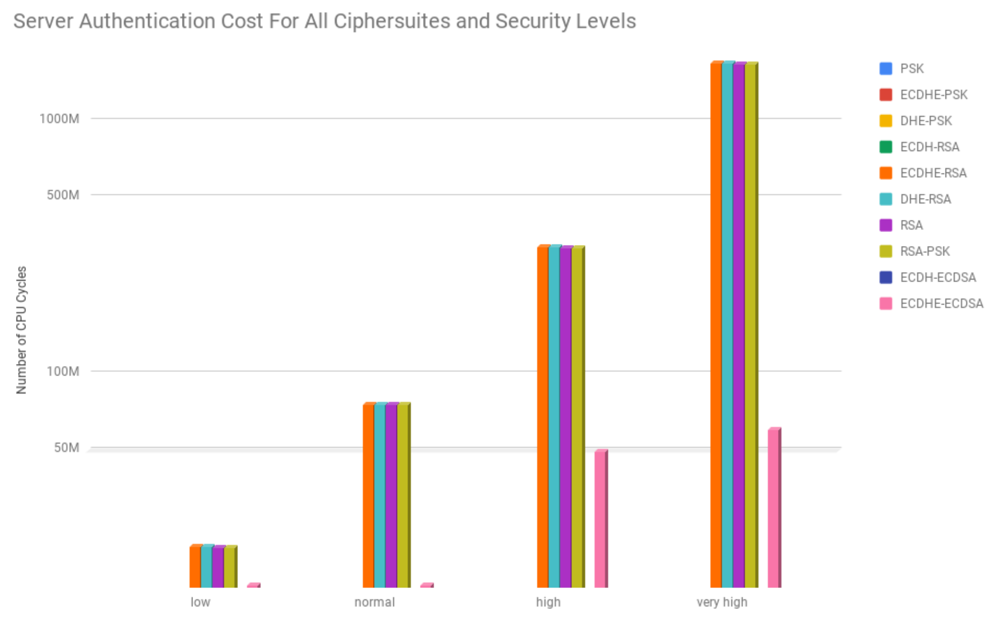


Fig. 19: Server authentication cost (logarithmic scale)

1. *PSK*, *ECDHE-PSK*, *DHE-PSK*
2. *ECDH-RSA*
3. *ECDHE-RSA*, *DHE-RSA*
4. *RSA*, *RSA-PSK*
5. *ECHD-ECDSA*
6. *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*
4. *RSA*, *RSA-PSK*

Inside every group, the authentication cost is the same, regardless of the security level. 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 ciphersuites in the same group share a common set of operations that are performed to provide authentication. We will discuss what those operations are further down in the text. Each ciphersuite uses either PSK, RSA or ECDSA for authentication. If only PSK is used for authentication, the cost of authentication is 0. This is the case of Group 1 for the client and the server. At the end of the previous section we discussed which algorithm, RSA or ECDSA would be the least costly choice. An analysis of authentication costs in TLS goes in hand with that discussion. By analyzing tables A.5 and A.6, and figures 18 and 19, we can see that the cheapest choice for the client is RSA and for the server is ECDSA. Similarly, for PFS enabled ciphersuites, if the goal is to make the cost distribution as even as possible among the peers, ECDSA is the preferred choice. Having presented the costs of authentication for various key exchange methods and made a high-level analysis, we will now go into more detail and justify each value.

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. We can use the values for the appropriate security level from table A.1 to estimate them.

All RSA server certificates are signed with a 2048 bit RSA key and all ECDSA certificates are signed with a 256 bit ECDSA key. This signature is made over an *SHA-256* hash, which has an output size of 32 bytes. Thus, we can use the values from the *normal* security level from table A.1 to compute the certificate signature verification costs.

In *RSA* and *RSA-PSK* ciphersuites, the client uses the *PKCS#1 v2.1 RSAES-PKCS1-V1_5ENCRYPT*[85] encryption scheme to encrypt the 48 byte premaster secret. The server uses the corresponding *PKCS#1 v2.1 RSAES-PKCS1V1_5DECRYPT*[85] decryption scheme to decrypt it. The costs of those operations are higher than of the regular sign/verify ones, due to extra steps performed. Thus, to compute the authentication cost for those ciphersuites, in addition the the values from table A.1, the ones from table 10 will also be used. In *mbedTLS 2.7.0*, there are a total of 38 ciphersuites that use the *PKCS#1* *v2.1* scheme as part of the authentication process: 23 *RSA* ciphersuites and 15 *RSA-PSK* ciphersuites. The values in table 10 are an average of 38 runs: one for each ciphersuite. The numbers in parenthesis is the standard deviation.

Security Level / Operation	**low**	**normal**	**high**	**very high**
Encrypt	542291 (836)	1504367 (2012)	4167693 (2866)	15280266 (3748)
Decrypt	20362831 (125590)	75129504 (252921)	314975365 (676291)	1691976601 (2015526)

Table 10: Cost of using *PKCS#1 V2.1 RSAES-PKCS1-v1_5* encryption and decryption schemes with various security levels

The encryption operation uses the server's public key and the decryption operation the corresponding private key. Thus, as expected, decryption is more costly than the encryption and both of the values increase, as the security level increases.

In order to authenticate, the client and the server perform different steps, depending on the ciphersuite. More specifically:

– in *PSK*, *DHE-PSK* and *ECDHE-PSK* the authentication is done exclusively through the pre-shared secret, without any operations, thus the authentication cost is 0.
– in *RSA* and *RSA-PSK* the client has to verify the server's RSA-signed certificate and encrypt the premaster secret with the server's public RSA key, while the server has to decrypt the premaster secret with the corresponding private key.
– in *ECDH-RSA* ciphersuites, the client has to verify the server's RSA-signed certificate and the server does not need to perform any operations.
– in *ECDH-ECDSA* ciphersuites, the client has to verify the server's ECDSA-signed certificate and the server does not need to perform any operations.
– in *ECHDE-RSA* and *DHE-RSA* ciphersuites the client has to verify the server's RSA-signed certificate and the *(EC)DH* RSA-signed parameters, while the server has to perform an RSA signature over the hash of *(EC)DH* parameters.
– in *ECHDE-ECDSA* ciphersuites the client has to verify the server's ECDSA-signed certificate and the *(EC)DH* ECDSA-signed parameters, while the server has to perform an ECDSA signature over the hash of *(EC)DH* parameters.

In order to compute the authentication cost for each peer, we use the values from tables A.1 and 10, sum them up according to the steps described above. For example, when an *ECDHE-ECDSA* ciphersuite is used at the *normal* security level, the client verifies two ECDSA signatures: one from the parsed server's certificate and one from the *ServerKeyExchange* message, while the server only performs a signature over the *ECDHE* parameters. Thus, the authentication cost for the client will be $56260702 + 56260702 = 112521404$ and for the server 29512991. Similarly, when an *RSA* or *RSA-PSK* ciphersuite is used at the *normal* security level, the client will verify the RSA signature in the server's certificate and perform a *PKCS#1 v2.1 RSAES-PKCS1V1_5* encryption, while the server will only need to perform a *PKCS#1 v2.1 RSAES-PKCS1V1_5* decryption. Thus, the authentication cost for the client will be $1117868 + 1504367 = 2622235$ and for the server 75129504. The remaining entries in tables A.5 and A.6 are computed in a similar manner. It is important to remember that independently of the security level, in our evaluation the server's certificates are always signed with either a 2048 bit RSA keys or 256 bit ECDSA keys. Thus, for the certificate signature verification cost on the client side, the values from the *normal* security row of table A.1 are used.

Since in our evaluated scenario only the server authenticates, tables A.5 and A.6 are non-identical. If mutual authentication was used (*i.e.* with both, the client and server authenticating one to another), there would be an additional cost of creating a signature for the client and of verifying the corresponding signature for the server.

6.5 Perfect Forward Secrecy Cost Analysis

In TLS there are two ways of achieving PFS: either by using the DH algorithm or its ECC counterpart ECDH. In section 6.2 we estimated the cost of PFS and compared the two methods of achieving it by analyzing the total cost of the Handshake with different ciphersuites. In this section, we will analyze this security service in detail, including the underlying algorithms. In DH and ECDH 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 *mbedTLS 2.7.0* there are a total of 78 ciphersuites that offer PFS. 41 of them use the ECDH algorithm and 37 of them use the DH algorithm. There are also 26 ciphersuites that do not offer PFS, but still use the ECDH algorithm. The metrics obtained from the Handshake analysis with those ciphersuites can be used to analyze the cost of the algorithms. Since the operations performed at the client and the server side are identical, we do not need to present two separate analysis, like we did throughout Section 6.4. Moreover for *ECDHE* and *DHE*, we can analyze the metrics from both of the peers in conjunction, thus doubling the sample size. As for *ECDH* ciphersuites, we can use the client-side results to analyze the cost of all ECDH operations and the server-side results to analyze the cost of the shared ECDH secret generation operation.

Thus, all the presented cost values for ECDH's ephemeral key pair generation are an average computed from 108 runs (41 from each peer's PFS ciphersuites and 26 from client's non-PFS ciphersuites), and for the shared secret generation an average of 134 runs (41 from each peer's PFS ciphersuites and 26 from each peer's non-PFS ciphersuites). For the DH algorithm, the average for both operations is computed from 74 runs (37 from each peer).

Figure 20 compares the cost of ECDH's and DH's operations for the **normal** security level. The total cost is the of the keypair and shared secret generation values for the corresponding algorithm. By analyzing the graph, we can can observe two things:

– ECDH is less costly than DH
– for both algorithms, the costs of generating the key pair and the shared secret are very similar.

The first observation is true starting from the **normal** security level and onwards. The fact that for each algorithm the cost of generating the public/private keypair is very similar to the cost of generating secret is not a coincidence. ECDH and DH use different mathematical operations, but for each algorithm, those operations are the same for the

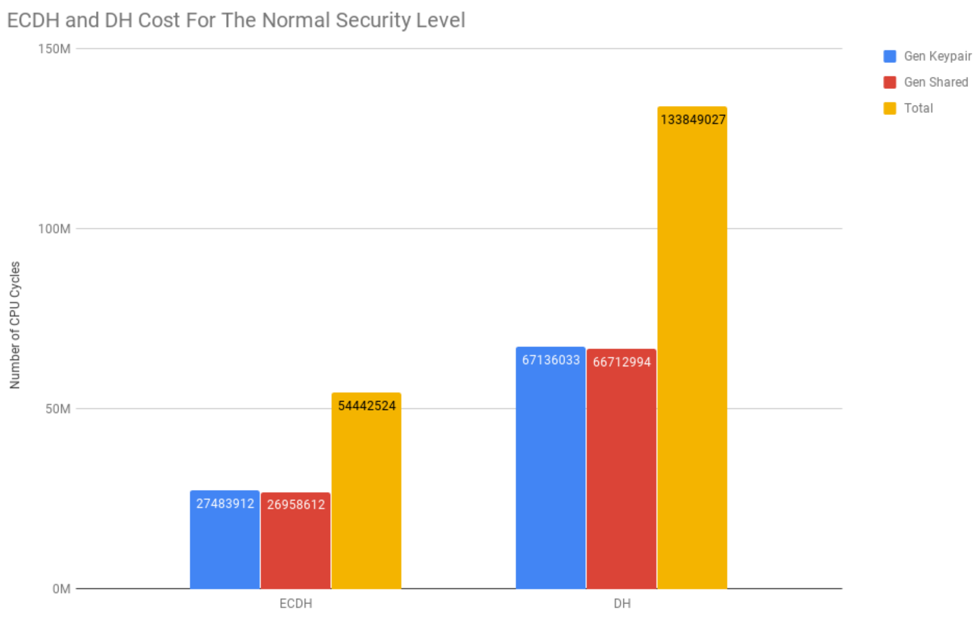


Fig. 20: ECDH and DH operations cost for normal security level

operations of keypair shared secret generation. Thus, it is expected for the cost of both steps being almost identical. We will now analyze each one of the algorithms and their costs in more detail.

ECDH Cost Analysis In ECDH, two basic operations are performed by each peer: first, a public/private ECC keypair is generated, followed by generation of the shared secret. 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 preshared secret. 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.

We have already seen the costs of ECDH for the **normal** security level and now we will analyze the remaining ones. Table A.7 shows the number of CPU cycles for each ECDH and DH operation. As already mentioned, the values are an average of 108 runs for the ECDH's key generation, 134 runs for ECDH's shared secret generation and of 34 runs for both of DH's operations. The numbers presented in parenthesis is the standard deviation. All of the values are rounded up to significant digits. Table A.8 presents the sum of keypair and shared secret generation operations for for ECDH and DH, which we call the *Total* cost. For both algorithms, the cost of generating the private key is less than 1% of the keypair generation operation. This is expected as the private key is just a randomly generated number, with size specific to the elliptic curve's group.

Figure 21 depicts the cost of ECDH operations graphically. For all security levels the total cost is almost evenly divided between the keypair and the shared secret generation.

This is justified by the fact that the underlying mathematical operation is the same when generating the public/private keys and the shared secret: multiplications of a scalar by a point on the curve. An analysis of the figure also shows that a logarithmic trendline is the one that best fits the cost increase for all operations.

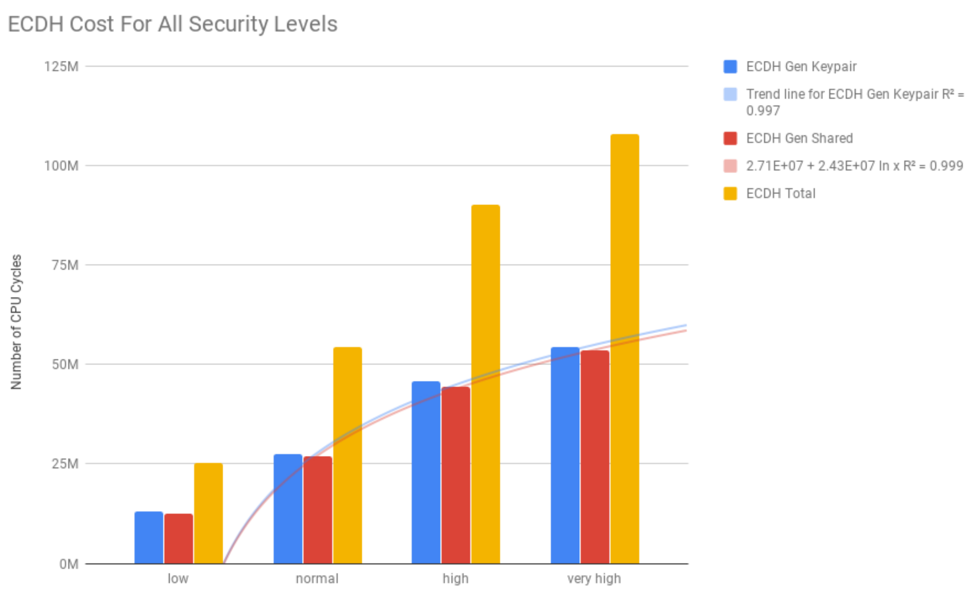


Fig. 21: ECDH operations costs for all security levels

Figure 22 shows the relative cost increase of ECDH operations from the previous security. The cost increase of generating the keypair, the shared secret, and consequently of their sum, is very similar, thus the 3 lines overlap. As expected, the total relative cost increase represents the average of the values of keypair and shared secret generation relative increase. By analyzing the graph, we can see that as the security level increases, the cost increase from the previous security level becomes smaller. The effect decrease on the absolute cost increase can be seen in table A.9. This table shows the absolute increase in the number of CPU cycles from the previous security level. What can be clearly seen in this table is that after the **high** security level, the absolute cost increase starts going down. Although not presented here, those trends continue for higher security levels.

There is a striking resemblance between the cost analysis of ECDH and ECDSA. In both algorithms, the cost increase is logarithmic. In fact, the percentages for ECDSA and ECDH in figures 15 and 22, we can see that the numbers are very similar. This similarity also reflects in the trend that can be seen in tables A.4 and A.9, where in both cases, the absolute cost increase starts going down from the **high** security level and onwards. Those similarities are not a coincidence, but rather a result of the ECC properties of both algorithms, more specifically, a consequence of the multiplication of a scalar by a point on the curve being the core mathematical operation.

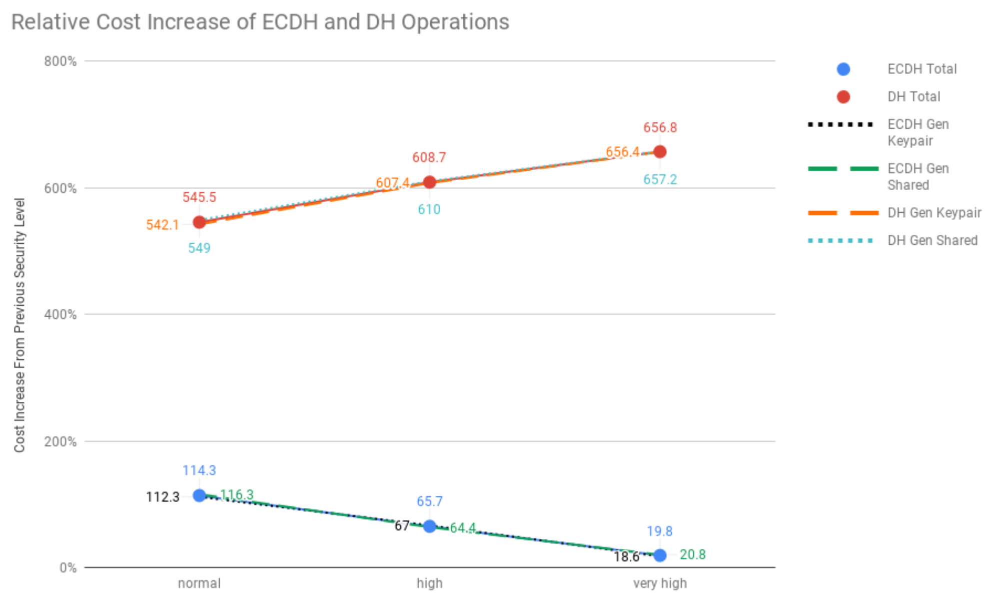


Fig. 22: Relative increase of ECDH and DH operation costs

DH Cost Analysis Similarly to ECDH, in DH two basic operations are performed by each peer: first, a public/private DH keypair is generated, followed by generation of the shared secret, which in TLS, is used as the premaster secret. 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 both of the operations involve modular exponentiations.

We will once again refer to table A.7 for the cost analysis, but this time focusing on DH. Figure 23 shows the costs of DH's operations for all security levels graphically, in logarithmic scale. Just like in ECDH, the total cost is almost evenly divided between the keypair and shared secret generation. This is a consequence of the fact that generating the public/private keys and the shared secret involves the same type of mathematical operations, thus their costs are similar. An analysis of the figure also shows that an exponential trendline is the one that best fits the cost increase for all operations.

Figure 22 shows the relative cost increase of DH operations from the previous security level. The cost increase of generating the keypair, the shared secret, and consequently of their sum, is very similar, thus the 3 lines overlap. As expected, the total relative cost increase represents the average of the values of keypair and shared secret generation relative increase. An analyzis of figure 22 shows as the security level increases, the relative

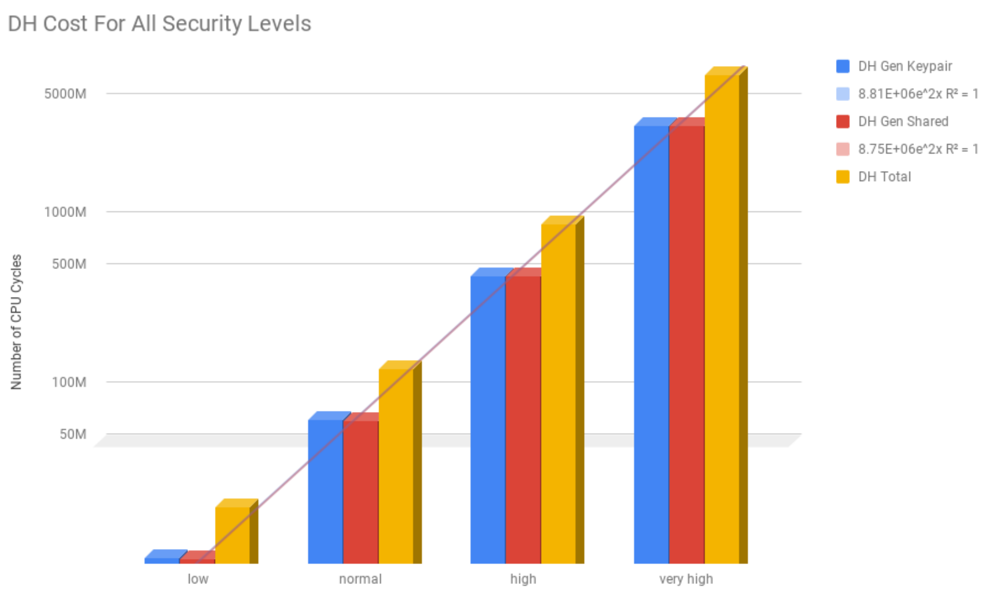


Fig. 23: DH operations costs for all security levels (logarithmic scale)

DH cost increase becomes larger. In fact, this cost increase is exponential. Although not presented here, this holds true for security levels higher than **very high**. The effect of this exponential increase can be seen in table A.10, which shows the absolute cost increase from the previous security level.

Similarly to RSA, DH has an exponential cost increase. This similarity is a consequence of both algorithms having the same mathematical operation at their core: modular exponentiation.

ECDH vs DH Having analyzed the costs of ECDH and DH, we will now compare them and draw some conclusions. Unlike in our comparison of RSA and ECDSA in Section 6.4, 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 analyzing table A.8 and choosing which algorithm has the smallest value for each security level. However, in order to simplify the analysis, we have plotted the table A.7 in figure 24, which presents the data in logarithmic scale.

By looking at figure 24 it is easy to see which algorithm has the smallest costs. 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. Moreover, we can clearly see that for each algorithm, the costs of their operations is very similar, since we have overlapping lines. The logarithmic and exponential properties of ECDH and DH, respectively, are also visible by shape of the lines. Since we are using logarithmic scale for the y axis, the exponential cost growth of DH manifests in the shape of a $y = mx + b$ line.

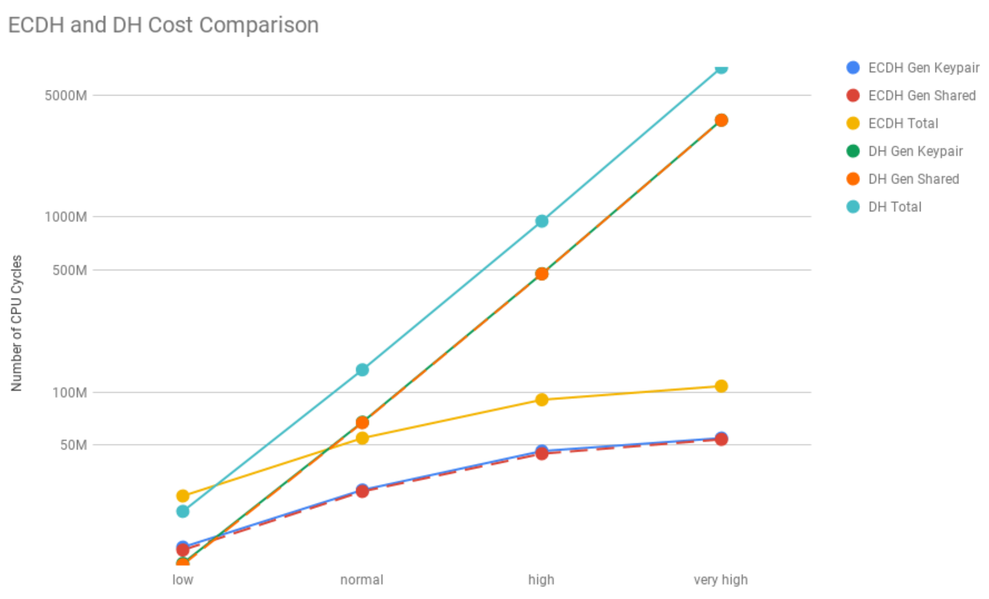


Fig. 24: ECDH and DH cost comparison (logarithmic scale)

Although not presented here, the logarithmic and exponential growth trends for ECDH and DH, respectively, hold true even for security levels higher than **very high**. For security levels above **low**, there are no cost or security advantages of choosing DH over ECDH, for either peer. For this reason, for security level above **low** it makes sense to abandon the use of DH, in favor of ECDH completely. As a benefit of removing the DH code implementation, the storage footprint will be smaller.

The Cost Of PFS In TLS Having analyzed the costs of the algorithms that can be used for PFS, in this section we will describe the cost of this security service for each one of the ciphersuites for the client and the server. In order to avoid confusion in the text that follows, it is important to distinguish the *ECDH* ciphersuite from the ECDH algorithm.

In TLS there are *ECDH* and *ECDHE* ciphersuites. Both use the ECDH algorithm, but only *ECDHE* ciphersuites offer PFS. The *E* in *ECDHE* stands for *ephemeral*. While in *ECHDE* ciphersuites, both the client and the server generate ephemeral ECDH parameters, in *ECDH* ciphersuites at least one of the peer's parameters are **fixed** within its certificate. In our scenario, only the server authenticates to the client, thus in *ECDH* ciphersuites the server's ECDH parameters will be fixed within its certificate, while the client will have to generate new ones for each connection. More specifically, in *ECHDE* ciphersuites, with each new Handshake, both the client and the server generate a new (*i.e.*

ephemeral) public/private ECC key pair that will be used with the ECDH algorithm. On the other hand, in *ECDH* ciphersuites, only the client will generate a new public/private ECC key pair, since the server's public key is fixed within its certificate. Thus in *ECDH* ciphersuites, if the server's long-term shared secret, *i.e.* its private ECC key, is compromised the previous communication's secrets can be computed (if the client's public ECDH parameters that are sent in the clear are recorded), thus also compromising the confidentiality of the past communications. The distinction between the *DH* ciphersuite from DH algorithm is equivalent. Although *DH* (*i.e* non-ephemeral DH) ciphersuites exist in TLS, they are not implemented in *mbedTLS 2.7.0*.

Even though the *ECDH* key exchange does not offer PFS, it is still closely related to the *ECDHE*, 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 tables, we decided to show the cost of PFS and the *ECDH* key exchange in a single one.

Table A.11 shows the cots of PFS and the ECDH key exchange for the client and table A.12 for the server. As a reminder, only *ECDHE* and *DHE* ciphersuites offer PFS. *ECDH-RSA* and *ECDH-ECDSA* ciphersuites (rows highlighted in gray) do not offer PFS and the presented costs are the ones of *ECDH* key exchange. Note, that the highlighted rows are the only ones whose cost differs for the client and the server. For the server, the cost of ECDH key exchange is the cost of generating the shared secret only.

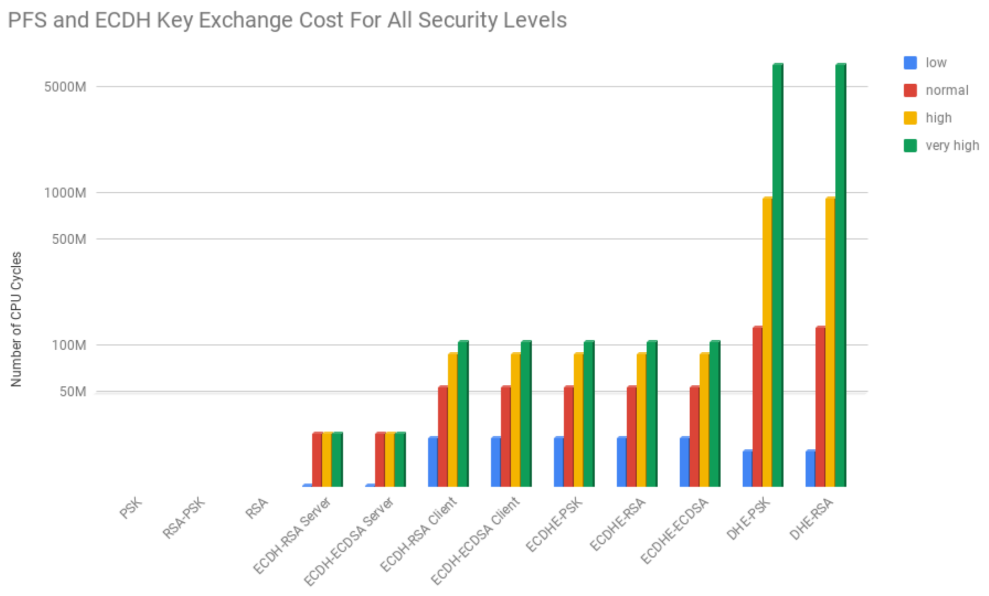


Fig. 25: PFS and *ECDH* key exchange costs (logarithmic scale)

Figure 25 shows the costs of tables A.11 and A.12 graphically, in logarithmic scale. Since the costs of *ECDH* ciphersuites are the only ones that differ between the client and the server, we decided to present both of the tables in a single graph. Notice how the costs of *ECDH-RSA* and *ECDH-ECDSA* are distinguished explicitly for the client and the server. In the figure, we can group the costs into 4 groups, presented in increasing cost order:

1. non-ECDH(E)/DH(E) ciphersuites, which have the total cost of 0
2. *ECDH* server-side ciphersuites
3. *ECDH* client-side and *ECDHE* ciphersuites
4. *DHE* ciphersuite

In the evaluated scenario, only the server authenticates to the client. If mutual authentication was used (*i.e.* with both, the client and server authenticating one to another), the costs of *ECDH* ciphersuites would become the same for the client and the server, with the client having its cost reduced. In the mutual authentication scenario, the client, just like the server, would only have to generate the shared secret, since its public key would be fixed in its certificate. Thus, the client's PFS and *ECDH* costs would be identical to the ones of the server, presented in table A.12. Figure 25 would still look the same, except that the client's *ECDH* ciphersuite costs would become the same as the server's. Thus, the costs could still be separated into the 4 groups mentioned above, but now there would be no *ECDH* client/server side separation:

1. non-ECDH(E)/DH(E) ciphersuites, which have the total cost of 0
2. *ECDH* ciphersuites
3. *ECDHE* ciphersuites
4. *DHE* ciphersuite

6.6 TLS Handshake Cost Analysis

Having analyzed the costs of the authentication and PFS security services in TLS, we will now put everything together and analyze the total cost of the handshake. We have already seen the Handshake cost for the **normal** security level in Section 6.2. In this section, we will present a more detailed analysis of this and the remaining security levels.

Tables A.13 and A.14 show the average handshake cost for each one of the key exchange methods, for the client and server, respectively. The number in parenthesis in each table entry is the standard deviation. The evaluated scenario is the most common one: only the server authenticates to the client.

In *mbedTLS 2.7.0* there is a total of 161 ciphersuites. Table 11 shows the number of ciphersuites per each key exchange method available in *mbedTLS 2.7.0*. For each key exchange method, the average was obtained from a run with each one of the ciphersuites that use that key exchange.

Figures 26 and 27 are a graphical representation of tables A.13 and A.14, respectively. The x axis specifies the security level. The y axis presents the number of CPU cycles used when performing the handshake with the specified key exchange method and security level. The values are presented in logarithmic scale. What can be clearly seen in those graphs, is that even though there are 10 unique key exchange methods, the costs of some of them

Key Exchange	Number of Ciphersuites
PSK	19
RSA-PSK	15
RSA	23
ECDH-RSA	13
ECDH-ECDSA	13
ECDHE-PSK	11
ECDHE-RSA	13
ECDHE-ECDSA	17
DHE-PSK	19
DHE-RSA	18
Total	161

Table 11: Number of ciphersuites per key exchange method in *mbedTLS 2.7.0*

overlap, because they are similar. 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 26 and 27 two or more key exchange methods belong to the same cost group if their lines overlap throughout all security levels. By analyzing figure 26 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*

Similarly, by analyzing figure 27, we can find 8 cost groups for the server side:

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* costs come very close, 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 manifestation of logarithmic (for ECC) vs exponential (for non-ECC) cost increase.

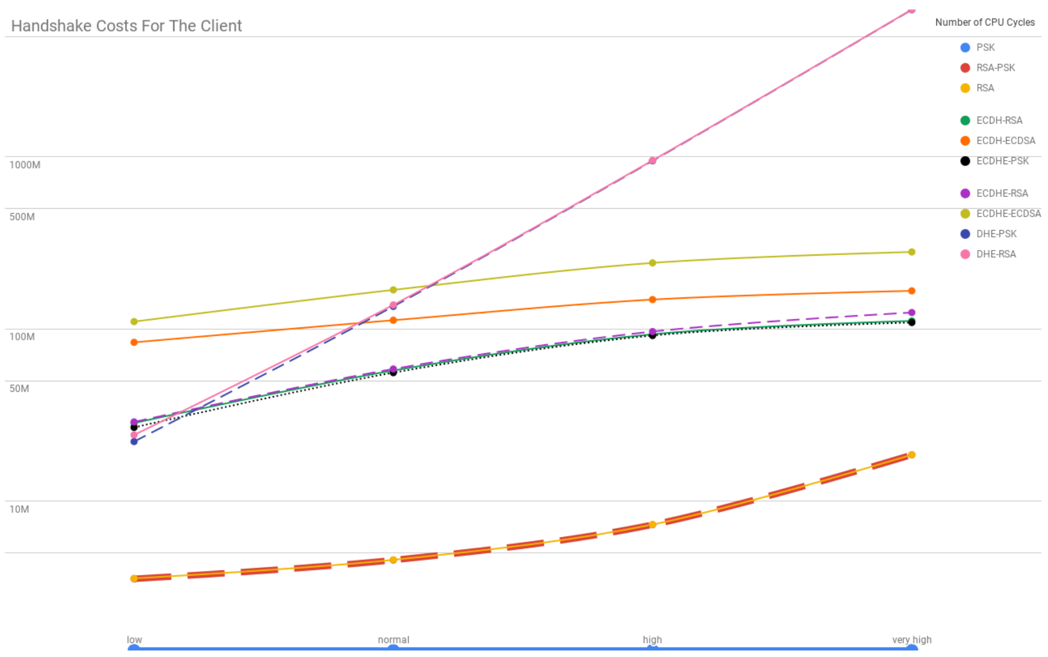


Fig. 26: Client handshake costs for all security levels

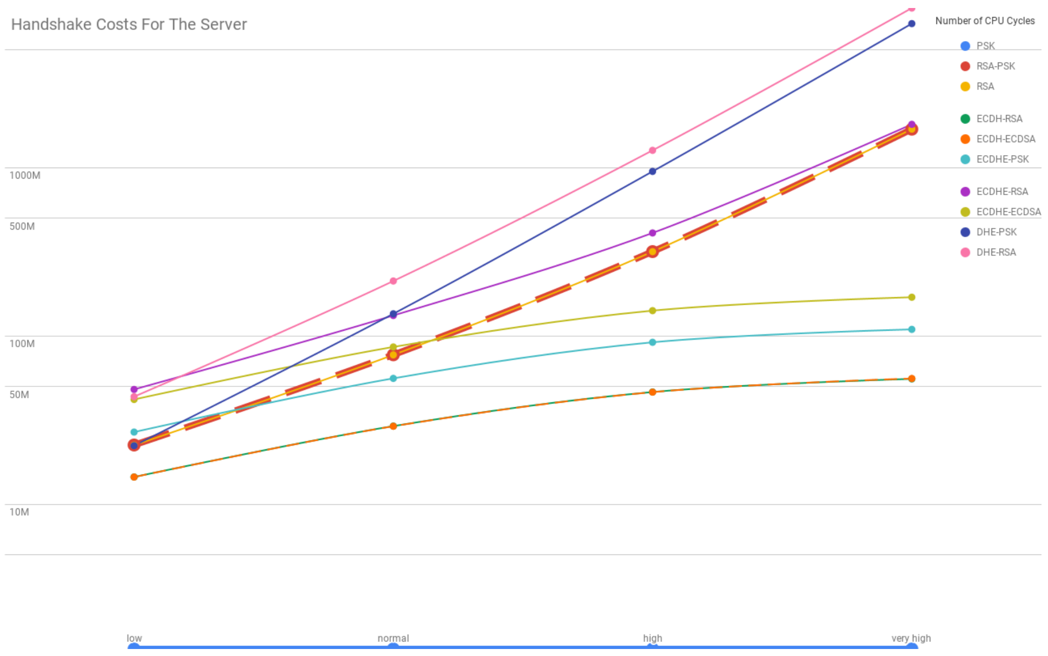


Fig. 27: Server handshake costs for all security levels

Key Exchange	Asymmetric Auth	PFS
PSK	No	No
RSA-PSK	Yes	No
RSA	Yes	No
ECDH-RSA	Yes	No
ECDH-ECDSA	Yes	No
ECDHE-PSK	No	Yes
ECDHE-RSA	Yes	Yes
ECDHE-ECDSA	Yes	Yes
DHE-PSK	No	Yes
DHE-RSA	Yes	Yes

Table 12: Security services offered by each key exchange

Due to the constrained nature of IoT devices, it is important to choose the least costly option for the desired secure connection properties. The cost differences between key exchange methods exist for two reasons. First, different key exchange methods offer different security services. Table 12 shows the security services offered by each key exchange method. Second, as discussed and analyzed throughout Section 6.4, different algorithms can be used to offer the same security service, thus leading to different costs. In order to make a choice of the cheapest key exchange method, we need to answer 4 questions:

1. what is the desired security level?
2. do we want to optimize for the client, the server or both?
3. is asymmetric authentication necessary?
4. is PFS necessary?

Figures 28, 29, 30, 31 are decision trees that help in selecting the cheapest key exchange method for the **low**, **normal**, **high** and **very high** security levels, respectively, according to the security requirements. Each on the decision trees answers the 4 questions mentioned above. 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.

As an example of how to use the decision trees in order to select the cheapest key exchange method, let us consider the scenario where the desired security level is **normal**, asymmetric authentication is required, PFS is not and we would like to minimize the cost for the client. Since the chosen security level is normal, figure 29 will be used. We begin at the *Optimize For* node and go left, to follow the *client* path, since that's the peer that we want to optimize for. Since asymmetric authentication is required, from the *Asymm Auth?* node we follow the *yes* path. This takes us to the *PFS?* node, where we go right to follow the *no* path, since PFS is not required. We finally arrive at a terminal node which lists the suitable key exchange methods, starting with the cheapest one. In our case, the least costly key exchange method is either *RSA* or *RSA-PSK*. Both of them belong to the same cost group for having very similar costs, thus both are presented in the same numeric position. The second least costly option is *ECDH-RSA*, and the most costly one is *ECDH-ECDSA*. Table A.13 can be consulted for precise Handshake costs. After making a decision on which key exchange method(s) is(are) acceptable, either one of ciphersuites that use them can be chosen.

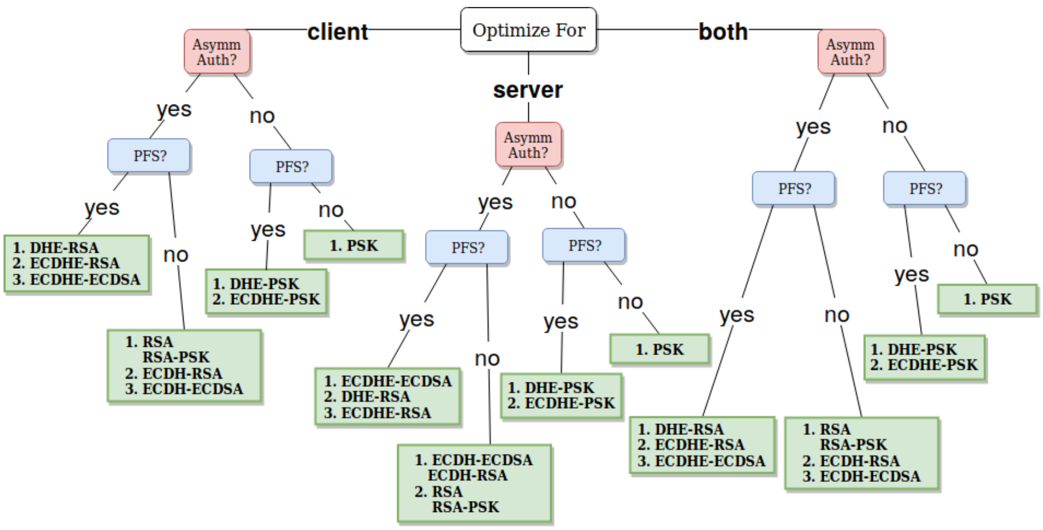


Fig. 28: **low** security level decision tree for the cheapest key exchange

In section 6.4 we analyzed the cost of authentication in TLS and in section 6.5 the cost of PFS in TLS. Tables A.5, A.12, A.11 and A.12 show the authentication and PFS costs for the client and the server, respectively. However, in order to compute the cost of the Handshake for a given key exchange method and security level, it is not enough to sum the corresponding values from the authentication and PFS cost tables. There is also an overhead of establishing the TLS connection and the parsing/record layer costs. As discussed previously, we can consider the cost of the *PSK* handshake as the TLS overhead. The parsing and the record layer costs are additional costs that the peers spend in parsing and reading/writing certain messages not included in the TLS overhead cost. Here, reading/writing does not refer to reading/writing to the network, but rather to the record layer. Namely, it is the cost of creating/verifying the message MAC. We will refer to those costs as *Additional Costs*.

The majority of the *Additional Costs* come from the *Certificate* message. This message is omitted entirely in the *PSK* key exchange. In ciphersuites that use asymmetric authentication, the client has to read the *Certificate* message from the record layer, parse the *der* encoded certificate into internal fields and check the validity of its fields, such as the not valid before/after dates. The server has to write the *Certificate* message to the record layer.

Thus, we can use the following formula to compute the cost of the TLS Handshake cost for a key exchange method and security level: $HandshakeCost = TLSOverhead +$

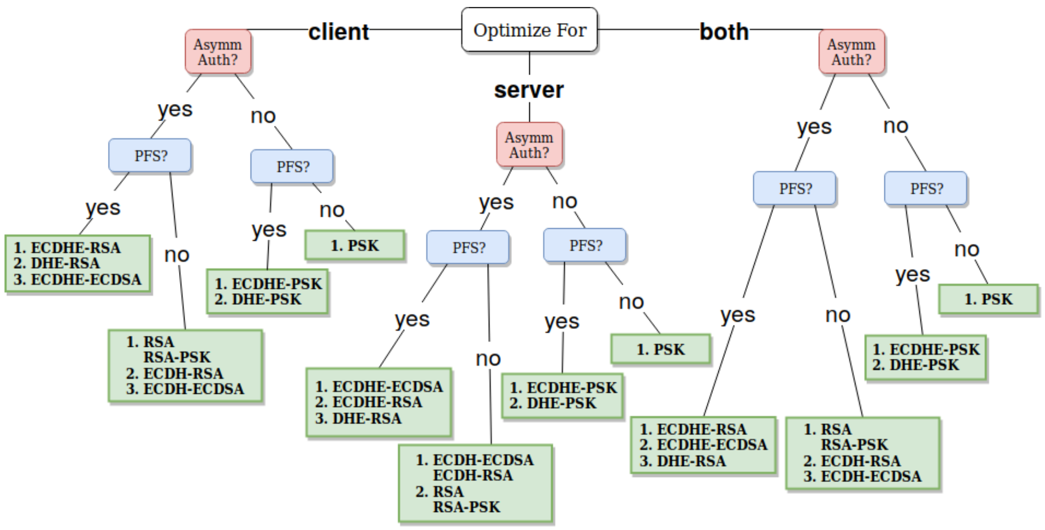


Fig. 29: **normal** security level decision tree for the cheapest key exchange

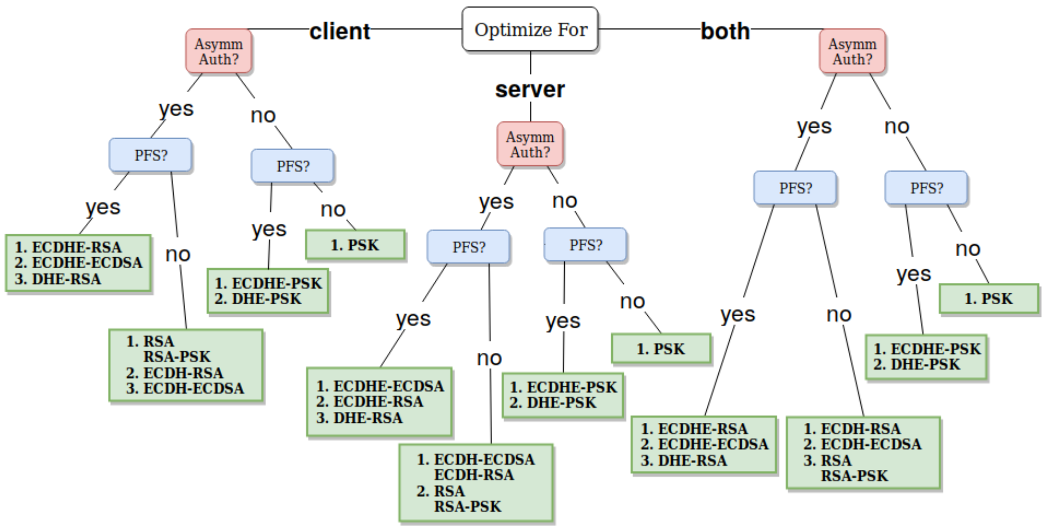


Fig. 30: **high** security level decision tree for the cheapest key exchange

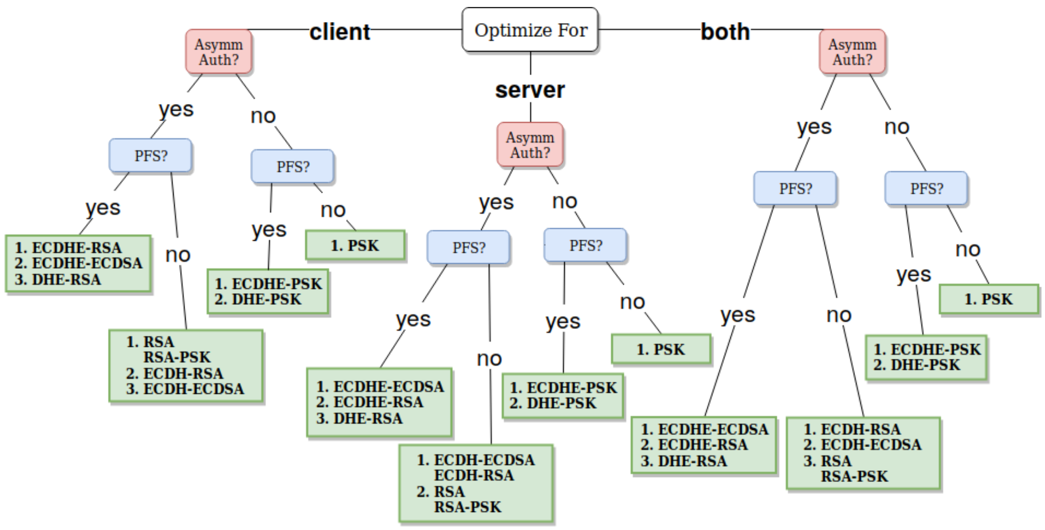

Optimize For
client
server
both
Asymm Auth?
yes
no
PFS?
1. ECDHE-RSA 2. ECDHE-ECDSA 3. DHE-RSA
1. RSA RSA-PSK 2. ECDH-RSA 3. ECDH-ECDSA
1. ECDHE-PSK 2. DHE-PSK
1. PSK
1. ECDHE-ECDSA 2. ECDHE-RSA 3. DHE-RSA
1. ECDH-ECDSA ECDH-RSA 2. RSA RSA-PSK
1. ECDHE-PSK 2. DHE-PSK
1. PSK
1. ECDHE-ECDSA 2. ECDHE-RSA 3. DHE-RSA
1. ECDH-RSA 2. ECDH-ECDSA 3. RSA RSA-PSK
1. ECDHE-PSK 2. DHE-PSK
1. PSK

Fig. 31: **very high** security level decision tree for the cheapest key exchange

$AuthCost + PFSCost + AdditionalCosts$. The *TLS Overhead* is the cost of the *PSK* key exchange and can be obtained from table A.13 for the client and from table A.14 for the server. *Auth Cost* is the cost of authentication and can be obtained from A.5 for the client and from table A.6 for the server. *PFS* cost is the cost of PFS and can be obtained from table A.11 for the client and from table A.12 for the server. The majority of *Additional Costs* can be obtained from table 13, which presents the most relevant costs when parsing the *Certificate* message.

The certificate is either signed with RSA or ECDSA. In *mbedTLS 2.7.0* there are 31 ciphersuites that use RSA-signed certificates and 17 ciphersuites that use ECDSA-signed certificates. Table 13 contains the cost of parsing the certificate for each algorithm and security level. This costs do not include checking the certificate's signature, but rather reading the certificate from the record layer, parsing the *der*-encoded certificate into internal fields, checking the validity of each field and creating a hash of the fields. The presented values an average of 31 executions for RSA and 17 for ECDSA, each one with a different ciphersuite. The numbers in parenthesis are standard deviation. An analysis of the table shows that the cost is very similar across all algorithms and security levels.

Security Level / Algorithm	**low**	**normal**	**high**	**very high**
RSA	511728 (45373)	544227 (47594)	607542 (51576)	735237 (59853)
ECDSA	548841 (45064)	662348 (45175)	666866 (45923)	704135 (48247)

Table 13: Additional costs of parsing the certificate

As an example, we will compute the client's Handshake cost for the **high** security level with *ECHDE-RSA* key exchange, from the TLS security service costs that we presented in sections 6.4 and 6.5. First, we will need to get the *TLSOverhead* value. We can obtain in from the entry located at the *ECDHE-RSA* row and *high* column in table A.13: 1355005. *AuthCost* can be obtained from the *ECDHE-RSA* row and *high* column in table A.5: 4413164. Similarly *PFSCost* can be obtained from table A.11, where the *PSK* row and *high* column intersect: 90231688. For the additional certificate parsing costs, we will consult table 13. Since the certificate is an RSA signed one, we will use the value from *RSA* row and *high* column: 607542. All that's left now is insert those values in the formula: $HandshakeCost = TLSOverhead + AuthCost + PFSCost + AdditionalCosts = 1355005 + 4413164 + 90231688 + 607542 = 96607399$.

This value was obtained by decomposing the TLS Handshake into the security services and adding up their costs. As we can see, the cost computed using the formula is very close to the actual Handshake cost when evaluated as a whole, which can be found in in table A.13 (*ECDHE-RSA* row and *high* column): 96730901. The difference of 123502 (96730901−96607399) CPU cycles comes from other additional costs, such as reading and writing slightly larger messages, namely the *ServerKeyExchange* and the *ClientKeyExchange*.

6.7 Confidentiality and Integrity Cost Analysis

Having analyzed the cost of authentication and PFS, we will now briefly analyze the cost of the confidentiality and integrity services. In TLS, it does not make sense to analyze both of the services separately, since they're offered as a whole as a part of the ciphersuite. For this reason, we will analyze them together.

In *mbedTLS 2.7.0* implements 13 non-null unique encryption algorithms and 4 unique hash functions. There are 6 *AES* algorithms, 4 *CAMELLIA* algorithms, 1 *DES* algorithm, 1 *3DES* algorithm and 1 *RC4* algorithm. The 4 available hash functions are *SHA-1*, *SHA-256*, *SHA-384* and *MD5*. For non-AEAD ciphers, the combination of a symmetric encryption algorithm and a hash function is used to provide the security services of confidentiality and integrity. For AEAD ciphers the symmetric encryption algorithm provides both, confidentiality and integrity, so there is no need to use an additional hash function.

Figure 32 shows the cost of encryption and MAC creation using each one of the 26 symmetric encryption algorithm/hash function pairs available in *mbedTLS 2.7.0*. As expected, their cost grows linearly with the amount of encrypted bytes. An analysis of the graph 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. The cost of integrity without encryption can be seen by looking at the null encryption ciphersuites: *NULL-MD5*, *NULL-SHA*, *NULL-SHA256* and *NULL-SHA384*. For both, AEAD algorithms and non-AEAD algorithms combined with a hash function, the majority of the cost comes from data encryption.

In Section 6.6 we analyzed the cost of the handshake. However, it only makes sense to heavily optimize handshake if the amount of transmitted data is small. This is, in fact, true for many IoT devices. But what exactly is a *small* amount of data?

In order to answer those questions 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). We selected those algorithms because they were among the cheapest

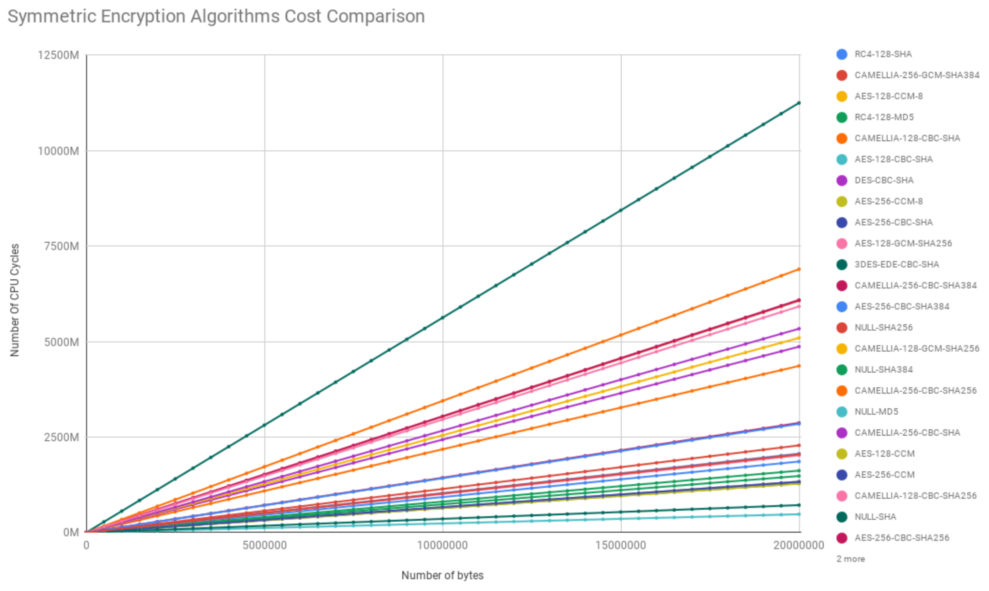


Fig. 32: Confidentiality and integrity cost with different algorithms

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[86]. For each one of the algorithms, we encrypted data sizes from starting from 1 byte up to 16250000 bytes (approximately 15.5 mega bytes), with 100 byte increments in each run. Thus, a total of 162500 runs was made with each algorithm. It is expected for encryption costs to grow linearly with the amount of encrypted data, thus we applied linear regression to the collected metrics in order in order to obtain a formula that approximates the cost of encryption for any amount of data. 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 we cam see. the cost of *AES-256-GCM* is slightly larger than of *AES-128-GCM*.

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. For example, let's compute this number for the client, when it performs an *ECDHE-ECDSA* handshake at the *normal* security level. First, we obtain the number of CPU cycles used to perform the handshake from table A.13: 168954007. After that we simply replace $NumCC$ with that value in the formula and solve for $NumBytes$: $168954007 = 104 * NumBytes + 22680 \Leftrightarrow NumBytes \approx 1624340$. Thus, the client needs to send 1624340 bytes ($\approx$ 1.5 mega bytes) for the encryption costs to equate the *ECDHE-ECDSA* handshake cots.

Figures 33, 34 and 35 help to visualize when when the encryption/decryption costs equate the handshake costs for the **low**, **normal** and **high** security levels, respectively. Figures 36, 37 and 38 show the same information for the server. We did not present the graphs for the *very high* security level, because the overwhelming cost of some key exchange methods made the visual analysis of the graph hard (due to scale).

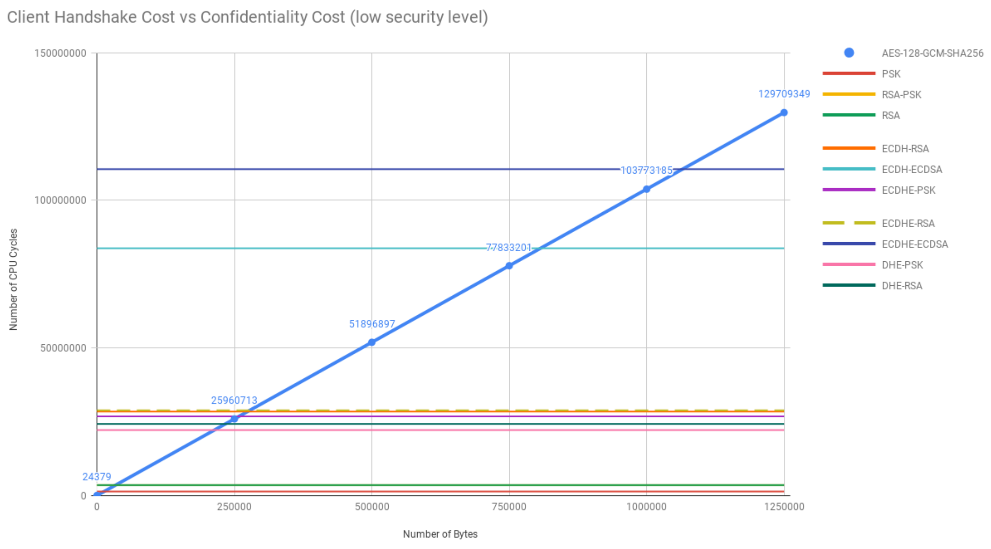


Fig. 33: Client handshake vs confidentiality cost for the **low** security level

6.8 Time Measurments with PAPI and Their Comparison To The Estimated Values

We also used PAPI to measure time. In this section, we will compare the estimates of the number of CPU cycles obtained with *valgrind* with the time values obtained with PAPI. In order to obtain the time metrics with PAPI we used the same approach and the same number of runs as with *valgrind*. The

The iron law of processor performance[87] states that the time taken by a program execution is proportional to the number of instuctions (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 taht 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.

PAPI Handhsake Time Cost Analysis Figures 39 and 40 show the time taken to preform the handshake with each one of the ciphersuites, for each one of the security levels. The x axis specifies the security level. The y axis specifies the time taken in microseconds,

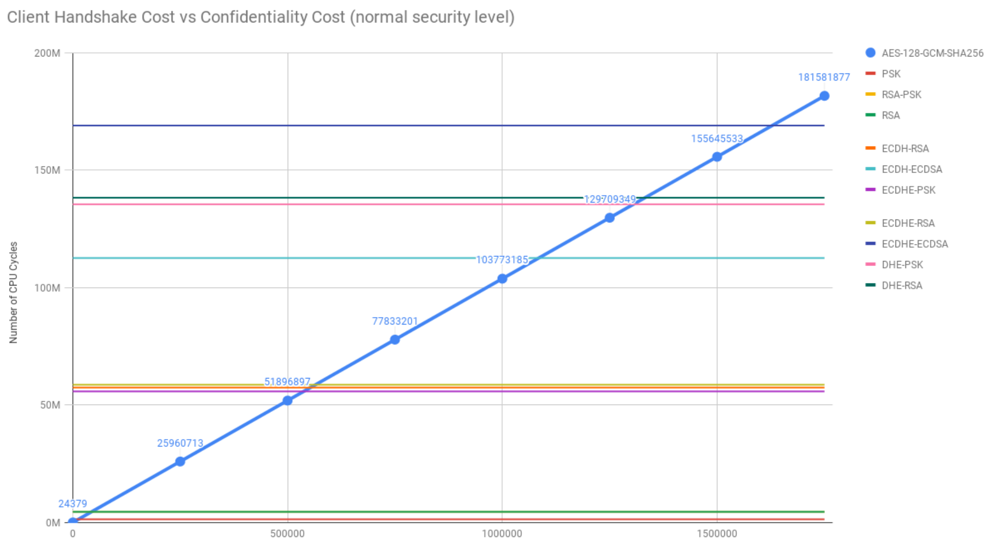


Fig. 34: Client handshake vs confidentiality cost for the **normal** security level

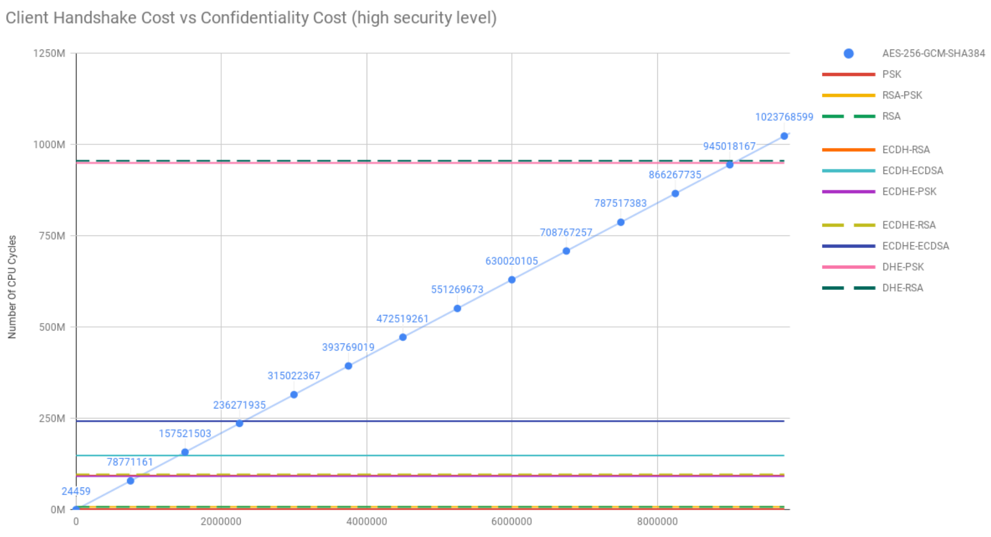


Fig. 35: Client handshake vs confidentiality cost for the **high** security level

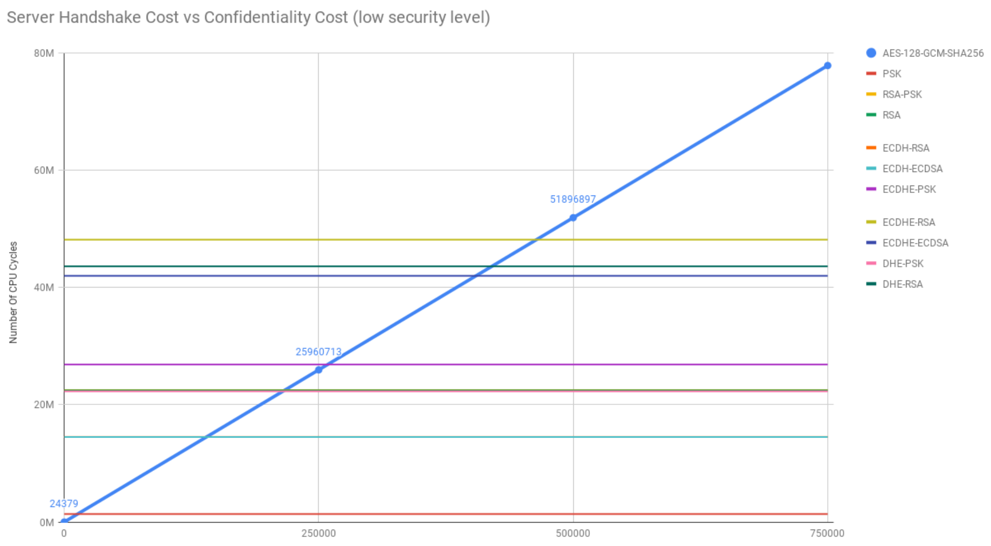


Fig. 36: Server handshake vs confidentiality cost for the **low** security level

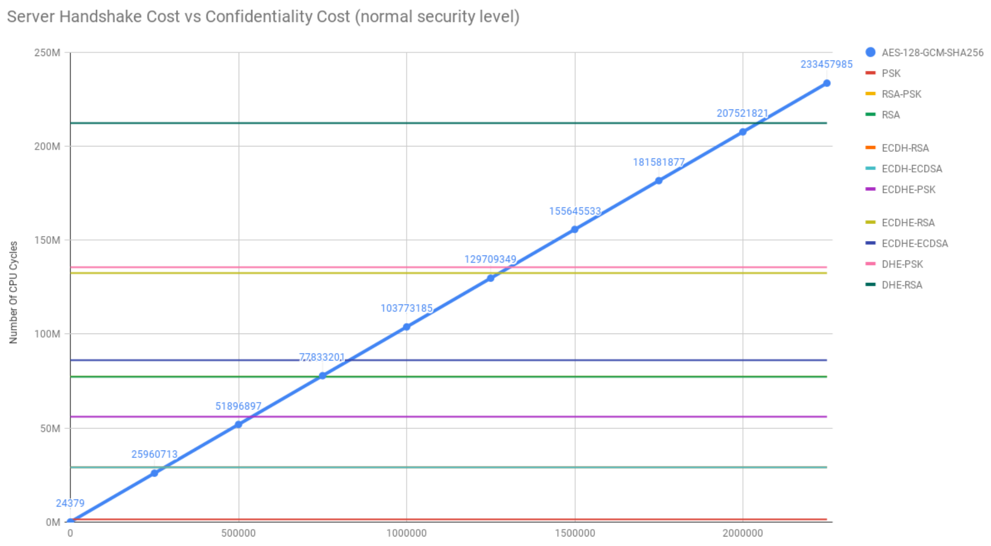


Fig. 37: Server handshake vs confidentiality cost for the **normal** security level

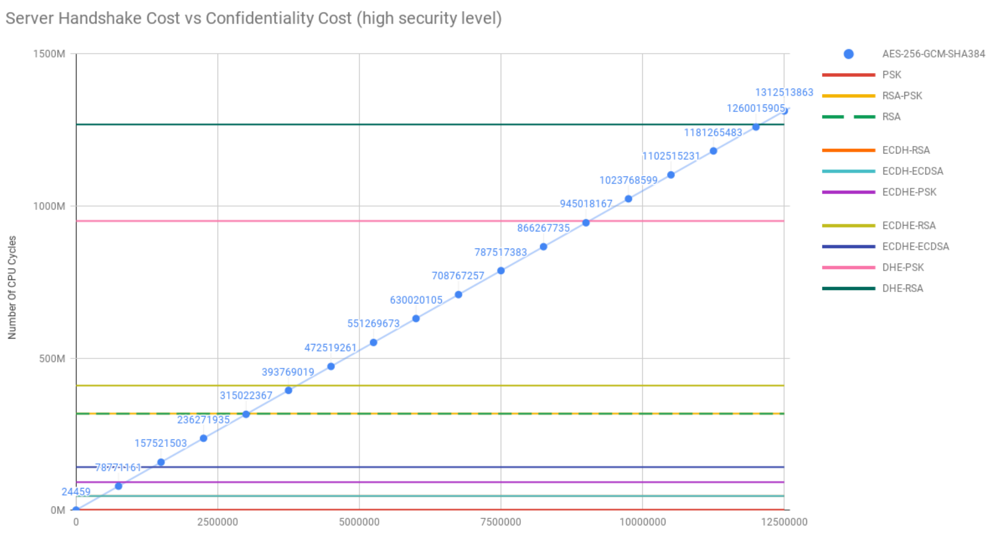


Fig. 38: Server handshake vs confidentiality cost for the **high** security level

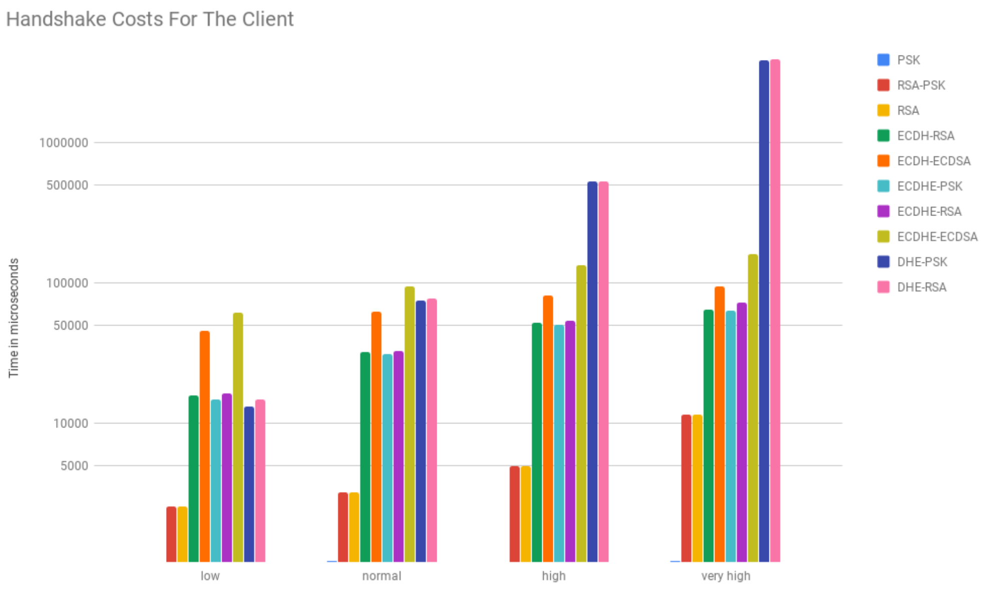


Fig. 39: Client handshake cost in microseconds for all security levels (logarithmic scale)

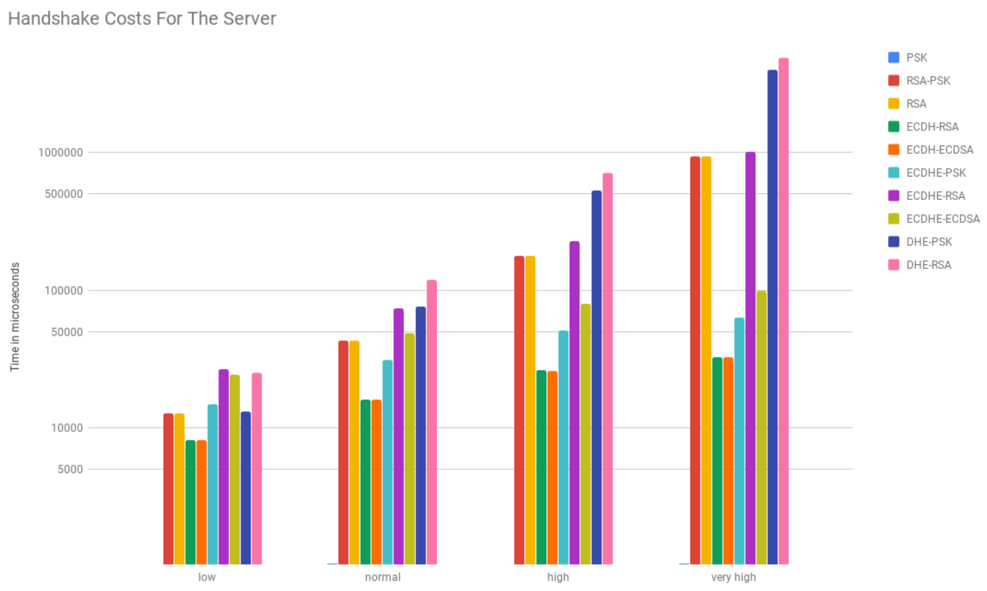


Fig. 40: Server handshake cost in microseconds for all security levels (logarithmic scale)

when performing the handshake with the specified key exchange method and security level. The values are presented in logarithmic scale. A comparison of 39 to 26 and of 40 to 40 shows that they look identical, differing only on the y values. This holds true not only for the overally *handshake* cost, but for all other costs that we analyzed as well. The ratio between each realted set of operations is also similar.

Comparing Valgrind and PAPI Ratios Figure 41 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 purpusefuly devided 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, *callgrind*'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 *callgrind* 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. *callgrind*'s RSA's sign/verify ratio is 24.87% larger than PAPI's for the *low* security level this differnce is only 3.6%. Following the 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.

Given that the ratios described above are similar, the algorithm cost comparisons for the various security levels remain valid for PAPI's time values as well. Consequently, this

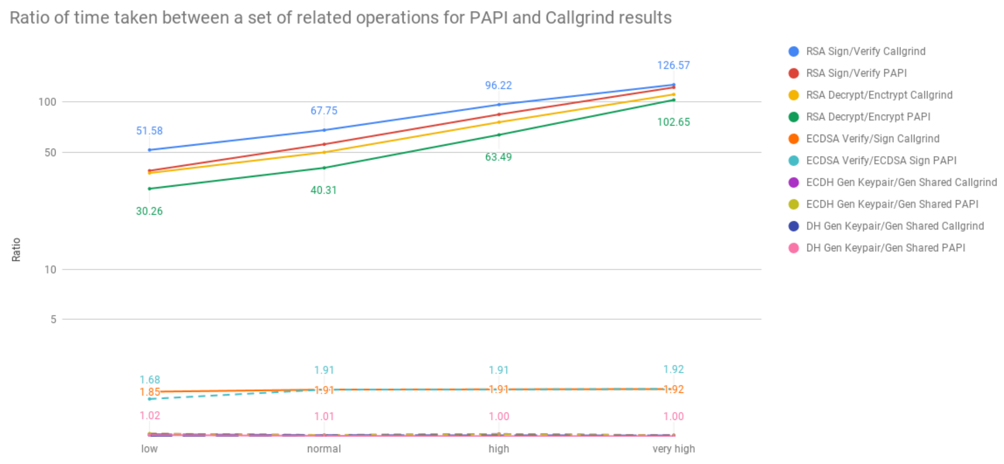


Fig. 41: Ratio of time taken between a set of related operations for PAPI and *callgrind* (logarithmic scale)

is also true for the security services, since they are composed of those algirithms. The graphs for the security services and algorithms look very similar the *valgrind* ones, for this reason, they have been included in the annex without any further analysis.

6.9 Discussion

In this section evaluated the cost of security services of authentication, PFS, confidentiality and integrity in TLS. To do this, we defined 4 security levels and analyzed the costs for each one. The majority of analysis was focused on the handshake and the security services that it offers. We did performed the analysis at two levels: first by using the estimated number of CPU cycles obtained with *valgrind*, and then using the execution time values obtained with PAPI.

In TLS there are two ways of doing authentication: PSK or asymmetric cryptography. We showed that the cost of PSK authentication is essentially 0 and we can consider the cost of performing the handshake with a *PSK* ciphersuite can be considered as the TLS overhead.

In TLS, authentication with asymmetric cryptography can be done in two ways: either by using RSA of ECDSA. We began by analyzing the costs of those two algorithms. We concluded that RSA was faster at public key operations (*i.e.* signature verification) and ECDSA was faster at private key operations (*i.e.* signature creation). An analyzes of the algorithm's costs showed that as the security level increases, while RSA's cost increase exponentially, ECDSA's increase logarithmically. This is a result of the ECC properties of the algorithm.

After evaluating the costs of individual algorithms that can be used to provide asymmetric authentication in TLS, we used those results to compute the authentication cost in TLS, for each one of its key exchange methods.

After analyzing the cost of authentication, we turned our attention to PFS. In TLS there are two ways of achieving PFS: either by using ECDH or DH. We decomposed each algorithm into individual steps and analyzed their cost. As the security level increases, the cost of DH grows exponentially, while the cost of ECDH logarithmically. In that sense, DH is similar to RSA and ECDH to ECDSA. This similarity is a consequence of the fact, the underlying mathematical operation is the same for DH and RSA (modular exponentiation), and for ECDH and ECDSA (multiplication of a scalar by a point on the elliptic curve). After evaluating the costs of the algorithms that can be used to provide PFS in TLS, we used those results to arrive at PFS cost for each on of the key exchange methods.

Having computed the costs of authentication and PFS we were now ready to compute the cost of the TLS handshake as a whole. We presented a formula that decomposed the handshake costs its individual parts: $HandshakeCost = TLSOverhead + AuthCost + PFSCost + AdditionalCosts$. We compared the costs obtained from the formula to the actual handshake cost values that we acquired by profiling various client-server executions and showed that both are similar.

Finally, we analyzed the costs of confidentiality and integrity. First, we compared the costs of all of the symmetric algorithm/hash function combinations available in *mbedTLS 2.7.0*. As expected, AEAD algorithms preformed the best among the block ciphers. After that we chose two AEAD ciphers and analyzed how much data it would be necessary to encrypt with them to equate the handshake costs.

The analysis the PAPI results showed us that the estimates obtained with *valgrind* do reflect the real world values. Not only the higher-level results are similar in comparison to one another, but at the lower level, the ratios between the algorithms used by the security services also are. As the *The iron law of processor performance* suggested, the time values can be approximated as the estimated number of CPU cycles multiplied by a factor smaller than 1. Thus, the estimates from the previous sections can be used for relative comparisons between the costs of security services and security levels in TLS, thus allowing to achieve the goals of this work. The time costs presented in this seciton and in the Annex provide concrete, real-world reference values for the costs of the various parts of the TLS protocol.

7 Conclusion

The lack of security in IoT is a serious issue that can lead to a high monetary costs, especially when botnets infect the devices. Recent attacks clearly show that serious damage can be caused [3]. An old saying attributed to the US National Security Agency (NSA) states that "Attacks always get better; they never get worse". Combined with the fact that the number of IoT devices is growing at a high pace, without any major improvements to their security, makes it clear that it is fundamental for this issue to be addressed.

TLS is one of the most used communication security protocols in the world. It 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 some of the security services on a per-connection basis.

While TLS does have many configurations, not all of them can be used with the IoT, due to the constrained nature of the devices. In many cases it is necessary to make security/cost trade-offs. In order to be able to make them, it is important to know the costs of each security service. Current work offers no such information. Thus, a software developer wishing to use TLS for connection security in a constrained environment does not have a reference to go to.

In our work, we decomposed TLS into individual parts and evaluated the cost of each security service. We have evaluated the cost of every TLS configuration available in *mbedTLS 2.7.0* and the underlying algorithms. The herein presented results can be used to make informed decisions about the security/cost trade-offs, specific to the environment.

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. The formula used to obtain the CPU cycle count estimate can also be changed to one's needs.

Future Work In our work we obtained and analyzed a large number of metrics obtained with *callgrind*. While *callgrind* provides only an estiamtes 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 mertics 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 factors such as a lower clock frequency, memory and cache size. Thus, it would be interesting to measure time on an glsiot device, since the results will differ.

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 interesting conclusions, such as: *Using the TLS configuration X would reduce the device's battery life by Y days.*

References

1. • iot: number of connected devices worldwide 2012-2025. `https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/`. (Accessed on 11/18/2018).
2. SMALT - The Smart Home Device That Elevates Your Dining Experience. Electronic Gadget, 2017. (Accessed on 01/05/2018).
3. Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. 2017.
4. Brian Krebs. Reaper: Calm before the iot security storm? — krebs on security. `https://krebsonsecurity.com/2017/10/reaper-calm-before-the-iot-security-storm/`, 10 2017. (Accessed on 01/04/2018).
5. Addition to tls 1.3 contributors list. `https://github.com/tlswg/tls13-spec/commit/43461876882a60251ecf24fb097f0ce2d7be4745`. (Accessed on 01/11/2018).

6. E. Rescorla. The transport layer security (tls) protocol version 1.3. RFC 8446, RFC Editor, August 2018.
7. Commits by iluxonchik tlswg/dtls13-spec. `https://github.com/tlswg/dtls13-spec/commits?author=iluxonchik`. (Accessed on 11/18/2018).
8. Nvd - cve-2018-1000520. `https://nvd.nist.gov/vuln/detail/CVE-2018-1000520`. (Accessed on 10/10/2018).
9. 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).
10. 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).
11. Shahid Raza, Ludwig Seitz, Denis Sitenkov, and Göran 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.
12. Carsten Bormann, Simon Lemay, Hannes Tschofenig, Klaus Hartke, Bill Silverajan, and Brian 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`.
13. mbed TLS (formely PolarSSL): SSL/TLS Library For Embedded Devices. (Accessed on 12/19/2017).
14. Qualys ssl labs - ssl pulse. `https://www.ssllabs.com/ssl-pulse/`. (Accessed on 09/08/2018).
15. Elaine Barker. Recommendation for Key Management, Part 1: General. *NIST special publication*, 2016.
16. Bundesamt fur Sicherheit in der Informationstechnik (BSI). Kryptographische Verfahren: Empfehlungen und Schlüssellängen, Version 2017-01. 2017.
17. C. Bormann, M. Ersue, and A. Keranen. Terminology for Constrained-Node Networks. RFC 7228, RFC Editor, May 2014. `http://www.rfc-editor.org/rfc/rfc7228.txt`.
18. D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, RFC Editor, May 2008. `http://www.rfc-editor.org/rfc/rfc5280.txt`.
19. T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, RFC Editor, August 2008. `http://www.rfc-editor.org/rfc/rfc5246.txt`.
20. A. Freier, P. Karlton, and P. Kocher. The secure sockets layer (ssl) protocol version 3.0. RFC 6101, RFC Editor, August 2011. `http://www.rfc-editor.org/rfc/rfc6101.txt`.
21. J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077, RFC Editor, January 2008. `http://www.rfc-editor.org/rfc/rfc5077.txt`.
22. Stephen Checkoway, Jacob Maskiewicz, Christina Garman, Joshua Fried, Shaanan Cohney, Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, and Hovav Shacham. A systematic analysis of the juniper dual ec incident. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 468–479. ACM, 2016.
23. S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492, RFC Editor, May 2006. `http://www.rfc-editor.org/rfc/rfc4492.txt`.
24. P. Wouters, H. Tschofenig, J. Gilmore, S. Weiler, and T. Kivinen. Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7250, RFC Editor, June 2014.
25. S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport Layer Security (TLS) Extensions. RFC 4366, RFC Editor, April 2006. `http://www.rfc-editor.org/rfc/rfc4366.txt`.
26. Tim Dierks and Christopher Allen. The TLS Protocol Version 1.0. RFC 2246, RFC Editor, January 1999. `http://www.rfc-editor.org/rfc/rfc2246.txt`.

27. Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. Internet-Draft draft-ietf-tls-tls13-22, IETF Secretariat, November 2017. `http://www.ietf.org/internet-drafts/draft-ietf-tls-tls13-22.txt`.
28. H. Krawczyk and P. Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, RFC Editor, May 2010. `http://www.rfc-editor.org/rfc/rfc5869.txt`.
29. E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, RFC Editor, January 2012. `http://www.rfc-editor.org/rfc/rfc6347.txt`.
30. Eric Rescorla, Hannes Tschofenig, and Nagendra Modadugu. The Datagram Transport Layer Security (DTLS) Protocol Version 1.3. Internet-Draft draft-ietf-tls-dtls13-22, IETF Secretariat, November 2017. `http://www.ietf.org/internet-drafts/draft-ietf-tls-dtls13-22.txt`.
31. Alex Biryukov and Léo Paul Perrin. State of the art in lightweight symmetric cryptography. 2017.
32. Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin O'Flynn. Iot goes nuclear: Creating a zigbee chain reaction. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 195–212. IEEE, 2017.
33. Masanobu Katagi and Shiho Moriai. Lightweight cryptography for the internet of things. *Sony Corporation*, pages 7–10, 2012.
34. C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5). RFC 4120, RFC Editor, July 2005. `http://www.rfc-editor.org/rfc/rfc4120.txt`.
35. P. Eronen and H. Tschofenig. Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). RFC 4279, RFC Editor, December 2005. `http://www.rfc-editor.org/rfc/rfc4279.txt`.
36. G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944, RFC Editor, September 2007. `http://www.rfc-editor.org/rfc/rfc4944.txt`.
37. J. Hui and P. Thubert. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. RFC 6282, RFC Editor, September 2011. `http://www.rfc-editor.org/rfc/rfc6282.txt`.
38. Shahid Raza, Daniele Trabalza, and Thiemo Voigt. 6lowpan compressed dtls for coap. In *Distributed Computing in Sensor Systems (DCOSS), 2012 IEEE 8th International Conference on*, pages 287–289. IEEE, 2012.
39. C. Bormann. 6LoWPAN-GHC: Generic Header Compression for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs). RFC 7400, RFC Editor, November 2014.
40. Ahmed Ayadi, David Ros, and Laurent Toutain. TCP header compression for 6LoWPAN. Internet-Draft draft-aayadi-6lowpan-tcphc-01, IETF Secretariat, October 2010. `http://www.ietf.org/internet-drafts/draft-aayadi-6lowpan-tcphc-01.txt`.
41. Shahid Raza, Hossein Shafagh, Kasun Hewage, René Hummen, and Thiemo Voigt. Lithe: Lightweight secure coap for the internet of things. *IEEE Sensors Journal*, 13(10):3711–3720, 2013.
42. C. Bormann and Z. Shelby. Block-Wise Transfers in the Constrained Application Protocol (CoAP). RFC 7959, RFC Editor, August 2016.
43. Contiki: The open source operating system for the internet of things. `http://www.contiki-os.org/`. (Accessed on 01/03/2018); HTTPS link was not used because it is broken.
44. Angelo Capossele, Valerio Cervo, Gianluca De Cicco, and Chiara Petrioli. Security as a coap resource: an optimized dtls implementation for the iot. In *Communications (ICC), 2015 IEEE International Conference on*, pages 549–554. IEEE, 2015.
45. Sye Keoh, Sandeep Kumar, and Zach Shelby. Profiling of DTLS for CoAP-based IoT Applications. Internet-Draft draft-keoh-dtls-profile-iot-00, IETF Secretariat, June 2013. `http://www.ietf.org/internet-drafts/draft-keoh-dtls-profile-iot-00.txt`.
46. Z. Shelby, K. Hartke, and C. Bormann. The constrained application protocol (coap). RFC 7252, RFC Editor, June 2014. `http://www.rfc-editor.org/rfc/rfc7252.txt`.

47. Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and rsa on 8-bit cpus. In *CHES*, volume 4, pages 119–132. Springer, 2004.
48. Giuseppe Ateniese, Giuseppe Bianchi, Angelo Capossele, and Chiara Petrioli. Low-cost standard signatures in wireless sensor networks: a case for reviving pre-computation techniques? In *Proceedings of NDSS 2013*, 2013.
49. H. Tschofenig and T. Fossati. Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things. RFC 7925, RFC Editor, July 2016.
50. D. Simon, B. Aboba, and R. Hurst. The EAP-TLS Authentication Protocol. RFC 5216, RFC Editor, March 2008. `http://www.rfc-editor.org/rfc/rfc5216.txt`.
51. D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, RFC Editor, May 2008. `http://www.rfc-editor.org/rfc/rfc5280.txt`.
52. S. Santesson and H. Tschofenig. Transport Layer Security (TLS) Cached Information Extension. RFC 7924, RFC Editor, July 2016.
53. S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 6960, RFC Editor, June 2013. `http://www.rfc-editor.org/rfc/rfc6960.txt`.
54. D. Eastlake. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066, RFC Editor, January 2011. `http://www.rfc-editor.org/rfc/rfc6066.txt`.
55. Y. Sheffer, R. Holz, and P. Saint-Andre. Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). BCP 195, RFC Editor, May 2015. `http://www.rfc-editor.org/rfc/rfc7525.txt`.
56. Amichai Shulman Tal Be'ery. A perfect CRIME? Only TIME will tell. `https://media.blackhat.com/eu-13/briefings/Beery/bh-eu-13-a-perfect-crime-beery-wp.pdf`, 2013. (Accessed on 01/04/2018).
57. R. Seggelmann, M. Tuexen, and M. Williams. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. RFC 6520, RFC Editor, February 2012. `http://www.rfc-editor.org/rfc/rfc6520.txt`.
58. D. Eastlake, J. Schiller, and S. Crocker. Randomness Requirements for Security. BCP 106, RFC Editor, June 2005. `http://www.rfc-editor.org/rfc/rfc4086.txt`.
59. P. Gutmann. Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7366, RFC Editor, September 2014. `http://www.rfc-editor.org/rfc/rfc7366.txt`.
60. K. Bhargavan, A. Delignat-Lavaud, A. Pironti, A. Langley, and M. Ray. Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension. RFC 7627, RFC Editor, September 2015.
61. Karthikeyan Bhargavan, Antoine Delignat Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over tls. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 98–113. IEEE, 2014.
62. E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746, RFC Editor, February 2010. `http://www.rfc-editor.org/rfc/rfc5746.txt`.
63. A. Popov. Prohibiting RC4 Cipher Suites. RFC 7465, RFC Editor, February 2015. `http://www.rfc-editor.org/rfc/rfc7465.txt`.
64. LightweightM2M V1.0 Overview, 2017. (Accessed on 12/30/2017).
65. Brendan Moran, Milosch Meriac, and Hannes Tschofenig. A Firmware Update Architecture for Internet of Things Devices. Internet-Draft draft-moran-suit-architecture-00, IETF Secretariat, October 2017. `http://www.ietf.org/internet-drafts/draft-moran-suit-architecture-00.txt`.

66. IANA. Transport layer security (tls) parameters. `https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml`. (Accessed on 08/20/2018).
67. Supported ssl / tls ciphersuites - mbedtls. `https://tls.mbed.org/supported-ssl-ciphersuites`. (Accessed on 08/20/2018).
68. Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010.
69. globalconfig.cpp source file. `https://api.kde.org/4.14-api/kdesdk-apidocs/kcachegrind/html/globalconfig_8cpp_source.html#l00073`. (Accessed on 08/22/2018).
70. James Charles, Preet Jassi, Narayan S Ananth, Abbas Sadat, and Alexandra Fedorova. Evaluation of the intel® core^{TM} i7 turbo boost feature. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 188–197. IEEE, 2009.
71. Callgrind cpu cycles counter. `https://github.com/iluxonchik/callgrind-cpu-cycles-counter`. (Accessed on 10/14/2018).
72. Profile and collect papi metrics. `https://github.com/iluxonchik/python-papi-profiler`. (Accessed on 04/22/2019).
73. Unique encryption ciphersuites. `https://github.com/iluxonchik/mbedtls-unique-encryption-ciphersuites`. (Accessed on 10/14/2018).
74. Callgrind profiling data analyzer. `https://github.com/iluxonchik/profiling-data-analyzer`. (Accessed on 10/14/2018).
75. Mbedtls code used for valgrind and papi profilings. `https://github.com/iluxonchik/mbedtls-playground`. (Accessed on 04/22/2019).
76. Aamer Nadeem and M Younus Javed. A performance comparison of data encryption algorithms. In *Information and communication technologies, 2005. ICICT 2005. First international conference on*, pages 84–89. IEEE, 2005.
77. Sheena Mathew and K Poulose Jacob. Performance evaluation of popular hash functions. *World Academy of Science, Engineering and Technology*, 61:449–452, 2010.
78. Arjen K Lenstra. Key length. contribution to the handbook of information security. 2004.
79. K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017, RFC Editor, November 2016.
80. Thomas Pornin. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). RFC 6979, August 2013.
81. Google internet authority g2 – google. `https://pki.google.com/`. (Accessed on 09/05/2018).
82. Browser statistics. `https://www.w3schools.com/browsers/`. (Accessed on 09/08/2018).
83. Dan Boneh and Hovav Shacham. Fast variants of rsa. *CryptoBytes*, 5(1):1–9, 2002.
84. James A Muir. Seifert's rsa fault attack: Simplified analysis and generalizations. In *International Conference on Information and Communications Security*, pages 420–434. Springer, 2006.
85. Jakob Jonsson and Burt Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447, February 2003.
86. Levent Ertaul, Anup Mudan, and Nausheen Sarfaraz. Performance comparison of aes-ccm and aes-gcm authenticated encryption modes. In *Proceedings of the International Conference on Security and Management (SAM)*, page 331. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2016.
87. A. Yadin. *Computer Systems Architecture*, page 119. Chapman & Hall/CRC Textbooks in Computing. Taylor & Francis, 2016.
88. M. Badra and I. Hajjeh. ECDHE_PSK Cipher Suites for Transport Layer Security (TLS). RFC 5489, RFC Editor, March 2009.
89. U. Blumenthal and P. Goel. Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport Layer Security (TLS). RFC 4785, RFC Editor, January 2007.
90. F. Cusack and M. Forssen. Generic Message Exchange Authentication for the Secure Shell Protocol (SSH). RFC 4256, RFC Editor, January 2006.

91. Re: [TLS] TLS Extensions: Omitting Handshake Messages. `https://www.ietf.org/mail-archive/web/tls/current/msg24932.html`, 2017. (Accessed on 01/04/2018).
92. Leo Laporte Steve Gibson. Securitynow podcast episodes.
93. Kerry Maletsky. Rsa vs ecc comparison for embedded systems. *White Paper, Atmel*, page 5, 2015.

A Appendix

Operation / Security Level	**RSA Sign**	**RSA Verify**	**ECDSA Sign**	**ECDSA Verify**
low	20559190 (117075)	398584 (198)	14497591 (49045)	26839273 (50816)
normal	75738802 (317047)	1117868 (748)	29512991 (95776)	56260702 (162365)
high	317087210 (716961)	3295296 (254)	49150396 (82047)	94077150 (130275)
very high	1700652764 (2283718)	13436728 (702)	59732056 (441815)	114744021 (843861)

Table A.1: RSA and ECDSA signature creation and verification costs in number of CPU cycles. Numbers in parenthesis is the standard deviation

Operation / Security Level	**RSA Total**	**ECDSA Total**
low	20957774	41336864
normal	76856670	85773693
high	320382506	143227546
very high	1714089492	174476077

Table A.2: RSA and ECDSA costs of signature creation + signature verification in number of CPU cycles

Opeation / Security Level	**RSA Sign**	**RSA Verify**	**RSA Total**
low	-	-	-
normal	55179612	719284	55898896
high	241348408	2177428	243525836
very high	1383565554	10141432	1393706986

Table A.3: Absolute increase of RSA operation costs from previous security level in number of CPU cycles

Opeation / Security Level	**ECDSA Sign**	**ECDSA Verify**	**ECDSA Total**
low	-	-	-
normal	15015400	29421429	44436829
high	19637405	37816448	57453853
very high	10581660	20666871	31248531

Table A.4: Absolute increase of ECDSA operations cost from previous security level in number of CPU cycles

Security Level / Key Exchange	**low**	**normal**	**high**	**very high**
PSK	0	0	0	0
RSA	654077	2622235	5285561	16398134
RSA-PSK	654077	2622235	5285561	16398134
ECDH-RSA	1117868	1117868	1117868	1117868
ECDH-ECDSA	56260702	56260702	56260702	56260702
ECDHE-PSK	0	0	0	0
ECDHE-RSA	1516452	2235736	4413164	14554596
ECDHE-ECDSA	83099975	112521404	150337852	171004723
DHE-PSK	0	0	0	0
DHE-RSA	1516452	2235736	4413164	14554596

Table A.5: Client authentication costs for all ciphersuites and security levels in number of CPU cycles

Security Level / Key Exchange	**low**	**normal**	**high**	**very high**
PSK	0	0	0	0
RSA	20362831	75129504	314975365	1691976601
RSA-PSK	20362831	75129504	314975365	1691976601
ECDH-RSA	0	0	0	0
ECDH-ECDSA	0	0	0	0
ECDHE-PSK	0	0	0	0
ECDHE-RSA	20559190	75738802	317087210	1700652764
ECDHE-ECDSA	14497591	29512991	49150396	59732056
DHE-PSK	0	0	0	0
DHE-RSA	20559190	75738802	317087210	1700652764

Table A.6: Server authentication costs for all ciphersuites and security levels in number of CPU cycles

Opeation / Security Level	**ECDH Gen Keypair**	**ECDH Gen Secret**	**DH Gen Keypair**	**DH Gen Secret**
low	12942518 (33356)	12462677 (55222)	10455300 (31005)	10279378 (27044)
normal	27483912 (94940)	26958612 (137745)	67136033 (108793)	66712994 (107669)
high	45900358 (65731)	44331330 (100040)	474938146 (490496)	473634908 (493588)
very high	54449740 (487567)	53531554 (776984)	3592631108 (2792006)	3586324217 (2791154)

Table A.7: ECDH and DH costs for all security levels in number of CPU cycles. Numbers in parenthesis are the standard deviation.

Operation / Security Level	**ECDH Total**	**DH Total**
low	25405195	20734678
normal	54442524	133849027
high	90231688	948573054
very high	90231688	7178955325

Table A.8: ECDH and DH costs of the sum of keypair and shared secret generation in number of CPU cycles

Operation / Security Level	**ECDH Gen Keypair**	**ECDH Gen Shared**	**ECDH Total**
low	-	-	-
normal	14541394	14495935	29037329
high	18416446	17372718	35789164
very high	8549382	9200224	17749606

Table A.9: Absolute increase of ECDH operation costs from previous security level in number of CPU cycles

Security Level \ Operation	**DH Gen Keypair**	**DH Gen Shared**	**DH Total**
low	-	-	-
normal	56680733	56433616	113114349
high	407802113	406921914	814724027
very high	3117692962	3112689309	6230382271

Table A.10: Absolute increase of DH operation costs from previous security level in number of CPU cycles

Key Exchange \ Security Level	**low**	**normal**	**high**	**very high**
PSK	0	0	0	0
RSA-PSK	0	0	0	0
RSA	0	0	0	0
ECDH-RSA	25405195	54442524	90231688	107981294
ECDH-ECDSA	25405195	54442524	90231688	107981294
ECDHE-PSK	25405195	54442524	90231688	107981294
ECDHE-RSA	25405195	54442524	90231688	107981294
ECDHE-ECDSA	25405195	54442524	90231688	107981294
DHE-PSK	20734678	133849027	948573054	7178955325
DHE-RSA	25405195	133849027	948573054	7178955325

Table A.11: PFS and *ECDH* key exchange cost for the client in number of CPU cycles

Key Exchange \ Security Level	**low**	**normal**	**high**	**very high**
PSK	0	0	0	0
RSA-PSK	0	0	0	0
RSA	0	0	0	0
ECDH-RSA	12462677	26958612	44331330	53531554
ECDH-ECDSA	12462677	26958612	44331330	53531554
ECDHE-PSK	25405195	54442524	90231688	107981294
ECDHE-RSA	25405195	54442524	90231688	107981294
ECDHE-ECDSA	25405195	54442524	90231688	107981294
DHE-PSK	20734678	133849027	948573054	7178955325
DHE-RSA	20734678	133849027	948573054	7178955325

Table A.12: PFS and *ECDH* key exchange cost for the server in number of CPU cycles

Security Level Key Exchange	**low**	**normal**	**high**	**very high**
PSK	1354543 (51058)	1353865 (51289)	1355005 (51004)	1354829 (51043)
RSA-PSK	3536763 (71446)	4543238 (76038)	7293473 (84814)	18578392 (103341)
RSA	3556430 (59010)	4558935 (68078)	7309587 (71844)	18608492 (82623)
ECDH-RSA	28433250 (97731)	57412269 (122298)	93325335 (102107)	111532116 (623350)
ECDH-ECDSA	83731926 (78435)	112585306 (128327)	148415289 (126285)	166815336 (554471)
ECDHE-PSK	26819362 (68991)	55805138 (138561)	91713096 (142807)	109203285 (813689)
ECDHE-RSA	28862986 (105477)	58608367 (214574)	96730901 (93708)	124773055 (696488)
ECDHE-ECDSA	110524836 (93990)	168954007 (267635)	242501399 (149155)	280626510 (1071418)
DHE-PSK	22189619 (55603)	135442518 (309751)	950212732 (787892)	7181764421 (5195398)
DHE-RSA	24250886 (94421)	138197524 (198704)	955840114 (861069)	7196088947 (6405987)

Table A.13: Handshake costs for the client in number of CPU cycles

Security Level Key Exchange	**low**	**normal**	**high**	**very high**
PSK	1380956 (50888)	1382849 (50737)	1381497 (50891)	1382002 (50903)
RSA-PSK	22515609 (111975)	77368868 (275574)	317389619 (662953)	1694178055 (2427763)
RSA	22456180 (121140)	77260738 (239418)	317196961 (698637)	1694747500 (1705523)
ECDH-RSA	14534087 (49806)	29154666 (127699)	46480720 (107771)	55636527 (576001)
ECDH-ECDSA	14521154 (58001)	29094320 (111114)	46390603 (132250)	55927921 (688074)
ECDHE-PSK	26869449 (110405)	56020895 (147825)	91773474 (94656)	109500050 (905013)
ECDHE-RSA	48164890 (126885)	132433386 (379327)	409652623 (951326)	1810563620 (3011341)
ECDHE-ECDSA	41984222 (82371)	86096716 (192885)	141506181 (147013)	169996896 (1051307)
DHE-PSK	22308029 (76182)	135535409 (232955)	950691653 (1289706)	7181243788 (5402427)
DHE-RSA	43602094 (154553)	212251542 (364611)	1268235354 (857921)	8883866054 (5993186)

Table A.14: Handshake costs for the server in number of CPU cycles

Operation Security Level	**RSA Sign**	**RSA Verify**	**ECDSA Sign**	**ECSDA Verify**
low	11355 (138)	293 (5)	8487 (2010)	14219 (175)
normal	41672 (342)	747 (8)	16130 (441)	30750 (246)
high	176255 (997)	2096 (15)	26961 (221)	51457 (272)
very high	939928 (3238)	7704 (33)	34246 (334)	65877 (549)

Table A.15: RSA and ECDSA signature creation and verification costs in microseconds. Numbers in parenthesis is the standard deviation