Table of Contents

Transport Layer Security Protocol For Internet Of Things

Illya Gerasymchuk
illya@iluxonchik.me

Instituto Superior Tcnico
Supervisors: Ricardo Chaves, Aleksandar Ilic

Abstract. Transport Layer Security (TLS) is, by far, the most used communication security protocol, it is however, not suitable in the context of Internet Of Things (IoT). The resource-limited nature of a big part of IoT devices does not allow for the use of computationally complex and memory demanding operations present in a standard TLS implementation. Most of previous work focused entirely on Datagram TLS (DTLS) and can not be easily integrated with existing deployments. This work focuses on how TLS and the extension mechanism can be used to define a framework to adapt the protocol to specific needs. This is the approach that will be followed in the second part of the work. Having an adaptable and easy to use solution is crucial for its adaptation in IoT, where security might have been completely foregone otherwise.

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

1 Intoduction

The Internet of Things (IoT) is a network of devices, from simple sensors to smartphones and wearables which are connected together. In fact, it can be any other object that has an assigned IP address and is provided with the ability to transfer data over a network. Even a salt shaker[**?**] 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 privacy and 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[**?**] 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[**?**] could be even worse if it is ever put to malicious use. Similar attacks will inadvertently come in the future.

TLS is one of the most used security protocols in the world, allowing two peers to communicate securely. It is designed to run on top of a reliable, connection-oriented protocol, such as TCP. Datagram TLS (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, the use of TCP with IoT is usually not the best option. For this reason, DTLS, which runs on top of UDP, is used more frequently in such devices. The work that will be done in the context of this dissertation, can however, be applied to either one of them, so even though mostly TLS will be mentioned, almost everything can also be applied to DTLS. This is a consequence of DTLS being just an adaption of TLS over unreliable transport protocols, with no changes done to the core protocol.

The problem in using (D)TLS in IoT is that it is not lightweight, since it has not been designed for such environments. An IoT device may only have 256 KB of RAM and needs to conserve the battery, while sending and receiving a large amount of small information constantly. For example, consider the case of a temperature sensor that sends temperature measures every 30 seconds to a server. In this case it just needs to send a few bytes of data and do it with minimal overhead, to conserve RAM and battery. If that sensor is going to use (D)TLS 1.2, it will need two extra roundtrips before it can send any data. This can result in an overhead of several hundreds of milliseconds. Besides that, it will need to perform heavy mathematical operations involved in cryptography, using even more energy and taking even more time. Given this, there is a clear need for a more lightweight (D)TLS for the IoT.

The goal of this work is to develop a lightweight version of (D)TLS that is fully backwards compatible and does not require any third-party entities, in order to simplify its deployment process. The solution will be developed for (D)TLS version 1.2, while also bearing in mind the new 1.3 version. The idea is to make it customizable, depending on the security requirements and the context of its usage.

In the process of the work on this dissertation, we have already made several contributions to the TLS 1.3 specification, being recognized as contributors[**?**].

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 (version 1.3 is 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 provides an architecture of the solution that will be developed in the second part of the dissertation, and describes how the results will be evaluated and presents a general work plan. Finally, the conclusion of the work is done in Section 6.

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 **??**. Section **??** 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 **??**. When compared to other public key cryptography approaches, Elliptic Curve Cryptography (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 **??**.

2.1 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, CPUs are slower at performing the underlying mathematical operations involved in AC, namely exponentiation requires $O(loge)$ multiplications for an exponent e. The 2016 NIST report [**?**] 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 [**?**] (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[**?**] which have approximately 10KB of RAM and 100KB of persistent memory. We measured and compared the resulting size of the *mbedTLS 2.6.0* library[**?**] binary when it was compiled with and without the Rivest-Shamir-Adleman (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[**?**].

2.3 Authenticated Encryption With Associated Data (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 Handshke 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 **??** and described in more detail in Section **??**.

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[**?**] 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[**?**] served as the basis for TLS 1.0[**?**].

Section **??** 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 **??**. The way each record is processed when sending and receiving data is covered in Section **??**. The symmetric keys involved in cryptographic operations that provide confidentiality and security are described in Section **??**. Section **??** 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 **??**. The TLS Extension mechanism will be covered in Section **??**. 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 **??**. This section ends with an outline of the main differences from DTLS to TLS in Section **??**.

3.1 TLS (Sub)Protocols

TLS is composed of several protocols, which are illustrated in Figure **??** 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 **??**.
- **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) 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

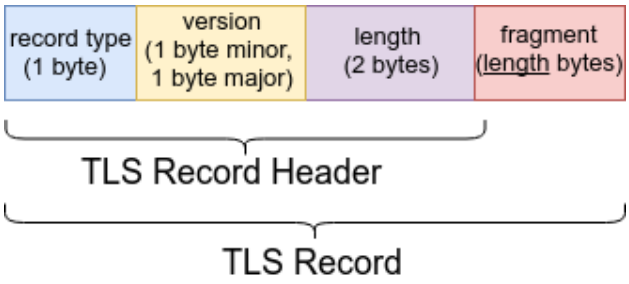


Fig. 1: TLS Record header

Fig. 2: TLS (Sub)protocols and Layers

example, let us say that a Hypertext Markup Language (HTML) page is being downloaded over the Hypertext Transfer Protocol Secure (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`[**?**] and **session tickets**, defined in `RFC 5077`[**?**].

– **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 **??** 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).

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

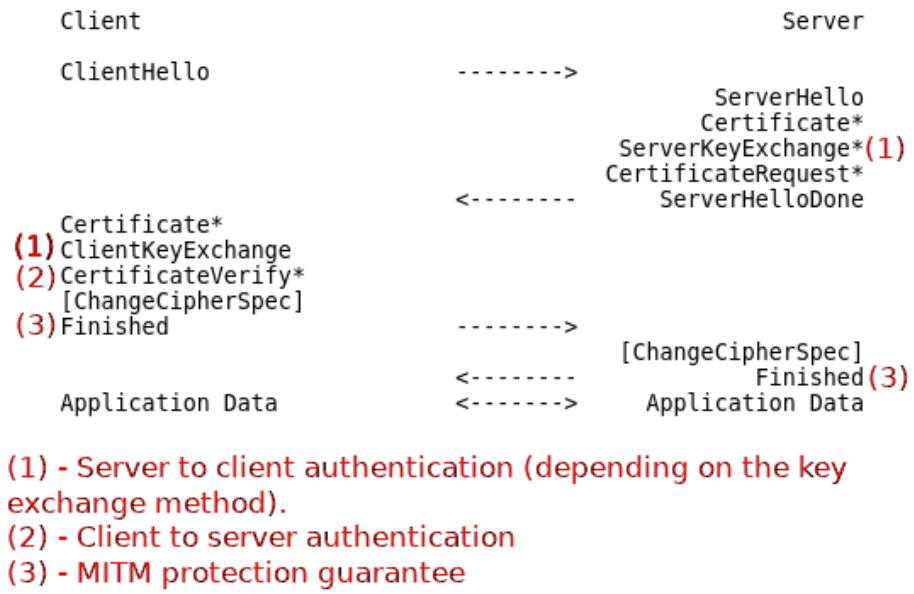


Fig. 3: TLS 1.2 message flow for a full handshake

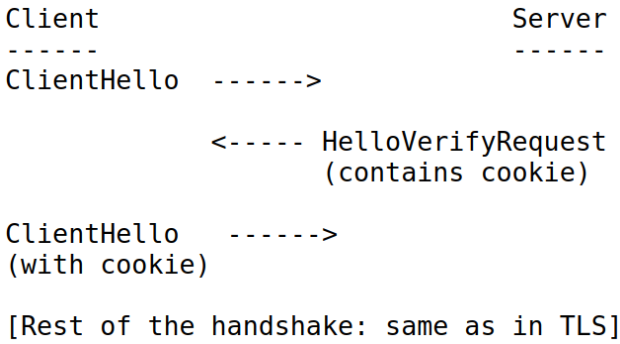


Fig. 4: DTLS handshake with HelloVerifyRequest containing the cookie

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"[**?**]. 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 **??**. 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 **??**. 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
- client write MAC key (TLS 1.2 only) - used by the client to authenticate the data to be sent

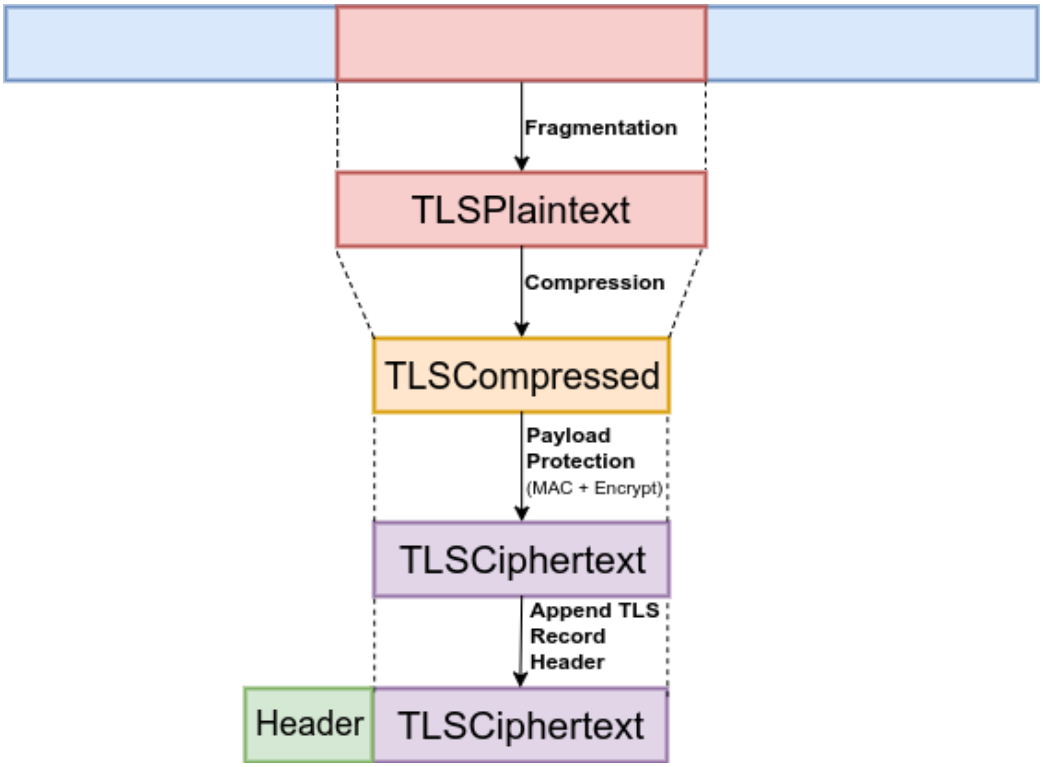


Fig. 5: TLS 1.2 Record Processing

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

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`[**?**]), while others in separate Request For Comment (RFC)s. For example, the ECC based key exchange, specified in `RFC4492` [**?**]).

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. Perfect Forward Secrecy (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 **??** 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[**?**] 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 (Elliptic Curve Digital Signature Algorithm (ECDSA)) algorithms with TLS is described in `RFC4492`[**?**]. 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[?] 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 ClientHellos that they do not understand (section 7.4.1.2 of RFC 2246[?]). 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[?] will not be described in detail. The focus was on TLS 1.2 instead, because TLS 1.3 is still in draft mode and 1.2 is the latest and the recommended to use version. Despite the protocol name not suggesting it, TLS 1.3 is very different from TLS 1.2. It should have probably been called TLS 2.0 instead.

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 ClientHellos 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 [**?**] 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[**?**]. There is a draft of DTLS 1.3 [**?**] 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 **??**. 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*[**?**] 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 [**?**], 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 **??**.

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

	Ultra-Lightweight	**IoT**
Block Size	64 bits	128 bits
Security Level	80 bits	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*[**?**] 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"[?] 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[?], 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`[?]. 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`[?].

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` [?]. To achieve this, a new certificate structure is defined, alongside a new `certificate_type`. The new certificate structure is just the `RFC7250` [?] structure, with an additional MAC.

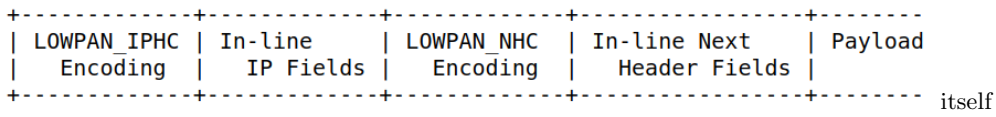
The hash function used for key derivation is SHA256. The authors evaluated the performance of their solution with and without SHA2 hardware acceleration and concluded that while it had significant impact on key derivation, it had little impact on the total handshake time (`711.11 ms` instead of `775.05 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[?] 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`[?]. The compression relies on the shared context between the communicating peers.

The work proposed in [?] 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[?] 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.

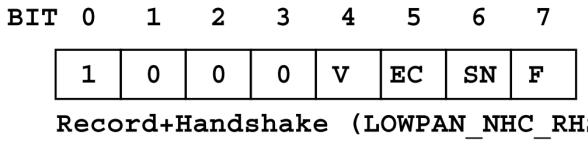
Their work defines how the DTLS Record header, the DTLS Handshake header, the `ClientHello` and the `ServerHello` messages can be compressed, but notes that the same compression techniques can be used to compress the remaining handshake messages. They explore two cases for the header compression: compressing 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.



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 **??** 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`[**?**]) as in `RFC7400` and define the format of the `In-line Next Header Fields` (also defined in [**?**]), 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 **??**), 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[**?**], 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[**?**], which is a lightweight security solution for Constrained Application Protocol (CoAP) that uses the same DTLS header compression technique as in [**?**] with the goal of implementing it as a security support for CoAP. CoAP[**?**] 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[**?**].

The authors evaluated their system in a simulated environment in *Contiki OS*[**?**], 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 [**?**]), 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.* [**?**] 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[**?**], 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 [**?**]), 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[**?**], 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 [**?**] 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 [**?**], 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`[**?**] 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[**?**] 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`[**?**]. 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[?]. 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[?] 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[?], 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[?] 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[?]. For certificate revocation, neither the Online Certificate Status Protocol (OCSP)[?], nor the Certificate Revocation List (CRL)[?] 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[?] 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[?] 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[?] 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 [?] 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[?], 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[?].

The use of compression is not recommended for two reasons. First, RFC7525[?] recommends disabling (D)TLS level compression, due to attacks such as CRIME[?]. 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[?] 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[?].

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[?], nor the truncated MAC[?] extensions are applicable to this specification and must not be used.

The Server Name Indication (SNI) extension[?] 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[?] 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[?] defines an extension that binds the master secret to the log of the full handshake, thus preventing MITM attacks, such as the triple handshake[?]. 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[?], 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[?].

RFC 7925[?] 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[?] 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[?] currently in Internet-Draft state.

5 Solution

5.1 Things That Might Be Useful To Include Somewhere

Not all IoT devices are limited to the point of not being able to use public key cryptography altogether. For some of them, the use of RPKs, which is considered the first entry point into the area of public key cryptography, is acceptable, while others are powerful enough to take advantage of certificates and Public Key Infrastructure (PKI), at least up to a point.

5.2 Evaluation

In order to evaluate the quality of the work, both, the original protocol implementation and the one provided as part of the solution, will be profiled and compared. The relevant profiling metrics are power consumption, RAM usage, storage usage, CPU cycles elapsed and time taken. The profiling will be done over various simulated scenarios, which emulate real-life usage, such as connecting to the server multiple times over a short time period and transferring small quantities of data, and connecting to the server and transferring a large amount of data, all at once.

Due to limited time and the fact that TLS 1.3 still lacks stable implementations, most likely, only the solution under TLS 1.2 will be implemented and evaluated.

5.3 Planning

During the month of February, up until mid-March 2018, the solution will be defined precisely. Due to the nature of the solution, it will have many different versions, depending on the target device and required security services. Most likely, there will be no time to implement and evaluate every possible scenario, so only a subset of them will be chosen.

From mid-March until mid-April 2018, a version that uses existing configuration options and protocol extensions to best support the IoT environment will be developed. In essence, this would involve incorporating a lightweight profile (like RFC 7925 does) into the solution. The system will be implemented in code, by modifying the *mbedTLS 2.6.0* library[?].

From mid-April until June 2018, the customized part will be implemented. This might involve custom cipher suites, key exchange methods and changes in the Handshake Protocol.

From June until July 2018, the work will be focused around PSK solutions. This might involve adapting the existing PSK configurations or creating new ones.

From July until August 2018, the work will be evaluated. The most important evaluation metrics will be chosen and the related profiling code set up. Testing scenarios will be designed and implemented.

From mid-July until mid-September 2018, the focus will be on writing the dissertation's text. Some minor improvements and additions might also be done during this period of time.

6 Objectives

IoT devices have limited resources. Those resources are processing speed, memory and power. Communication security is a desirable property in the context of interconnected devices. There are many protocols that can be used to provide communication security. (D)TLS is one of the most used protocols for this purpose.

Due to the constrained nature of the IoT devices, typical (D)TLS configurations cannot be used in many cases.

(D)TLS is a complex protocol with numerous possible configurations. Each configuration implies a certain security level and resource usage. In fact, it's almost always a tradeoff between these two. When configured properly, (D)TLS can run on constrained devices. Such a configuration might imply foregoing some of the security serivices, or using a lower security level.

A (D)TLS configuration consists of a key exchange algorithm, an encryption algorithm, a hash function and the associated key sizes. There are numerous choices for each, which leads to numerous possible configurations. As an example, the *mbedTLS 2.7.0* library has a total of 161 possible configurations, without taking into account the asymmetric cryptography key sizes. Existing work does not explore the costs of the various configurations. It also fails to establish a relationship between the security services, security level and their associated costs in the context of (D)TLS. Developers wishing to deploy the (D)TLS protocol in constrained environments do not have a tool that would help them to select a (D)TLS configuration appropriate to the environment's needs and limitations.

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

Another topic that existing work fails to explore with enough detail is TLS optimization. Most of the work has been centered around DTLS and not all of it can be applied to TLS, since it Herein we want to further explore TLS optimization. There is clearly a need for that, specially with CoAP over TCP and TLS standard 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.

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 level/resource usage tradeoffs, according to the environment's needs and limitations. In order to achieve

this goal, the costs of each individual security service will be evaluated. With this information, the programmer 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, he can decide on an alternative configuration, possibly with a loss of some security services and a lower security level, or forgo using (D)TLS altogether.

7 Methodology

Things to cover:

- local machine specs
- why CPU cycles is a good measure (how CPU cycles relate to time taken; say that cryptography is CPU-bound)
- why collecting on a local machine makes sense (should be similar on others)
- which tools I develped any why I did (too many configurations to evaluate manually)

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 *cihersuite*. 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 ((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 step. 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 of them.

(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 [**?**].

mbedTLS implements a subset of those ciphersuites. As of version 2.7.0, *mbedTLS* has a total of 161 ciphersuites [**?**]. 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. The obtained metrics reflect the algorithm's implementations used within the library.

7.1 Evaluated Metrics

Explain what was evaluated. Explain why those metrics make sense (e.g. explain why number of CPU Cycles) is a good measure.

7.2 Limitations

My cache:

L1d cache: 32K L1i cache: 32K L2 cache: 256K L3 cache: 6144K

* what is valgrind * how does valgrind work * what is callgrind * how does callgrind work * explain how this limits the accuracy of results * explain why I went with those instead of actual measures (speed, time limit, etc) * desribe CA (RSA 2048 bit key) – put here or other section? – * with which options I profiled the resutls with * why I chose those options

In order to estimate the number of executed instructions, 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 recomplied machine code will then run on the host CPU (the JIT compilation step). This process is illustrated in Figure **??**.

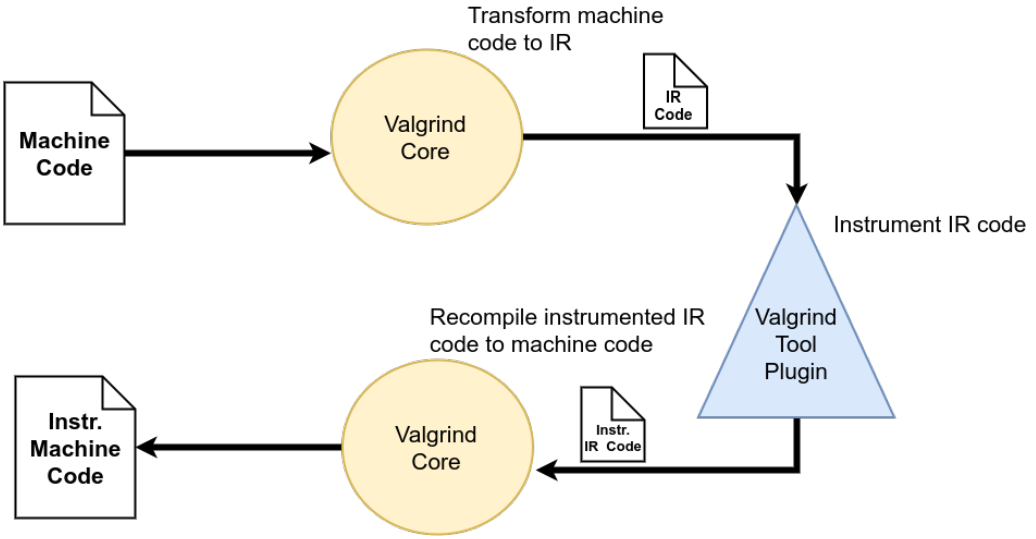


Fig. 8: How Valgrind works

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* in conjunction with *KCachegrind* are widely used 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 instructions 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 [**?**]. It uses the following formula: $CEst = Ir + 10 * Bm + 10 * L1m + 20 * Ge + 100 * L2m + 100 * LLm$, where

- $CEst$ - estimated CPU cycles
- Ir - instruction fetches
- Bm - mispredicted 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 $CEst = Ir + 10 * Bm + 10 * L1m + 100 * L2m$.

7.3 Developed Tools

Describe the tools that have been developed.

7.4 mbedTLS Set Up

* Tell that I activated things in mbedTLS that come deactivated by default, since they have been deprecated. Some of those things are deactivated, because there are known security problems with them, but they might still make sense for IoT, beacuse it might be just "good enough" encryption.

For all of the evaluated scenarios, a certificate chain with two certificates was used: the CA's certificate and the server's certificate. This is the smallest chain you can have. The CA's keys and signing algorithm was the same for all of the certificates. More specifically, all of the server certificates were signed with a 2048-bit RSA with SHA-256 secure signing scheme, defined in *RFC 8017* [**?**]. The chosen signature algorithm and key size is based on the usual CA practices. For example, Google's root certificate with the common name *Google Internet Authority G3* uses this set up.

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.

8 Results and Data Analysis

* tell what ws evaluated * tell why we concentrated on the handshake protocol * section for handshake protocol evaluation For each cipher, show evaluation * section for encryption evaluation * section that compares handshake with encryption * put the join graph * analyze when the handshake cost equals to encrypiton cost * mention that in the extreme case, the handshake can only be done once and the conncection kept alive indefintely.

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 guarantees data integrity. 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. TODO: ref The is an approximately linear relation between the ammount of data encrypted and the the cost. TODO: ref.

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, we will however, discuss practical limitations later in this text TODO: write TODO: what about stream ciphers? they can have an infinite key size. make this more clear.
- 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 TODO: ref.

Another cryptographic operation that is done in the Handshake phase is 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's 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 8.1 describes the various security levels used for evaluation. Section 8.2 presents an overview of the costs of authentication and PFS. Section 8.3 goes into more detail of the costs of those services, with an evaluation of various security levels. In section 8.4 we analyse the cost of the Handshake phase for the various *ciphersuites* and security levels. Section 8.5 discusses the cost of confidentiality and integrity. Section 8.6 the cost of the Record protocol is analyzed. In that same section we also compare the costs of the Handshake protocol to the cost of the Record protocol.

8.1 Security Levels

We defined various security levels for evaluation. Those levels are presented in Table **??**. This table is based on the approximate comparable key sizes for symmetric and asymmetric key algorithms based on the best known algorithms for attacking them (table **??**, which can be found in RFC 4492[**?**].

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.

TODO: add forumula and explanation on how I got the ECC sizes for sizes of RSA below 1024 (look at notes in md)

TABLE: (go from 500KB to 8K, as in notes from meetings)

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

[1] The closest value available in mbedTLS 2.7.0 (rounded up) is 192
[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

	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]

We based the **normal** security level on the current most used TLS configuration on the internet [**?**]. We could not find any typical values used in the sphere of IoT. A detailed description of the **normal** security level follows.

The normal security level TODO: add section for other security levels

A certificate chain with two certificates is used: the CA's certificate and the server's certificate. The CA's keys and signing algorithm was the same for all of the certificates. More specifically, all of the server certificates were signed with a 2048 bit RSA with SHA-256 secure signing scheme, defined in *RFC 8017* [**?**]. The chosen signature algorithm and key size is based on the usual CA practices. For example, Google's root certificate with the common name *Google Internet Authority G3* uses this set up. Only server-side authentication is used.

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 certificates will contain an 2048 bit RSA key;
- 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[**?**].

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. Adding more certificates to the chain would not provide additional information, since that cost can be computed by adding the costs of making/verifying additional signatures.

8.2 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 used ciphersuite. 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 in making security/performance tradeoffs. A *ciphersuite* can be thought of as TLS *configuration*, thus, these terms be used interchangeably.

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 **medium** 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 encrypiton algorithms and 4 unqiue 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 cerntain scenarios, if their cost considerably lower. For this reason, we have enabled and evaluated all of the possible configurations.

Since the evaluted security level 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.

Figures **??** and **??** 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 instructions 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. We decided to omit the name of those combinations, since this information is not crutial for this analysis and it clutters up the graph. What is important to see is the fact taht 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 the keying material (either using PFS or not).

From the anaylysis of Figures **??** and **??**, it becomes obvious that some of the ciphersuites have much lower Hanshake costs than the others. For the client, the *PSK* based ciphesuites 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 for each key exchange method. This information is presented in table **??** for the client and in table **??** for the server. Figures **??** and **??** depict this information graphically.

Each row and column intersection in the tables **??** and **??** forms a ciphersuite. The rows separate the various authentication algorithms that can be used. The columns sperate the key exhchange methods that offer PFS, from the ones that don't. Each entry of the table presents the average number of CPU cycles with its standard deviation in parenthesis.

Client By analyzing table **??** figure **??** we can approximate the costs of the security serives 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 here are taken directy from table **??** and will be in millions CPU instructions, rounded to 3 decimal places.

If PFS is used, the cost of the Handshake varies from 52.883 to 133.524 million CPU instrutions for PSK based authentcation, and from 55.538 to 160.445 million CPU instructions for asymmetric authentcation. If the seucirity service of PFS is foregone, the cost of the Handshake varies from 1.199 to 42.041 for PSK based authentcation, and from 54.333 to 106.8 for asymmetric authentcation.

If we analyze the values in the *ECDHE* column in table **??**, thus fixing the algorithm used to offer PFS we can compare the costs of authentcation for various algorithms on the client-side. For example, we can see that *RSA* authentication costs 2.445 ($55.538 - 52.883$) more than PSK authentication and *ECDSA* authentcation costs 104.97 ($160.445 - 55.538$)

million CPU instructions more than *RSA* authentcation. The total cost of using PSK for authentcation can be estimated by looking at the value in the *PSK* row and *X* column, thus fixing authentcation to *PSK* and not using any other algorithm for key agreement: 1.119. In the same manner, the cost of using *RSA* for authentcation can be approximated by taking the value located in row *RSA* and column *X*, thus fixing authentcation to RSA and not using any other algorithm for key agreement: 4.211. Even though we cannot estimate the cost of using *ECDSA* for authentcation directly from the table, we have already seen that *ECDSA* authentcation is 104.97 more costly than *RSA* authentcation. Thus, the cost of using *ECDSA* for authentcation is of 109.181(4.211 + 104.97) million CPU instructions. When compared, authentcation with RSA is 276.318% more expensive than with PSK and authentcation with ECDSA is 2492.757% 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 authentcation algorithm. Using *DHE* is 80.641 (133.524 − 52.883) million CPU instructions more (+152.489%) 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 authentcation algorithm to PSK and subtracting the cost of the Handshake when no PFS is used, from the cost of the Hanshake when *ECDHE* is used to provide PFS. Thus, the cost of using *ECDHE* to provide PFS is of 51.684 million CPU cycles (52.883 − 1.199). Since we already know that *DHE* costs 80.641 million CPU instructions more than *ECDHE*, we can compute the cost of using the *DHE* to provide PFS: 132.325(51.684 + 80.641) million CPU instructions.

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

If PFS is used, the cost of the Handshake varies from 52.938 to 133.583 million CPU instrutions for PSK based authentcation, and from 81.502 to 208.721 million CPU instructions for asymmetric authentcation. If the seucirity service of PFS is foregone, the cost of the Handshake varies from 1.217 to 75.593 for PSK based authentcation, and from 27.519 to 274.607 for asymmetric authentcation.

If we analyze the values in the *ECDHE* column in table **??**, thus fixing the algorithm used to offer PFS we can compare the costs of authentcation for various algorithms on the server-side. For example, we can see that *RSA* authentication costs 74.819 (127.757 − 52.938) more than PSK authentication and *ECDSA* authentcation costs 46.255 (81.502 − 127.757) million CPU instructions less than *RSA* authentcation. The total cost of using PSK for authentcation can be estimated by looking at the value in the *PSK* row and *X* column, thus fixing authentcation to *PSK* and not using any other algorithm for key agreement: 1.217. In the same manner, the cost of using *RSA* for authentcation can be approximated by taking the value located in row *RSA* and column *X*, thus fixing authentcation to RSA and not using any other algorithm for key agreement: 75.840. Even though we cannot estimate the cost of using *ECDSA* for authentcation directly from the table, we have already seen that *ECDSA* authentcation is 46.255 less costly than *RSA* authentcation. Thus, the cost of using *ECDSA* for authentcation is of (29.585)75.840 − 46.255 million CPU instructions. When compared, authentcation with ECDSA is 2330.978% more

expensive than with PSK and authentcation with RSA is 156.346% 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 authentcation algorithm. Using *DHE* is 80.645 (133.583 − 52.938) million CPU instructions more expensive (+152.339%) 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 authentcation algorithm to PSK and subtracting the cost of the Handshake when no PFS is used, from the cost of the Hanshake when *ECDHE* is used to provide PFS. Thus, the cost of using *ECDHE* to provide PFS is of 51.721 million CPU cycles (52.938 − 1.217). Since we already know that *DHE* costs 80.645 million CPU instructions more than *ECDHE*, we can compute the cost of using the *DHE* to provide PFS: 132.372(51.727 + 80.645) million CPU instructions.

Conclusion By looking at figures **??** and **??** it is clear that some ciphersuites are more costly than others. This dissimilarity can explaiened 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 authentcation 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 TODO: ref. Symmetric authentcation is, by far, the least costly one for both peers. For the client, RSA is the second least costly authentcation 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 4.*X*, when we analyse 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 analyse both of the security services in more detail.

The previous section gave an overview of the security services of authentcation and PFS. We deduced their approximated cost by analyzing the costs of the Handshake with various ciphersuites. In the follwoing two sections we will do a more detailed analysis of those serivces and the underlying algorithms that support them.

8.3 Authentication Cost Analysis

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.

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

PSK Authentication Cost Analysis In the prevoius seciton we approximated the cost of PSK authentcation to be 1.119 million CPU instructions for the client and 1.217 million CPU instructions for the server. In reality, this cost is very close to zero, since both of the parties already have the authentcation key. The approximated value is actually the cost of performing the smallest and least costly Handshake in *mbedTLS*. This value can be seen as the "TLS Handshake" ovehead. Approximately 69% of those CPU instructions are spent on executing to PRF function to derive the shared keying material. The tasks of computing own and cheking the other party's *Finished* message, both cost approximately 10% of the total Handshake cost. The remaining CPU instrucitons are spent on reading/wrting to/from the network and computing the remaining TLS Handshake messages.

Table **??** shows the number of CPU instrucitons spent in the PRF and in computing the *Finished* message for all of the defined security levels. This number is the average of all of the 161 ciphersuites, with the standard deviation shown in parenthesis. An analysis of the table shows that this value is amost the same for all security levels. This can be explaiened by the fact that both of the operations are essentially hashing operations and even if the input size varies by a few hundred bytes, the total cost does not increase by a lot.

TODO: table

8.4 Asymmetric Algorithms Authenticaition Cost Analysis

If asymmetric cryptography is used for authentcation, there are two choices of algorithms: RSA and ECDSA. Each one of them has advantages and disadvantages, depending on the scenario. We will analyse 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. Figures **??** 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.

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 analyse each one of the algorithms and their costs in more detail.

RSA We have already seen the costs of RSA for the **normal** security level and now we will analyse the remaining ones. Table **??** shows the number of CPU instuctions used when signing a message with size 20, 32 or 48 bytes and verifying the resulting signature with

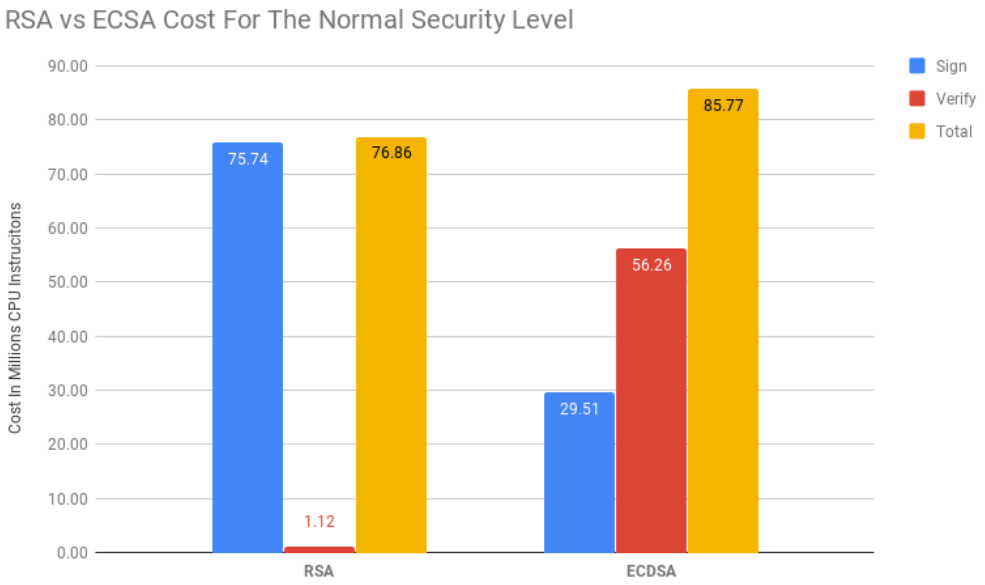


Fig. 9: RSA and ECDSA operations cost for normal security level

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 **??** 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 ceritifcates have signatures over a 32 byte value (output of *SHA-256*), so the values in Table **??** apply to those costs too.

Figure **??** depics 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 incrase for both, signature creation and verification. This is a result of modular exponentiation being RSA's core operaion. Moreover, for all security levels, the cost of private key operations is significatly 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 singature 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 instuctions 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. This information is shown Table **??** and graphically in Figure **??**. 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 **??**, wich presents this relative increase in terms of absolute values. The numbers are always going up, with the signature creation increase being sinficantly larger than the signature verification one. An analysis Figure **??** 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 $x1.5$ larger than the verifying one. This has a cumulative effect: the constatly larger value increses even more. This is shown graphically in Figure **??**, which for RSA, presents the ratio between the cost of private and public key operations. As the seucirity level increases, the value becomes larger. The fact that the signing operation dominates the total cost can also be seen graphically in Figure **??**. 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 incerase is only 1.4% smaller than the signing opeartion cost incerase.

	RSA Sign	**RSA Verify**	**ECDSA Sign**	**ECDSA Verify**
low	20559190 (117075)	398584 (198)	14497591 (49045)	26839273 (50816)
normal	75738802 (317047)	111786 (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 3: RSA and ECDSA signature cereation and verification costs

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

Table 4: RSA and ECDSA costs of signature creation + signature verification

	RSA Sign	**RSA Verify**	**Total**
low	-	-	-
normal	268.4%	180.5%	266.7%
high	318.7%	194.8%	316.9%
very high	436.3%	307.8%	435%

Table 5: Relative increase of RSA operation costs from previous security level

	RSA Sign	RSA Verify	RSA Total
low	-	-	-
normal	55179612	719284	55898896
high	241348408	2177428	243525836
very high	1383565554	10141432	1393706986

Table 6: Absolute increase of RSA operation costs from previous security level

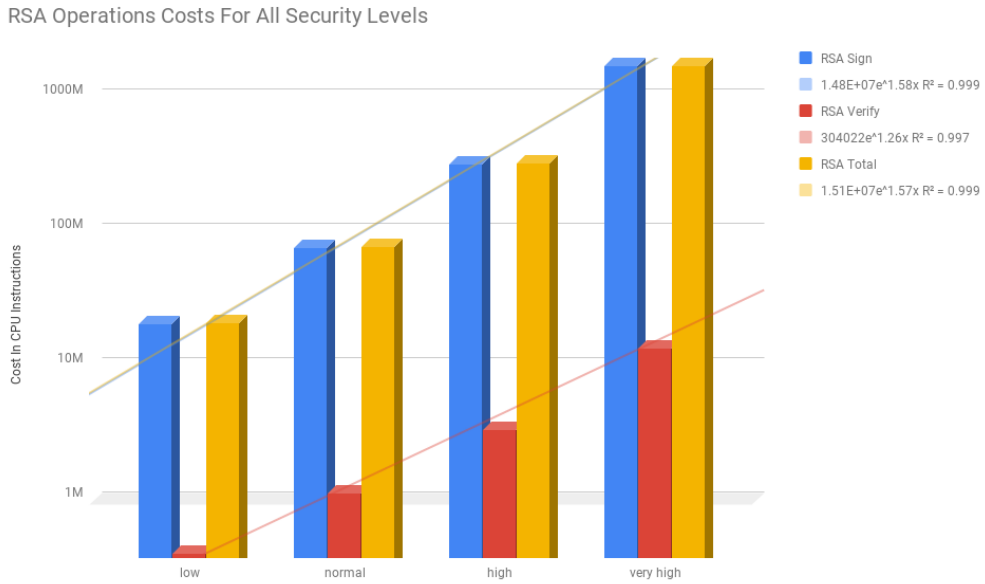


Fig. 10: RSA operations costs for all security levels (logarithmic scale)

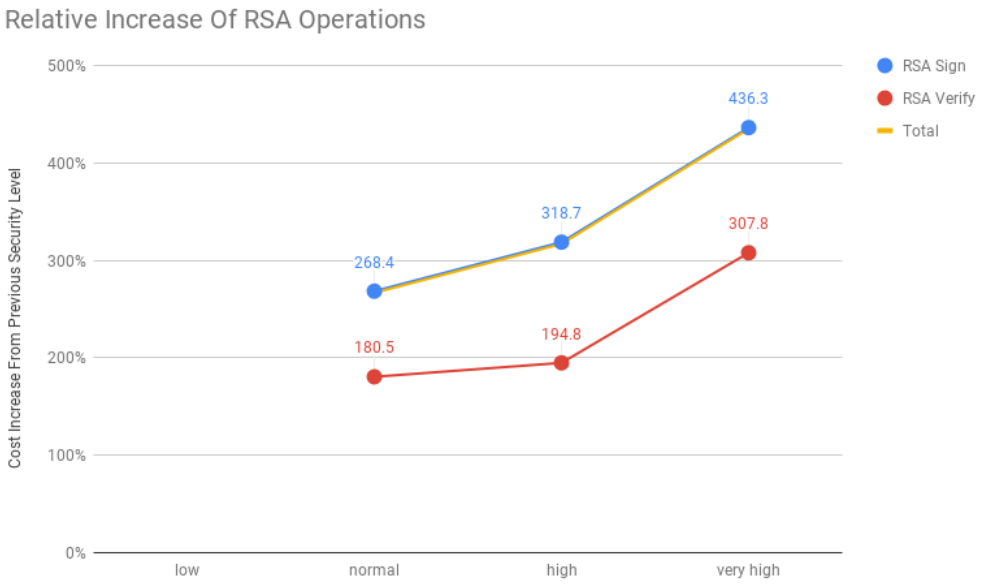


Fig. 11: Relative increase of RSA operation costs

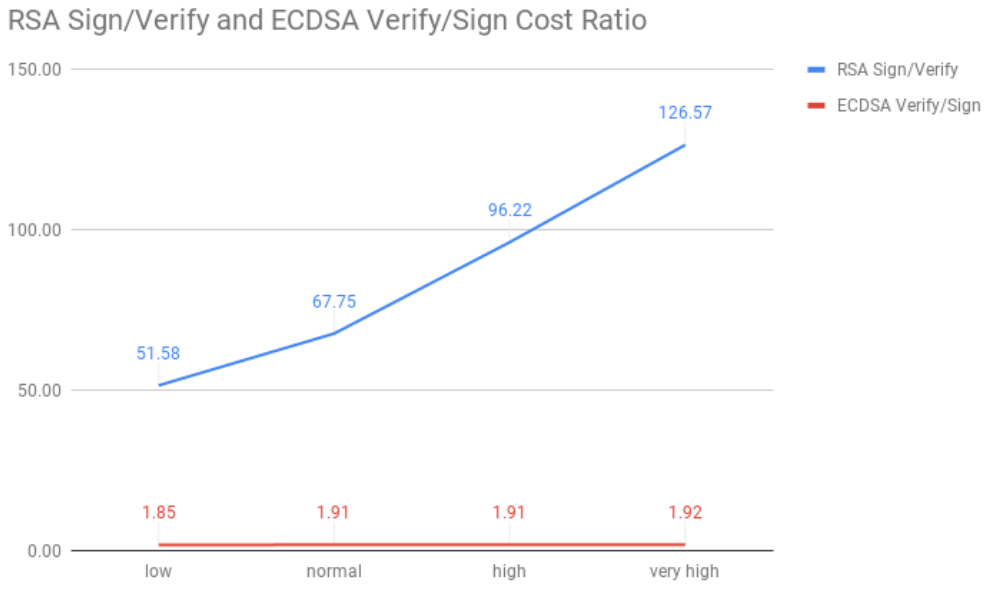


Fig. 12: Ratio between the most and the least costly operation for RSA and ECDSA

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 [**?**][**?**]. 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 After analyzing the costs of RSA, we will now perform a similar analysis for ECDSA. Figure **??** shows the costs of ECDSA opeartions graphically. The values are taken from Table **??**. Unlike in RSA, in ECDSA the private key operation is the least costly one. Figure **??** shows this ratio for both, RSA and ECDSA. Besides the ratio being a lot smaller for ECDSA, it vaires 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's shown in figure **??**, a logarithmic trendline is the one that best fits the cost incrase for both, signature creation and verification. Table **??** shows the relative cost increse of of ECDSA's operations. The percentages show the relative cost increase from the previous security level. Figure **??** presents this information graphically. There are two major differnces from RSA. First, the relative cost incerase 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 **??**, 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 as much as it happens in RSA. Second, as the security level increases, the cost increase from the prevoius 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 **??**. 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.

	ECDSA Sign	**ECDSA Verify**	**Total**
low	-	-	-
normal	103.6%	109.6%	107.5%
high	66.5%	67.2%	67%
very high	21.5%	22%	21.8%

Table 7: Relative increase of ECDSA operations cost from previous security level

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

	ECDSA Sign	ECDSA Verify	ECDSA Total
low	-	-	-
normal	15015400	29421429	44436829
high	19637405	37816448	57453853
very high	10581660	20666871	31248531

Table 8: Absolute increase of ECDSA operations cost from previous security level

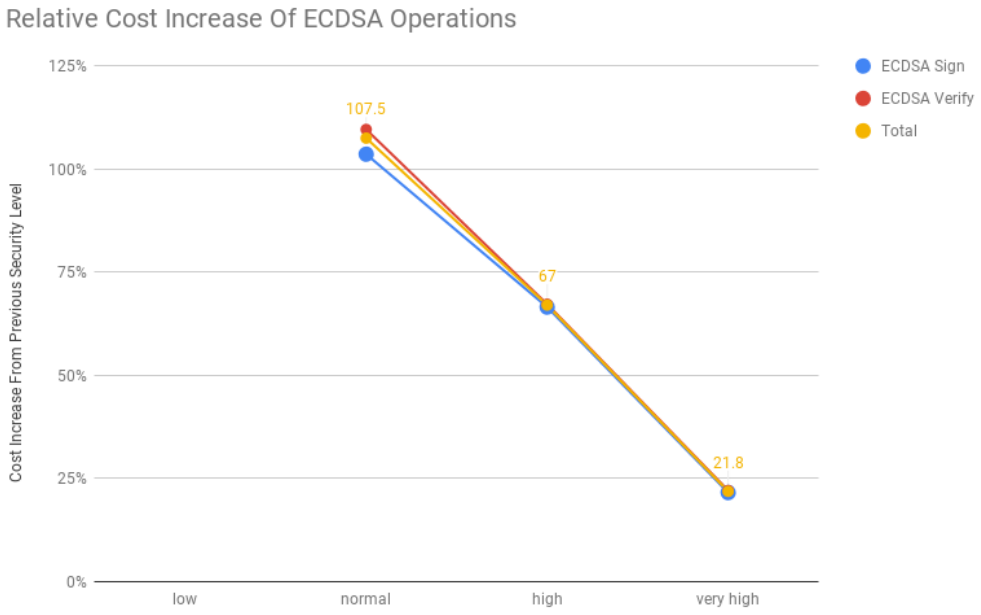


Fig. 13: Relative increase of ECDSA operations cost

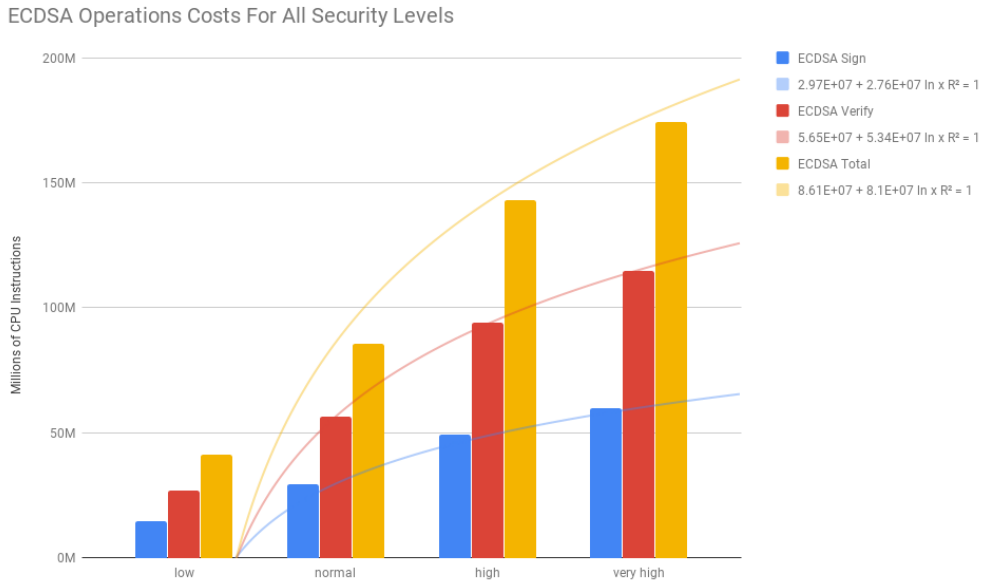


Fig. 14: ECDSA costs for all security levels

for ECDSA. This can be explained by the fact that smaller keys are used in ECDSA, as we have alerady seen in Table **??**, and is a big advantage of ECC.

RSA vs ECDSA After having analysed the costs of RSA and ECDSA, we will now compare both and draw some conlcusions. The answer to the question of which one of algorithms is less costly will vary depending on the security level and the operation. Figure **??** 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 opeartion, those two lines overlap in the graph.

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

- RSA is always less costly at signature verficiation
- 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 analayzing tables **??**, **??**, **??** and **??** which show the relative and absolute cost increases for RSA and ECDSA opeartions, repectively, we can see that as the secrity level increases, the cost of RSA operations increases more and more, while after the **normal** security level, the absolute cost increase of ECDSA operations starts going down. Although not presented here, our tests show that this trend holds true for higher security levels as

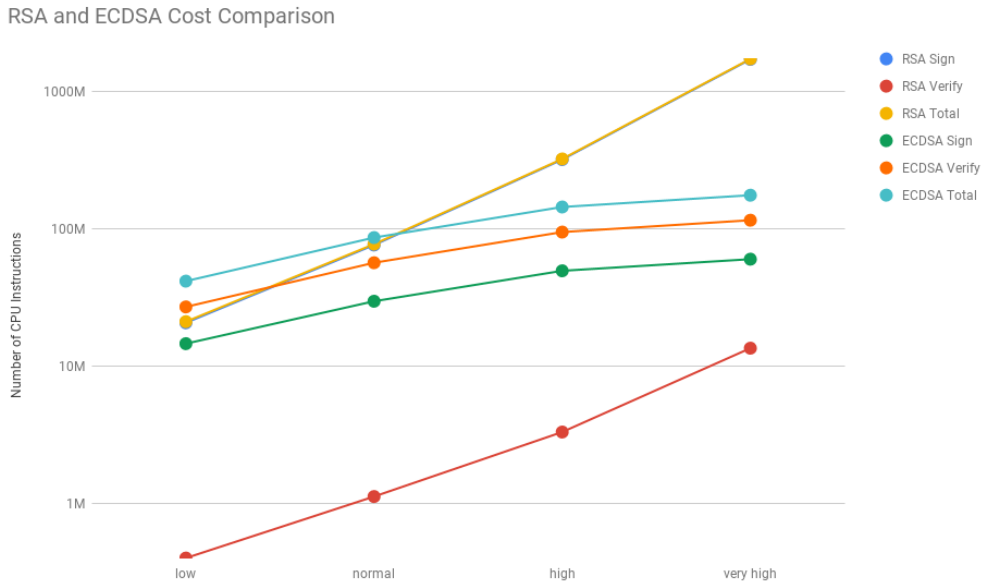


Fig. 15: RSA and ECDSA cost comparison

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' cost incrases exponentially due to the mathematical operation at the algorithm's core: modular exponentiation. Similarly, the cost increase in ECDSA is logarithmic, due it's 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 straighforward 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, RSA would be the least cotsly 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 possbible, ECDSA is the algorithm to use.

This information can also be used to make certificate choices for mutual authentcation 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 Authenticaition in TLS Having analyzed the costs of the algorithms that can be used for authentcation, we will now describe the cost of this security service

for each one of the ciphersuites for the client and the server. Table **??** shows authentcation costs for each ciphersuite for the client. Table **??** shows the same information for the server. Each row specifies a key exchange method and each column the security level.

	low	**normal**	**high**	**very high**
PSK	0	0	0	0
RSA	940875	2622235	7462989	28716994
RSA-PSK	940875	2622235	7462989	28716994
ECDH-RSA	398584	1117868	3295296	13436728
ECDH-ECDSA	26839273	56260702	94077150	114744021
ECDHE-PSK	0	0	0	0
ECDHE-RSA	797168	2235736	6590592	26873456
ECDHE-ECDSA	53678546	112521404	188154300	229488042
DHE-PSK	0	0	0	0
DHE-RSA	797168	2235736	6590592	26873456

Table 9: Client authentcation costs for all ciphersuites and security levels

	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 10: Server authentcation costs for all ciphersuites and security levels

Figures **??** and **??** are a graphical representation of tables **??** and **??**, 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 authentcation cost. For the client, those groups are:

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

For the server, those groups are:

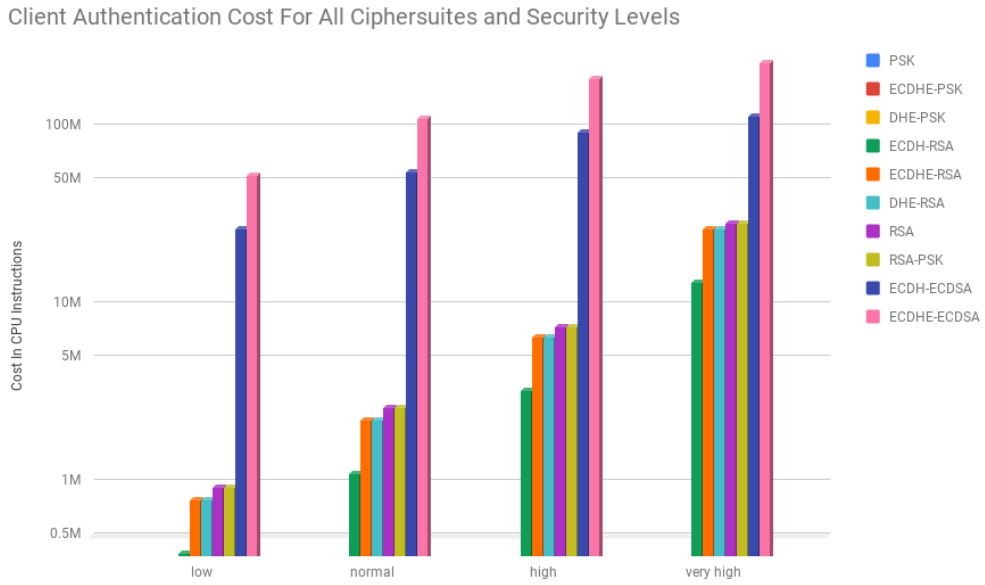


Fig. 16: Client authentcation cost (logarithmic scale)

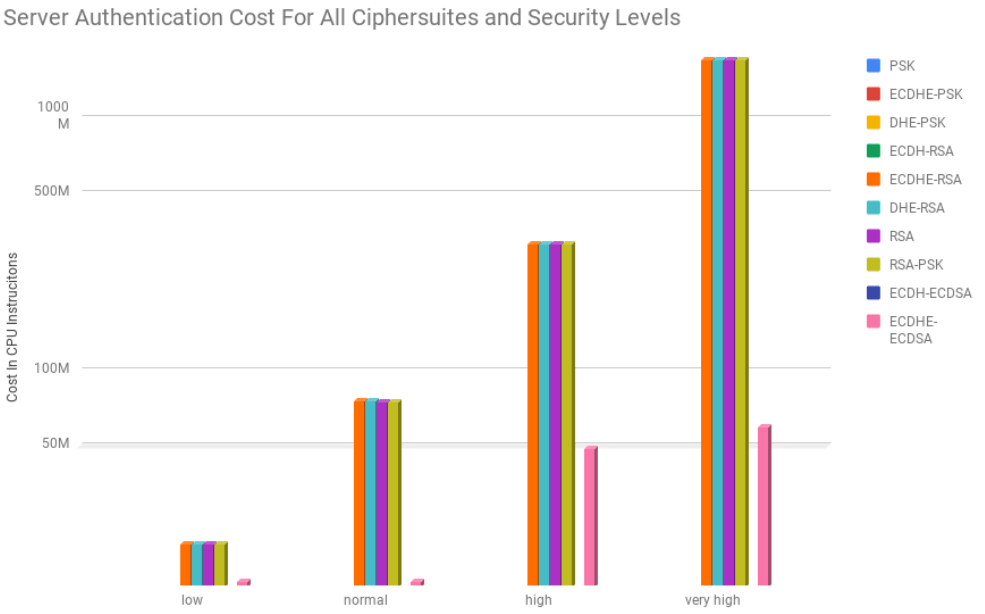


Fig. 17: Server authentcation cost (logarithmic scale)

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 authentcation cost is the same, no matter 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 authentcation. We will discuss what those operations are further down in the text. Each ciphersuite uses either PSK, RSA or ECDSA for authentcation. If only PSK is used for authentcation, the cost of authentcation 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 authentcation costs in TLS goes in hand with that discussion. By analyzing tables **??**,**??** and figures **??**, **??**, 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 possbile among the peers, ECDSA is the preferred choice. Having presented the costs of authentcation 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 of singature creation cost for the server and a 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 **??** to estimate them.

All RSA server certificates are signed with a 2048 bit RSA key and all ECDSA ceritifcates 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 **??** 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*[**?**] encrypiton scheme to encrypt the 48 byte premaster secret. The server uses the corresponding *PKCS#1 v2.1 RSAES-PKCS1V1_5DECRYPT*[**?**] decryption scheme to decrypt it. The cost 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 **??**, the ones from table **??** 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 authentcation process: 23 *RSA* ciphersuites and 15 *RSA-PSK* ciphersuites. The values in table **??** are an average of 38 runs: one for each ciphesuite. The numbers in parenthesis is the standard deviation.

The encyption 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 perfrom different steps, depending on the ciphersuite. More specifically:

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

Table 11: Cost of using *PKCS#1 V2.1 RSAES-PKCS1-v1_5* encrypiton and decryption schemes with various security levels

- in *PSK*, *DHE-PSK* and *ECDHE-PSK* the authentcation is done exclusively through the pre-shared secret, without any operations, thus the authentcation 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 perfrom 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 perfrom an ECDSA signature over the hash of *(EC)DH* parameters.

In order to compute the authentcation cost for each peer, we use the values from tables **??** and **??**, 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 verify two ECDSA signatures: one from the parsed server's certificate and one from the *ServerKeyExchange* message, while the server will only need to perform a signature over the *ECDHE* parameters. Thus, the authentcation cost for the client will be $56260702 + 56260702 = 112521404$ and for the server 29512991. Similarly, when an *RSA* or *RSA-PSK* ciphesuite 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 authentcation cost for the client will be $1117868+1504367 = 2622235$ and for the server 75129504. The remaining entries in tables **??** and **??** are computed in a similar manner.

Since in our evaluated scenario only the server authenticates, tables **??** and **??** are non-identical. If mutual authentcation was used (*i.e.* with both, the client and server authenticating one to another), the client and server's table would both be similar to Table **??**.

8.5 Perfect Forward Secrecy Cost

In TLS there are two ways of achieving PFS: either by using the DH algorithm or its ECC counterpart ECDH. In section **??** 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 analyse 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 ciphersuties 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 **??**. 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).

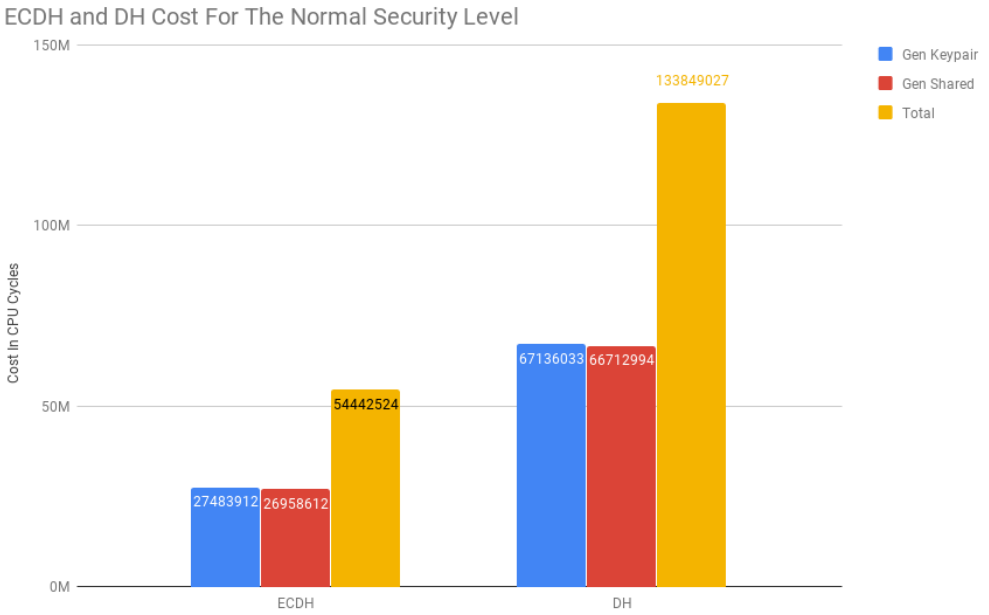


Fig. 18: ECDH and DH operations cost for normal security level

Figure **??** 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 one of them, those operations are the same for generating the keypair and the shared secret. Thus, it's 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's 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** secrurity level and now we will analyse the remaining ones. Table **??** shows the number of CPU instrucitons 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 opeartions. The numbers presented in parenthesis is the standard deviation. All of the values are rounded up to signinficant digits. Table **??** 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 curves group.

Figure **??** depicts the cost of ECDH operations graphically. For all security levels the total cost is almost evenly divided bewtween 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 incrase for all operations.

Table **??** shows the relative cost increase of ECDH operations from the previous security level, with figure **??** showing this same information graphically or the *Total* cost increase (the blue line). We're only presenting the the *ECDH Total* column in the graph, because all are very similar, thus the 3 lines would covering one another. The total relative cost increase represents the average of the values of keypair and shared secret generation relative increase. By analyzing the table and the corresponding graph, we can see that as the security level increases, the cost increase from the prevoius security level becomes smaller. The effect decrease on the absolute cost increase can be seen in table **??**. This table shows

	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 12: ECDH and DH costs for all secruity levels

	ECDH Total	DH Total
low	25405195	20734678
normal	54442524	133849027
high	90231688	948573054
very high	90231688	7178955325

Table 13: ECDH and DH costs of the sum of keypair and shared secret generation

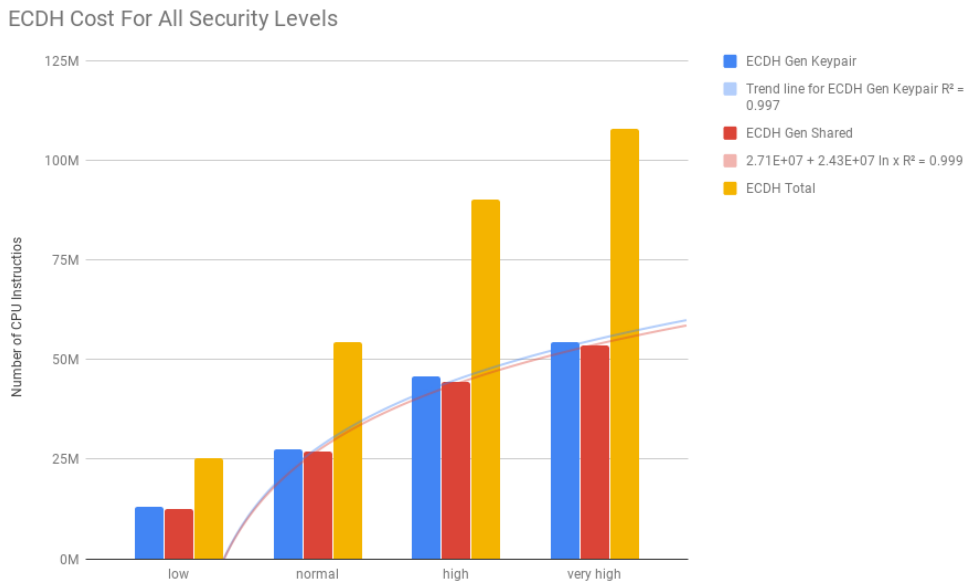


Fig. 19: ECDH operations costs for all security levels

the absolute increase in the number of CPU instructions 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 tables **??** and **??**, we can see that the numbers are very similar. This similarity also reflects in the trend that can be seen in tables **??** and **??**, 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 opeartion.

	ECDH Gen Keypair	**ECDH Gen Shared**	**ECDH Total**
low	-	-	-
normal	112.3%	116.3%	114.3%
high	67%	64.4%	65.7%
very high	18.6%	20.8%	19.8%

Table 14: Relative increase of ECDH operation costs from previous 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 15: Absolute increase of ECDH operation costs from previous security level

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's 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 mudular exponentiations.

We will once again refer to table **??** for the cost analysis, but this time focusing on DH. Figure **??** shows the costs of DH's operations for all security levels graphically. Just like in ECDH, the total cost is almost evenly devided 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 incrase for all operations.

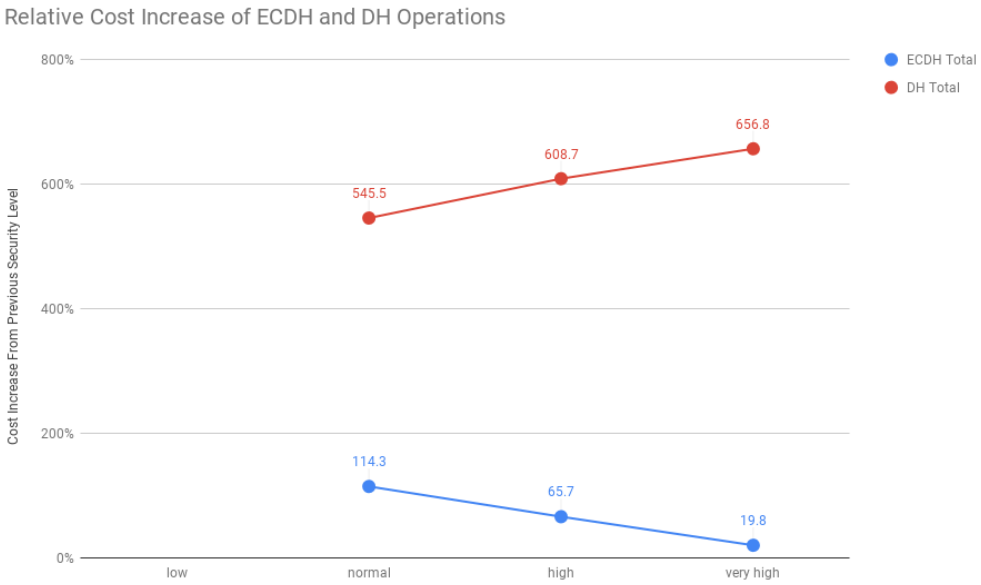


Fig. 20: Relative increase of ECDH and DH operation costs

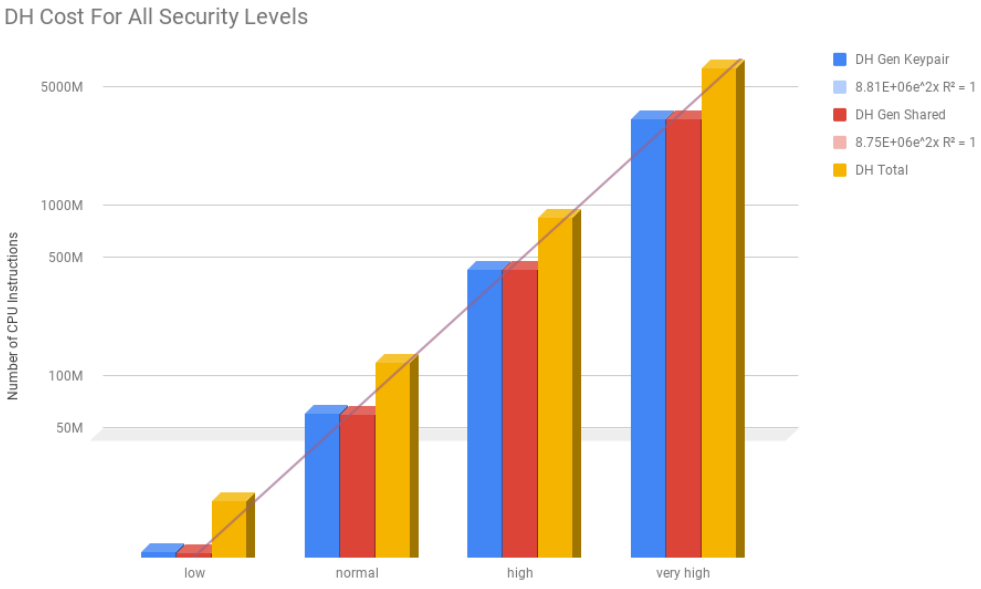


Fig. 21: DH operations costs for all security levels

Table **??** shows the relative cost increase of DH operations from the previous security level. The *Total* cost increase is depicted graphically in figure **??** (red line). We're only presenting the the *DH Total* column in the graph, because all are very similar, thus the 3 lines would covering one another. The total relative cost increase represents the average of the values of keypair and shared secret generation relative increase. An analayzing of table **??** and figure **??** shows as the security level increases, the relative 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 **??**, which shows the absolute cost increase from the previous security level.

DH Gen Keypair	**DH Gen Shared**	**DH Total**	
low	-	-	-
normal	542.1%	549%	545.5%
high	607.4%	610%	608.7%
very high	656.4%	657.2%	656.8%

Table 16: Relative increase of DH operation costs from previous security level

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

Table 17: Absolute increase of DH operation costs from 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 analysed 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 **??**, the answer to which one of the algorithms is less costly, is straighforward. 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 client or the server. 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 analysing table **??** and choosing which algorithm has the smallest value for each security level. However, in order to simplify the analysis, we have plotted the table **??** in figure **??**, which presents the data in logarithmic scale.

By looking at figure **??** it's 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 opeartions 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 line.

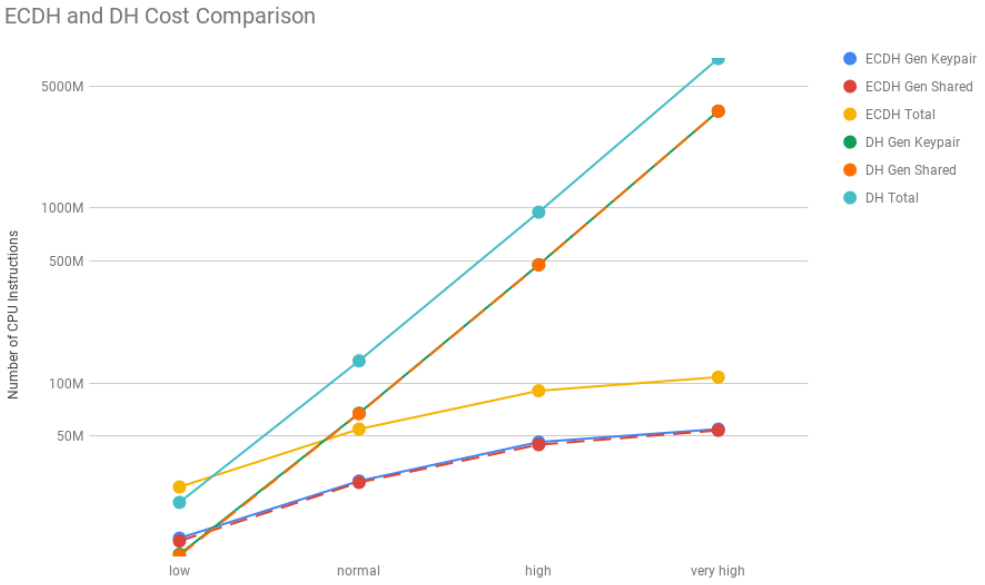


Fig. 22: ECDH and DH cost comparison (logarithmic scale)

Although not presented here, the logarithmic and exponential growth trends for ECDSA and DH, respectively, hold true even for security levels higher than **very high**. For security levels above **low**, from the costs and security prespective, there are no reasons of chooisng 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 footpirnt will be smaller.

(EC)DH algorithm and (EC)DH ciphersuites 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 *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* ciphersuties, 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 cretificate. 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 ther server's public key is fixed within its certificate. Thus in *ECDH* ciphersuites, if the server's long-term shared secret, *i.e.t* it's private ECC key, is compomised the previous comunication's secrets can be computed (if the client's public ECDH paramters that are sent in the clear are recorded), thus also compromising the confidentiality of the past communications. The destinction 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*.

8.6 TLS Handshake Cost Analysis

Here, describe the costs of each individual ciphersuite. Remember that we described the steps needed for authentcation and PFS previously. What we do here, is sum them up and then say that the costs are somwhat off the actual handshake values, because of some "hidden costs". Describe those hidden costs and then show that the values come pretty close.

The Hidden Costs For the client, there is an additional cost of parsing the *der*-encoded server certificate. The certificate are 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 **??** contains the cost of parsing the ceritificate for each algorithm and security level. This is the cost of parsing the *der*-encoded certificate into internal fields and does not include verifying the ceritifcate's signature. 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.

	low	**normal**	**high**	**very high**
RSA	133166 (12026)	136025 (10943)	141334 (4191)	157351 (1778)
ECDSA	149768 (5318)	153289 (2754)	170559 (1714)	156071 (14)

Table 18: Cost of parsing the *der* encoded certificate

Perfect Forward Secrecy In TLS, PFS is achieved either by the use of DH or ECDH key exchange. The average costs for the client and the server are the same: **X million CPU instructions. Only key exchanges which begin with *DHE-* and *ECDHE-* offer PFS. *ECDH-* ciphersuites have the server's public ECDH parameter fixed, thus such connections do not offer PFS.

8.7 Security Services Analysis

TODO: remove this seciton.

In this section, we will do a more detailed analysis of the the costs of authentication and PFS. We will begin by comparing the costs of RSA and ECDSA for private and public key operations at various security levels.

Next, the cost of PFS will be analyzed. For both, DH and ECDH we will deconstruct PFS into individual steps and examine their costs. This analysis will be done for various security levels.

The analysed operations will be fitted into the TLS Handshake process, clarifying its costs in the section that follows this one.

8.8 Handshake Protocol Analysis

The cost of making or verifying this signature is the difference between the *ECHE-** and its *ECDH-**. For example, ...

8.9 Key Exchange

* Show a graph with all of the ciphersuites * Narrow down to ciphersuite with unique key exchange (all use the same hash and encr func) * Do a high-level overview analysis of each sost (e.g. we can see that PSKs are the most efficient ones, ECC counterpart of DH is more efficient) * Note, that I will be profiling key exchanges * add full handshake graphs for other key exchange methods (only select few ciphers)

The *mbedTLS* library contains a total of 161 ciphersuites. Out of those 161 ciphersuites, there are 10 unique key exchange methods. As described in the Methodology section, in our evaluation we are using a typical TLS set up, *i.e.* with server-side authentication only. This means that the client and server evaluations results will be different. For this reason, for each part, there will be a subsection for the client and subsection for the server.

If the same type of authentication was used on the client-side, or if it was not used at all, both of the graphs would be identical. Some differences would be present, but they are negligible. Those differences could arise from factors such as implementation details, public key pairs (*e.g.* one peer's keys resulting is faster computations) and protocol-specific factors (*e.g. ClientKeyExchange* message is always sent, while *ServerKeyExchange* might be omitted; distinct internal functionality of the client and the server). For all intents and purposes, we can neglect those differences.

Figure **??** shows the the cost of the Handshake for each one of the ciphersuites for the client. Figure **??** shows the same information for the server. The graphs present a different shape. This is expected due to the use of public key cryptography. The cost of public and public key operations are different. With RSA, public key operations are generally faster than private keys ones, for ECC, it's the contrary. This is confirmed by both, our findings, and existing literature [**?**]. The *PSK* key exchange does not use asymmetric cryptography, for this reason it has identical costs for both peers.

By looking at the y axis of the client and server graphs, 8 different groups can be distinguished. Each one of them represents a key exchange. Those groups are depicted with the red color. As previously mentioned, *mbedTLS* has 10 distinct key exchange methods. However, since some key exchange methods have very similar operations involved, the costs for their cost is almost the same, thus only 8 of them stand out in the graphs. In particular, for the client, the cost of the *ECDHE-RSA* and *ECDH-RSA*, as well as the cost of *RSA* and *RSA-PSK* are very similar. At the server, there is an analogous situation for

the key exchanges *ECDH-RSA* and *ECDH-ECDSA* and the key exchanges *RSA* and *RSA-PSK*. Those observations are security-level specific and the situation might be different for different security levels. The reasons for that will be explained later in this section.

A TLS ciphersuite specifies the key exchange algorithm, the symmetric encryption algorithm and the hash function that will be used in HMAC. During the Handshake only one message is encrypted (*Finished*) and the hash function is used a few times: to generate the keying material and in the *Finished* message (the exact number of uses depends on the ciphersuite, but it is negligible). Thus, everything besides the key exchange will have almost no impact impact on the overall cost of the Handshake.

For a normalized comparison of the key exchange methods that follows, the symmetric encryption algorithm will be fixed to *AES_128_GCM* and the hash function to *SHA256*. This is the preferred choice of *Google Chrome 67*.

8.10 RSA Key Exchange

In RSA key exchange, the server authenticates himself to the client through an *X.509* public key certificate containing its public RSA key.

The most relevant parts of the RSA key exchange are as follows:

1. The client verifies the server's certificate [*RSA CA Cert Verify*]
2. The client generates a 48 byte premsater secret [*RSA PMS Gen*
3. The client encrypts the premaster secret with the server's public RSA key [*RSA Public*]
4. The server decrypts the premaster secret with its private RSA key [*RSA Private*]
5. Both peers generate the keying material [*Keying Gen*]

The costs in **millions CPU instructions**, for 2048 bit RSA level are as presented in Table **??**

	Millions CPU Instructions
RSA CA Cert Verify	1
RSA PMS Gen	0.377
RSA Public	1
RSA Private	74
Keying Gen	0.696

j

Client The total handshake cost for the client is XXXX.

8.11 DHE-RSA Key Exchange

RSA with Ephemeral Diffie Hellman key exchange.

8.12 ECDHE-RSA Key Exchange

RSA with Elliptic Curve Ephemeral Diffie Hellman key exchange

8.13 ECHD-RSA Key Exchange

RSA with Elliptic Curve Diffie Hellman key exchange (non-ephemeral).

8.14 ECDHE-ECDSA Key Exchange

ECDSA with Elliptic Curve Ephemeral Diffie Hellman key exchange.

8.15 ECDH-ECDSA Key Exchange

ECDSA with Elliptic Curve Diffie Hellman key exchange.

8.16 PSK Key Exchange

Pre Shared Key key exchange. * tell that PSK key size does not affect the results much and explain why

8.17 DHE-PSK Key Exchange

Pre Shared Key with Ephemeral Diffie Hellman key exchange.

8.18 ECDHE-PSK Key Exchange

Pre Shared Key with Elliptic Curve Ephemeral Diffie Hellman key exchange.

8.19 RSA-PSK Key Exchange

Pre Shared Key with RSA key exchange.

8.20 DHE vs ECDHE

Comparison of DHE and it's ECC counterpart ECDHE. Analyze by how much one is more expensive than other and why. Analyze this difference for differen security levels.

8.21 RSA vs ECDSA Signing

Comparison of RSA and an ECC alternative ECDSA. Compare the costs of signing with public and private key. Tell how signature verification (public key operation) is always less expensive in RSA (thus, for consrained client scenario, we will probably prefer RSA). Compare the total and signature making/varification costs for many security levels (put all in a graph). Analyse the costs of RSA and ECDSA individually, then compare.

8.22 Authenticaition

Desribe the cost of authentcation.

* Mention/predict how the size of the chain would affect all of this (i.e. tell a minimal-size chain is preferred)

8.23 Perferct Forward Secrecy

* How does ECC curves affect PFS level * The cost of PFS depends on two factors: curve and signature algorithm used to sign the SKE message. * Analyze the cost of PFS, for differnt security levels, etc. Tell why it still makes sense to use ECDH, even though it does not provide PFS. * do an ECHDE vs ECDH comparison

8.24 Confidentiality

Confidentiality

8.25 Integirty

TLS uses HMAC for integrity.

8.26 Conclusions

Tell that for constrained client scenario we want to use XYZ set-up, for constrained server scenario, we want to use ABC set-up and when both are constrained, use PSK (example). Tell that overall, PSK is the most efficient solution (if we want all security services). Put other conclusions...

9 Discussion

TODO

10 Further Work

TODO

11 Conclusion

The lack of security in IoT is a serious issue that can lead to a high monetary costs, when botnets infect the devices. Recent attacks clearly show that serious damage can be caused. 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.

While there are well established security solutions, not all of them can be used with IoT devices, due their constrained nature. One such example is the (D)TLS protocol, that because of its heavyweight nature is not suitable for a large part of IoT devices. With the proposed work, we want to contribute to this area, by designing a solution that is suitable for the IoT devices, transparent to the programmer and provides security services adaptable to the specific context needs.