Table of Contents

TLS For IoT

Transport Layer Security Protocol For Internet Of Things

Illya Gerasymchuk
`illya.gerasymchuk@tecnico.uliboa.pt`,
WWW home page: `https://iluxonchik.github.io/`

Instituto Superior Tcnico
Supervisors: Ricardo Chaves, Aleksandar Ilic

Abstract. The abstract should summarize the contents of the paper using at least 70 and at most 150 words. It will be set in 9-point font size and be inset 1.0 cm from the right and left margins. There will be two blank lines before and after the Abstract. . . .

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

1 Intoduction

TODO: Intruduce the topic: explain what is IoT; what is TLS; what are the issues with using RAW TLS with IoT(power, computation, limited resources).

TCP performance is known to be inefficient in wireless networks, due to its congestion control algorithm, and the situation is exacerbated with the low-power radios and lossy links found in sensor networks. Therefore, the connectionless UDP is mostly used in the IoT.

1.1 Goals

2 Symmetric vs Asymmetric Cryptography (Background Section)

Asymmetrical Cryptography (AC) more expensive than Symmetrical Cryptography (SC) in terms of performance. This is mainly due to two facts: larger key sizes are required for AC system to achieved the same level of security as in a SC system and `CPU`s are slower at performing the underlying mathematical operations involved in AC, namely exponentiation requires `O(log e)` multiplications for an exponent `e`. For example, the 2016 `NIST` report [6] suggests that an AC 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 Elliptic Curve Cryptography (ECC), which requires keys of `512 bits`, it is still slower than SC, though. The 2017 `BSI` report [3] (from the German federal office for information security) suggests similar numbers.

Another argument for avoiding as much as possible of the AC functionality is that it requires extra storage space to be used and this might be a problem for some Internet Of Things (IoT) devices, like `class 1` devices according to the terminology of constrained-code networks [15] which have about `10KB` of RAM and `100KB` of persistent memory. I measured the resulitng size of the complied mbedTLS 2.6.0 library [9] when compiled with and without the `RSA` module (located in the `rsa.c` file), from which I concluded that using that module adds an extra of `32KB`.

3 Related Work

Lightweight cryptography is an important topic in the context of IoT, since cryptography a fundamental part of security and it's fundamental for it to be lightweight in order to run on devices with limited memory and processing capabilities. A lot of work in IoT incorporates it in one way or another, so I'll begin with the work done in this area.

Alex *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. They also propose to split the field into two areas: ultra-lightweight and IoT cryptography.

The authors systematized the knowledge in the area of lightweight cryptography in order to define "lightweightness" more precisely. They observed that the design of lightweight cryptography algorithms vary greatly, the only unifying thread between them being the low computing power of the devices they're designed to run on.

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 quantiying how physically large a circuit implementing 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 per CPU cycle.

To acommondate the limitations of 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 use 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 smaller messages.

Even though though algorithms can be optimized in implementation: whether it's a software or a hardware now, dedicated lightweight algorithms are still needed. This comes down mainly to two factors: there are limitations to the the extent of the optimizations that you can make and the hardware-accelerated encryption is frequently vulnerable to various Side-Channel Attack (SCA)s (such as the attack done on the Phillips light bulbs [22], where the authors were able to recover a secret key used to authenticate updates, via an SCA).

It's 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're 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 bitate. 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 small 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 footpirnt at the cost of slower data processing. A capacity of `128 bits` and a bitrate of `8 bits` are common values for lighweight 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 algorithms" defintion into two distinct fields has been proposed:

– **Ultra-Lightweight Crypto** - algorithms running on very cheap devices which are **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 work focuses on **IoT Crypto** devices. A summary of differences between the both categories is summarized in table 1.

TODO: talk about RAW PK

For the reasons specifed above, the main key exchange mechanism used in IoT are Pre-Shared Key (PSK)s. SK3[**?**] proposes a key management architecture for resource-constrained devices, which allows devices that have no previous, direct security relation to use TLS or DTLS using one of two approaches: shared symmetric keys or raw public keys. The resource-constrained device is a server

Table 1. 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

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[**?**], without requiring 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 is used in the same way as defined in `RFC PSK`[**?**]. When connecting to the server, the client will then send the `psk_identity` that it received in the the (D)TLS handshake and the server will derive the `Kc`, using the `P_hash()` function defined in [17].

The second approach consists in the requesting an APK (the authors never defined what this acronym stands for, but I assume they mean "Authorization Public Key") from the TA. In his request, the client includes his Raw Public Key (RPK), which is used for authorization. The TA creates an authorizaton certificate, protects it with a MAC and sends it to the client alongside the server's Public Key (PubK). The client then sends this APK (instead of the RPK) when connecting to the server, which verifies the APK (to authorize the client) and proceeds with the handshake in the RPK mode, as defined in [**?**]. To achieve this, a new certificate structure is defined, alongside a new `certificate_type`. The new certificate strucure is just the RFC7250 [25] structure, with an additonal MAC.

The has 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 `ChangeCiperSpec` message which required the longest 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 and Next Header Compression (NHC) for the IP extension headers and the UDP header in [**?**]. The compression relies on the shared context between the communicating peers.

[1] 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, there is however, a proposed standard[14] for generic header compression for 6LoWPANs that can be used to compress the UDP payload. The authors propose a way to compress DTLS headers and messages using this mechanism.

The paper [1] defines how the Datagram TLS (DTLS) Record header, the DTLS Handshake header, the ClientHello and the ServerHello messages can be compressed, but notes that the same compression techniques can be used to compress the remaining Handshake messages. They explore two cases for the header compression: compressing both, the Record header and the Handshake header and compressing the Record header only, which is useful after the handshake has completed and the fragment field of the Record layer contains application data, instead of a handshake message.

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

Fig. 1. IPv6 Next Header Compression

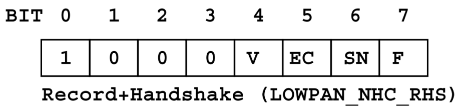


Fig. 2. LOWPAN_NHC_RHS structure

All of the cases follow the same basic idea, for this reason I'll only exemplify one of them. Each DTLS fragment is carried over in the UDP payload. In this case, the UDP payload carries a header-like payload (the DTLS record header). Figure 1 shows the way IPv6 next header compression is done. The authors use the same value for the `LOWPAN_NHC Encoding` field (defined in [**?**]) as in `RFC7400` and define the format of the `In-line Next Header Fields` (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.

I will exemplify 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 2), 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're fixed to `1000`, so that 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're not going to be sent at all (*i.e.* they're 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're 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[4], which is a lightweight security solution for Constrained Application Protocol (CoAP) that uses the same DTLS header compression technique as in [1] with the goal of implementing it as a security support for CoAP.CoAP[16] is a specialized RESTful Internet Application Protocol for constrained devices. It's 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". CoAP 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.

The authors evaluated their system in a simulated environment in Contiki OS and they obtained significant gains in terms of packet size (similiar numbers to the ones obsever in [1]), energy consumption (in average `15\%` less energy is used to transmitting 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* [8] proposed to integrate the DTLS protocol inside the CoAP, while also exploiting ECC optimizations and minimizing ROM occupancy. They've implemented their solution on 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's 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[7], 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[16], 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 Elliptic Curve Diffie-Hellman (ECDH) and Elliptic Curve Digital Signature Algorithm (ECDSA) rely upon. The first optimization had to do with modular arithmetic on large integers. A set of optimized assembly routines based on [2] allow the improved use of registers, reducing the number of memory operations needed to perform operations 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 [5], which is based on precumputation of a set of Discrete Log pairs. I will refrain from going into the mathematical details, since they're not relevant for this description. The *IBPV* technique was used to improve the performance of the ECDSA signature and was also extended to the ECDH protocol. In order to reduce the time it takes to do the 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%.

4 Background

TODO: Tell that first I describe the parts of TLS that are common to both and then specialize for TLS 1.2 and TLS 1.3

5 The TLS Protocol

TLS stands for Transport Layer Security, it's 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, Transport Layer Security (TLS) is placed between the **Transport** and **Application** layers. It's designed to make the application developer's life easier: all the developer has to do is create a "secure" connection, instead of a "normal" one.

TODO: Re-write what's below. It's good to include something like this, but I need to work on the wording. From the top-level view, in a typical connection, there are three basic steps that TLS is responsible for:

1. **Negotiate security parameters** - the communicating peers agree on a set of security parameters to be used in a TLS connection, such as the algorithm used for bulk data encrytion, as well as the secret keys.
2. **Authenticate one to another** - usually only the server authenticates to the client.
3. **Communicate securely** - use the negotiated security parameters to encrypt and authenticate the data, communicating securely one with another.

SSL vs TLS: What's The Difference? You will find the names Secure Sockets Layer (SSL) and TLS used interchangeably in the literature, so I think it's important to distinguish both. TLS is an evolution of the SSL protocol. The protocol changed its name from SSL to TLS when it was standardized by the Internet Engineering Task Force (IETF).SSL was a proprietary protocol owned by Netscape Communications, and The IETF decided that it was a good idea to standarize it, which resulted in RFC 2246 [18], specifying TLS 1.0, which was nothing more than a new version SSL 3.0, very few changes were made. In this document, I'll be concentrating on TLS 1.2 and TLS 1.3 protocols. The first one is the most recently standardized version of TLS and the latter is currently and in-draft version with many improvements and optimizations relevant for the topic of this dissertation. Despite the protocol name not suggesting it TLS 1.3 is very different from TLS 1.2, in fact, it should've probably been called TLS 2.0 instead. For this reason, I will first describe what is common to both protocols and then go into the relevant details about each one.

TODO: Explain what RFCs are?

5.1 Security Services

TLS provides the following 3 security services:

- **authentication** - both, **peer entity** and **data origin** (or **integrity**) authentication.
  - **peer entity authentication** - we can be sure that were talking to certain entity, for example, www.google.com. This is achieved thought the use of **asymmetrical** or Public Key Cryptography (PKC) (for example, RSA and DSA) or **symmetric key cryptography**, using a PSK.

- **confidentiality** - the data transmitted between the communicating entities (the client and the server) is encrypted. Symmetric cryptography is used of data encryption (for exmaple, `AES`).
- **integrity** (also called **data origin authentication**) - we can be sure that the data was not modified or forged, *i.e.*, be sure that the data that were receiving is coming from the expected entity (for example, we can be sure that the `index.html` file sent to us when we connected to `www.google.com` in fact came from `www.google.com` and that it was not modified (i.e tampered with) en route by an attacker (**data integrity**). This is achieved through the use of a keyed Message Authentication Code (MAC) or an Authenticated Encryption With Associated Data (AEAD) cipher.

Despite using PKC, TLS does **not** provide **non-repudiation services**: neither **non-repudiation with proof of origin**, which addresses the user denying having sent a message, not **non-repudiation with proof of delivery**, which addresses the user 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 **shared secret** to be used.

You are not required to use all of the 3 security services in every situation. You can think of TLS as a framework that allows you to select which security services you want to use for a communication session. As an example, you might ignore certificate validation, which means you're ignoring the **authentication** guarantee. There are some differences regarding this claim between TLS 1.2 and TLS 1.3, for example, while in the first you have 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.

Cipher Spec vs Cipher Suite The meaning of these terms differs in TLS 1.2 and TLS 1.3. For TLS 1.2, **cipher spec** defines the message encryption algorithm and the message authentication algorithm, while the **cipher suite** is the **cipher spec**, alongside the definition of the **key exchange** algorithm and the Pseudo-Random Function (PRF) (used in key generation). In TLS 1.3, the **cipher spec** has been removed altogether, since the **ChangeCipherSpec** protocol has been removed. The concept of **cipher suite** has been updated to define the pair of AEAD algorithm and 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. You'll find more details on this below.

5.2 TLS (Sub)Protocols

In reality TLS is composed of several protocols(illustrated in 4), a brief description of each one of which follows:

- **TLS Record Protocol** - the lowest layer in TLS. It's the layer that runs 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: a 1-byte record `type`, specifying the type of record that's encapsulated (ex: value `0x16` for the handshake protocol), a 2-byte `TLS version` field, a 2-byte `length` field (which means that a maximum TLS Record size is of `16384` bytes), specifying the length of the data in the record, excluding the header itself and a `fragment` field whose size in bytes is specified by the `length` field, which contains data that's transparent to the Record layer and should be dealt by a higher-level protocol, specified by the `type` field. This is illustrated in figure **??**.

- **TLS Handshake Protocol** - the core protocol of TLS. Allows the communicating peers to **authenticate** one to another and negotiate a **cipher suite** (**cipher suite** and key exchange algorithm in case of TLS 1.3) which will be used to provide the security services. For TLS 1.2, **compression** method is also negotiated here.
- **TLS Alert Protocol** - allows the communicating peers to signal potential problems.
- **TLS Application Data Protocol** - used to transmit data securely.
- **TLS Change Cipher Spec Protocol** (removed in TLS 1.3) - used to activate the initial **cipher spec** or change it during the connection.

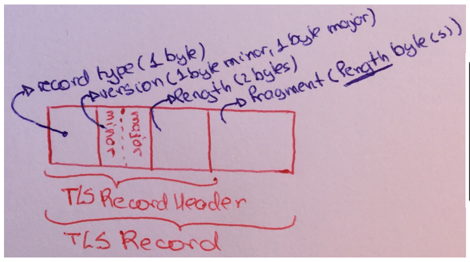


Fig. 3. TLS Record header

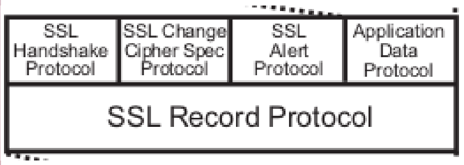


Fig. 4. TLS (Sub)protocols and Layers

TLS Connections and Sessions TODO: define what it means to be cryptographically protected?

It's important to distinguish between a **TLS session** and a **TLS connection**.

- **TLS sesion** - assosciation between two communicationg peers that's created by the **TLS Handshake Protocol**, wich defines a set of negotiated paramters (cyrptographic and others, depending on the TLS version, such as the compression algorithm) 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 purpuse is to avoid the expensive negotiation of new parameters for each **TLS connection**. For example, let's say you download an Hypertext Markup Language (HTML) page over Hypertext Transfer Protocol Secure (HTTPS) and that page referrences some images from that same server, also using HTTPS, instead of your web browser negotiating a new TLS session again, it can re-use the the one you established to download the HTML page in the first place, 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` [17], **session tickets**, defined in `RFC 5077` [23]. TODO: Re-write example better.

– **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**.

TLS Record Processing A TLS record must go through some processing before it can tbe sent over the netwrok. This processing involves the following steps (4 for TLS 1.2 and 3 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 `TLS Plaintext`. 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 the same as `TLSPlaintext`.
3. **Cryptographic Protection** - in case of TLS 1.2, either an AEAD cipher or a separate encryption and MAC functions transform a `TLSCompressed` fragment into a `TLSCipherText` fragment. In case of TLS 1.3, the `TLSPlaintext` fragment is transformed into a `TLSCipherText` by applying an AEAD cipher.
4. Append the **TLS Record Header** - encapsulate `TLSCipherText` in a `TLS Record`.

The process described above, as well as the structure names are depicted in figure 5. Step 2 is not present in TLS 1.3. The structure names are exactly as the appear in the TLS specifications.

5.3 TLS Keying Material

Secret keys are at the base of most cryptographic operations. In order for both com-

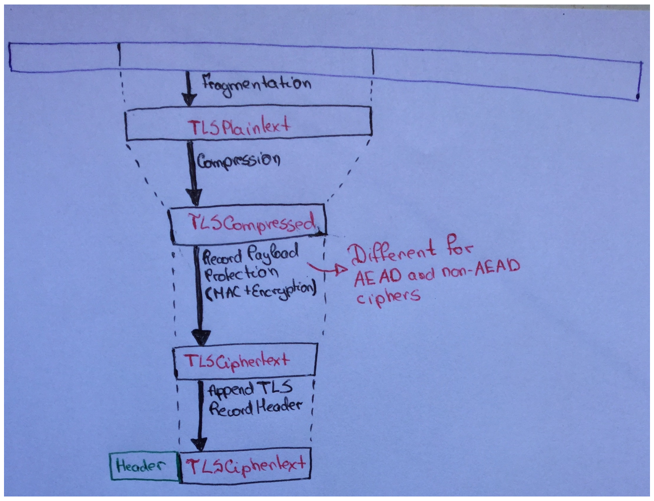


Fig. 5. TLS Record Processing

municating peers to be able to encrypt and decrypt data using symmetric cyrpto aglorithms, they need to **share** the same key somehow. In TLS, both, the client and a server derive the **same set of keys** independetely, through the exchanged messages in the TLS Handshake Protocol.

When communicating with one anohter, the client uses one key to encrypt the data to be sent to the server and another different key to decrypt the data that it receives from the server. This means that in order to deal with data encryption and decryption, both of the communicating entities have two keys: one to encrypt the outgoing data and one to decrypt the incoming data. Those keys have different names in TLS 1.2 and TLS 1.3, but they serve the same basic purpose. In this general description, I'll refer to them as `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) and `server_read_key`(used by the server to decrypt the incoming data from the client). Note, that the following relationships must hold: `client_write_key == server_write_key` and `client_read_key == server_write_key`.

Besides the secret keys mentioned previously, in TLS 1.2 you might also have other ones, depending on the cipher suite in use. TODO: Describe this in a little more detail, giving examples, when describing TLS 1.2 Key Managment.

TLS 1.3's keying material generation is a little more complex, since different keys are used to encrypt data throughout the Handshake Protocol, as well a new key is generated for the Application Data protocol. This can be explained by the fact that while in TLS 1.2 the data only begins to be encrypted after the handshake is complete, in the Application Data protocol, the encryption begins earlier, TLS 1.3, with some of the Hanshake messages encrypted, as well as features such as **early client/server data** and **0-RTT Data**.

With this the common description of the TLS of protocols ends and we'll jump into the specifics of the two verions. I'll be mostly concentrating on the **Handshake Protocol**, since this is where my work will be concentrated and it's the main part, where the most interesting and important things happen.

5.4 TLS 1.2

The latest standardized version of TLS is 1.2 and it's defined in `RFC 5246` [17]. TODO: DESCRIBE TLS 1.2 in genreal, put images of handshakes here, later refer to them in the specific parts, just like the tls RFCs do.

6 TLS 1.2 Keying Material Generation

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

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

TODO: talk about all of the keys present in TLS 1.2 HERE

7 TLS 1.2 Key Exchange Methods

The way the **permaster secret** is generated depends on the key exchange method used. In fact, this is the only phase of the keying material generation phase that is variable for a fixed cipher suite (because a cipher suite defines the PRF function to be used), the rest remains exactly the same. The derivation of the **master secret** from the **premaster secret**, as well as the derivation of the bulk encryption keys, MAC keys and Initialization Vector (IV)s from the **master secret** that follows **is not impacted by the key exchange method** in use.

You have quite a few choices when it comes to key exchange methods. Some of them are defined in the base spec (RFC5246 [17]), while others in separate RFCs (such as the ECC based key exchange, specified in RFC4492 [12]).

The base spec specifies 4 key exchange methods, one using Rivest-Shamir-Adleman (RSA) and 3 using Diffie-Hellman (DH):

– static RSA (RSA) [removed in TLS 1.3] - the client generates the premaster secret (PMS), encrypts it with the server's PubK (which it obtained from the server's X.509certificate), sending it to the server, which decrypts it using the corresponding Private Key (PrivK). This key exchange method offers authenticity, but does not offer Perfect Forward Secrecy (PFS).
– anonymous DH (DH_annon) [removed in TLS 1.3] - a DH key exchange is performed and an **ephemeral** key is generated, but the exchanged DH parameters are **not authenticated**, making the resulting key exchange vulnerable to Man In The Middle (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 no 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.
– epehemeral DH (DHE) - each run of the protocol, uses different pubic DH parameters, which are generated dynamically. This results in a different, epehemeral key being generated every time. The public parameters are then digitally signed in some way, usually using the sender's private RSA (DHE_RSA)) or \gls{dsa} DHE_DSS) key. This key exchange offers both authenticity and PFS.

When either of the DH variants is used, the value resulting from the exchange is used as the PMS (without the leading `0`'s). Usually, only the server's authenticity is desired, but client's can also be achieved if it provides the server its certificate. Whenever the server is authenticated, the server is secure against MITM attacks. Table **??** summarizes the security properties offered by each key exchange method.

Table 2. Key exchange methods and security properties

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

Note that in TLS 1.3, all of static RSA and DH cipher suites have been removed: all of the PubK exchange methods now provide PFS. Even though, anonymous DH has also been removed from TLS 1.3, you can still have unauthenticated connections by either using **raw public keys** [25] or by not verifying the certificate chain and any of it's contents.

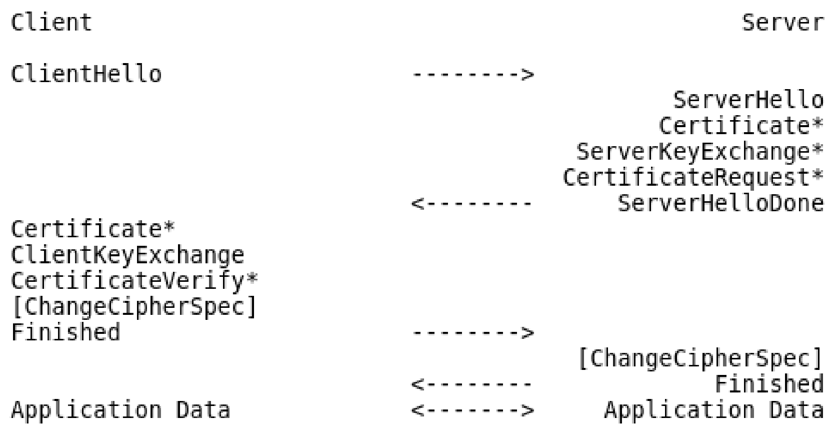
TODO: NOTE: I did't cover specifics of how the client generates the premaster secret, etc

The ECC-based key exchange (ECDH and Elliptic Curve Diffie-Hellman Ephemeral (ECDHE)) and authentication (ECDSA) algorithms are defined in `RFC4292` [**?**], which is also referenced in `RFC5246` [**?**]. 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 epehemeral (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 the TLS Extension Mechanism.

7.1 TLS 1.2 Handshake Protocol

In this phase the client and the server agree on which version of the TLS protocol to use, authenticate one to another and negotiate items like the cipher suites and the compression method to use. Figure 6 shows the message flow for the full TLS 1.2 handshake. Note that `*` indicates situation-dependent messages that are not always sent, while `ChangeCiperSpec` is a separate protocol, rather than a message type.

As I explained before every TLS Handshake message is encapsulated within a TLS Record. The actual Handshake message is contained within the `fragment` of the 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



```
Client                                               Server

ClientHello                  -------->
                                                ServerHello
                                               Certificate*
                                         ServerKeyExchange*
                                        CertificateRequest*
                             <--------      ServerHelloDone
Certificate*
ClientKeyExchange
CertificateVerify*
[ChangeCipherSpec]
Finished                     -------->
                                         [ChangeCipherSpec]
                             <--------             Finished
Application Data             <------->     Application Data
```

Fig. 6. TLS 1.2 message flow for a full handshake

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 `msh_type` (similar to `fragment` field in a TLS Record).

Now, I will describe a typical handshake message flow. I will only be mentioning the most important field of each message.

TODO: I don't have space to put all of the structures and things sent in every handshake message type (ex: ClientHello.session_id)

The connections begins with the client sending a `ClientHello`, containing .`random`, `cipher_suites` and `compresison_methods`, among other fields. A 32-byte `random` (3-bytes gmt unix time + 27 cryptographically random bytes) value that are used as an input to the PRF when generating the **master secret**, which will cause to contains a **list** of cipher suites (`cipher_suites`) and compression methods (`compression_methods`) that the client supports, ordered by preference, with the most preferred one appearing first.

7.2 Do We Really Need Two Randoms: One From Client and One From Server?

TODO: YES, replay attacks.

TODO: Mention HelloRequest

Notes and Comments. This is an example of a paragraph. Note the styling.

7.3 TLS 1.3

Despite the protocol name not suggesting it TLS 1.3 is very different from TLS 1.2, in fact, it should've probably been called TLS 2.0 instead.

How Do Peers Distinguish Different TLS Versions? TODO: Talk about version numbers

7.4 TLS Extension Mechanism

TODO: Describe the Extended ClientHello/ServerHello. Use one description for both, TLS 1.2 and TLS 1.3

7.5 The Problem With Compression In TLS

TODO: explain why compression was removed (BEAST and CRIME attacks) and how it can be fixed.

7.6 Theory

TODO: Explain: public key crypto, certificates, AEAD ciphers

References

1. 6lowpan compressed dtls for coap - ieee conference publication. http://ieeexplore.ieee.org/document/6227754/, (Accessed on 12/20/2017)
2. Comparing elliptic curve cryptography and rsa on 8-bit cpus — springerlink. https://link.springer.com/chapter/10.1007/978-3-540-28632-5$_9$, ($Accessedon$12/24/2017)
3. Kryptographische verfahren: Empfehlungen und schlssellngen, version 2017-01. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf?$_{blob=publicationFile,(Accessedon12/19/2017)}$
4. Lithe: Lightweight secure coap for the internet of things - ieee journals & magazine. http://ieeexplore.ieee.org/document/6576185/, (Accessed on 12/20/2017)
5. Low-cost standard signatures in wireless sensor networks: A case for reviving pre-computation techniques? (pdf download available). https://www.researchgate.net/publication/259811495$_Low - cost_Standard_Signatures_in_Wireless_Sensor_Networks_A_Case_for_Reviving_Pre - computation_Techniques, (Accessedon$12/24/2017)
6. Recommendation for key management, part 1: General. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf, (Accessed on 12/19/2017)
7. Rfc 7252 - the constrained application protocol (coap). https://tools.ietf.org/html/rfc7252page-8, (Accessed on 12/24/2017)
8. Security as a coap resource: An optimized dtls implementation for the iot - ieee conference publication. http://ieeexplore.ieee.org/document/7248379/, (Accessed on 12/24/2017)
9. Ssl library mbed tls / polarssl: Download for free or buy a commercial license. https://tls.mbed.org/, (Accessed on 12/19/2017)
10. Ayadi, A., Ros, D., Toutain, L.: TCP header compression for 6LoWPAN. Internet-Draft draft-aayadi-6lowpan-tcphc-01, IETF Secretariat (October 2010), http://www.ietf.org/internet-drafts/draft-aayadi-6lowpan-tcphc-01.txt, http://www.ietf.org/internet-drafts/draft-aayadi-6lowpan-tcphc-01.txt

11. Badra, M., Hajjeh, I.: ECDHE$_{P}SKCipherSuitesforTransportLayerSecurity(TLS).RFC5489, RFCEditor(March20$
12. Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., Moeller, B.: Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492, RFC Editor (May 2006), http://www.rfc-editor.org/rfc/rfc4492.txt, http://www.rfc-editor.org/rfc/rfc4492.txt
13. Blumenthal, U., Goel, P.: Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport Layer Security (TLS). RFC 4785, RFC Editor (January 2007)
14. Bormann, C.: 6LoWPAN-GHC: Generic Header Compression for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs). RFC 7400, RFC Editor (November 2014)
15. Bormann, C., Ersue, M., Keranen, A.: Terminology for Constrained-Node Networks. RFC 7228, RFC Editor (May 2014), http://www.rfc-editor.org/rfc/rfc7228.txt, http://www.rfc-editor.org/rfc/rfc7228.txt
16. Bormann, C., Shelby, Z.: Block-Wise Transfers in the Constrained Application Protocol (CoAP). RFC 7959, RFC Editor (August 2016)
17. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, RFC Editor (August 2008), http://www.rfc-editor.org/rfc/rfc5246.txt, http://www.rfc-editor.org/rfc/rfc5246.txt
18. Dierks, T., Allen, C.: The TLS Protocol Version 1.0. RFC 2246, RFC Editor (January 1999), http://www.rfc-editor.org/rfc/rfc2246.txt, http://www.rfc-editor.org/rfc/rfc2246.txt
19. Eronen, P., Tschofenig, H.: Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). RFC 4279, RFC Editor (December 2005), http://www.rfc-editor.org/rfc/rfc4279.txt, http://www.rfc-editor.org/rfc/rfc4279.txt
20. Keoh, S., Kumar, S., Shelby, Z.: Profiling of DTLS for CoAP-based IoT Applications. Internet-Draft draft-keoh-dtls-profile-iot-00, IETF Secretariat (June 2013), http://www.ietf.org/internet-drafts/draft-keoh-dtls-profile-iot-00.txt, http://www.ietf.org/internet-drafts/draft-keoh-dtls-profile-iot-00.txt
21. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. Internet-Draft draft-ietf-tls-tls13-21, IETF Secretariat (July 2017), http://www.ietf.org/internet-drafts/draft-ietf-tls-tls13-21.txt, http://www.ietf.org/internet-drafts/draft-ietf-tls-tls13-21.txt
22. Ronen, E., OFlynn, C., Shamir, A., Weingarten, A.O.: Iot goes nuclear: Creating a zigbee chain reaction. Cryptology ePrint Archive, Report 2016/1047 (2016), https://eprint.iacr.org/2016/1047
23. Salowey, J., Zhou, H., Eronen, P., Tschofenig, H.: Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077, RFC Editor (January 2008), http://www.rfc-editor.org/rfc/rfc5077.txt, http://www.rfc-editor.org/rfc/rfc5077.txt
24. Tschofenig, H., Fossati, T.: Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things. RFC 7925, RFC Editor (July 2016)
25. Wouters, P., Tschofenig, H., Gilmore, J., Weiler, S., Kivinen, T.: Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7250, RFC Editor (June 2014)

Glossary

AC	Asymmetrical Cryptography. 1, 2
AEAD	Authenticated Encryption With Associated Data. 9, 11
CoAP	Constrained Application Protocol. 6, 7
DH	Diffie-Hellman. 13, 14
DTLS	Datagram TLS. 5–7
EC	Elliptic Curve. 7
ECC	Elliptic Curve Cryptography. 1, 6, 7, 13, 14
ECDH	Elliptic Curve Diffie-Hellman. 7, 14
ECDHE	Elliptic Curve Diffie-Hellman Ephemeral. 14
ECDSA	Elliptic Curve Digital Signature Algorithm. 7, 14
HKDF	HMAC-based Extract-and-Expand Key Derivation Function. 9
HTML	Hypertext Markup Language. 11
HTTPS	Hypertext Transfer Protocol Secure. 11
IETF	Internet Engineering Task Force. 8
IoT	Internet Of Things. 2, 6
IV	Initialization Vector. 13
MAC	Message Authentication Code. 9, 11, 13
MITM	Man In The Middle. 13, 14
PFS	Perfect Forward Secrecy. 13, 14
PKC	Public Key Cryptography. 8, 9
PMS	premaster secret. 13, 14
PRF	Pseudo-Random Function. 9, 13, 15
PrivK	Private Key. 13
PSK	Pre-Shared Key. 3, 4, 8
PubK	Public Key. 4, 13, 14
RPK	Raw Public Key. 4
RSA	Rivest-Shamir-Adleman. 13, 14
SC	Symmetrical Cryptography. 1

SCA	Side-Channel Attack. 3
SSL	Secure Sockets Layer. 8
TLS	Transport Layer Security. 8–15