Table of Contents

TLS For IoT

Abstract. 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 permit the use of computationally complex and memory demanding operations present in a standard TLS implementation. Most of previous work focused entirely on DTLS and can not be easily integrated with existing deployments. This paper describes how the TLS extension mechanism can be used to define a framework that can 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 IoT is a network of devices, from simple sensors to smartphones and wearables that 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 something such as a salt shaker[1] can now be part of the global network.

The IoT technology can be used solve problems and make our lives easier, unfortunately, however, its technological development 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[2] 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, the *Reaper* botnet[3], however, could be even more devastating if it is ever put to malicious use. Others will inadvertently come in the future.

TLS is the most used security protocol in the world and it allows two peers to communicate securely. It is designed to run on top of a reliable, connection-oriented protocol, such as TCP. DTLS is the version of TLS that runs on top of an unreliable transport protocol, such as UDP. Most IoT devices have very limited processing power, storage and energy. Moreover, the performance of TCP is known to be inefficient in wireless networks, due to its congestion control algorithm and this situation is worsened with the low-power radios and lossy links found in sensor networks. Therefore, 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 also applies to DTLS as well, since it is just an adaption of TLS over unreliable transport protocols, with no changes done the core protocol.

The problem in using (D)TLS in IoT is that it is not lightweight, since it was not 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, imagine a temperature sensor that sends the 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, which can be an extra hundreds of milliseconds. Besides that, it will need to perform heavy mathematical operations involved in cryptography, using even more battery and taking even more time. 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 minimize the friction of adoption. The solution will be developed for (D)TLS versions 1.2 and 1.3. The idea is to make it customizable, depending on the security requirements of the context of its use, in that sense, it is similar to a framework.

In the process of my work on this dissertation, I have made several contributions to the TLS 1.3 specification and have been officially recognized as a contributor, my name will be on the final document specifying the TLS 1.3 protocol.

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. Finally, Section 5 provides an architecture of the solution that will be developed in the second part of the dissertation, an explanation of how the results will be evaluated, a general work plan and a conclusion of the work done in this part.

2 Background

TLS is a complex protocol that relies on various concepts to provide security. Some of them will be described here.

TLS uses Asymmetrical Cryptography (AC) for peer authentication and Symmetrical Cryptography (SC) for bulk data encryption, for this reason this topic will be covered in Section 2.1. Section 2.2 covers the most common way of peer authentication: Public Key (PubK) certificates. AEAD ciphers offer various advantages in the context of IoT, particularly less computational and spacial overhead. Furthermore, they are the only type of ciphers that can be used in TLS 1.3. For those reasons, they're covered in Section 2.3. When compared to other PubK cryptography approaches, ECC offers shorter keys, lower central processing unit (CPU) consumption and lower memory usage for equivalent se-

curity strength. It is also heavily used in TLS. For those reasons, this section ends with an overview of ECC in Section 2.4.

2.1 Symmetric vs Asymmetric Cryptography

AC is more expensive than SC in terms of performance. This is mainly due to two facts: larger key sizes are required for an AC system to achieve the same level of security as in a SC system and CPUs are slower at performing the underlying mathematical operations involved in AC, namely exponentiation requires $O(loge)$ multiplications for an exponent e. For example, the 2016 NIST report [4] 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 [5] (from the German federal office for information security) suggests similar numbers.

Another argument for avoiding as much as possible the use of AC algorithms is that they require additional storage space and this might be a problem for some IoT devices, like class 1 devices according to the terminology of constrained-code networks[6] which have approximately 10KB of RAM and 100KB of persistent memory. I measured and compared the resulting size of the complied *mbedTLS 2.6.0* library [7] when it was 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 extra of 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 PubK. This allows the other parties to rely upon assertions made by the Private Key (PrivK) that corresponds to the PubK 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 issued by a CA that is included in that device's trusted store.

In (D)TLS, the certificates are in the X.509 format, which is described in RFC 5280[8].

2.3 AEAD Ciphers

Authenticated Encryption (AE) and AEAD are forms of encryption which simultaneously provide confidentiality, integrity and authenticity guarantees on

the data. An AE cipher takes as input a `key`, a `nonce` and a `plaintext` and outputs the pair `(ciphertext, MAC)`, if it is encrypting and does the inverse process, while also performing the Message Authentication Code (MAC) check if it is decrypting.

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

2.4 ECC

PubK cryptography is based on the use of one-way math functions, which are function where it is 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 the elliptic curve discrete logarithm problem, which 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 parameters, such as the 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 3 and described in more detail in Section 3.2.

TLS provides the following **security services**:

- **authentication** - both, **peer entity** and **data origin** (or **integrity**) authentication.
  - **peer entity authentication** - a peer has a guarantee that it is talking to certain entity, for example, `www.google.com`. This is achieved thought the use of AC, also known as Public Key Cryptography (PKC), (*e.g.* `RSA` and `DSA`) or **symmetric key cryptography**, using a Pre-Shared Key (PSK).
- **confidentiality** - the data transmitted between the communicating entities (the client and the server) is encrypted. Symmetric cryptography is used for data encryption (*e.g.*, `AES`).
- **integrity** (also called **data origin authentication**) - a peer can be sure that the data was not modified or forged, *i.e.*, there is a guarantee that the received data is coming from the expected entity. For example, a peer can be sure that the `index.html` file that was sent to when it connected to `www.google.com` did, in fact, come from `www.google.com` and it was not tampered with by an attacker (**data integrity**). This is achieved either through the use of a keyed MAC or an AEAD cipher.

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[9] 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.

The terms Secure Sockets Layer (SSL) and TLS are often used interchangeably, but one is a predecessor of another - SSL 3.0[10] served as the basis for TLS 1.0[10].

Section 3.1 will begin with a brief overview of the various sub-protocols that compose TLS. The TLS Record Layer will be described in sufficient detail for the TLS Handshake Protocol description that follows in Section 3.2. The way each record is processed when sending and receiving data is covered in Section 3.3. The symmetric keys involved in cryptographic operations that provide confidentiality and security are described in Section 3.4. Section 3.5 explains how those keys are generated in TLS 1.2. There are various methods that the client and the server can use to exchange keys, those will be covered in Section 3.6. The TLS Extension mechanism will be covered in Section 3.7. There are various differences

from TLS 1.2 to 1.3 and those that were not covered in the previous sections will be in Section 3.8. This section ends with an outline of the main differences from DTLS to TLS in Section 3.9.

3.1 TLS (Sub)Protocols

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

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

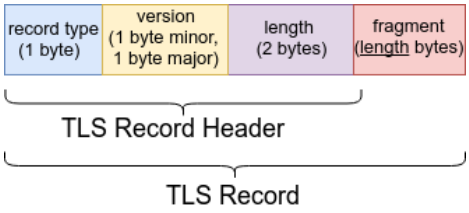


Fig. 1. TLS Record header

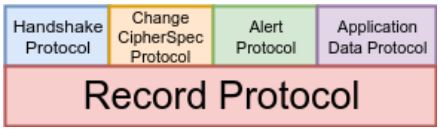


Fig. 2. TLS (Sub)protocols and Layers

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 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`[9] and **session tickets**, defined in `RFC 5077`[11].
- **TLS connection** - used to actually transmit the cryptographically protected data. For the data to be cryptographically protected, some parameters, such as the secret keys used to encrypt and authenticate the transmitted data need to be established; this is done when a **TLS session** is created, during the **TLS Handshake Protocol**.

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

As already mentioned, every TLS handshake message is encapsulated within a TLS record. The actual handshake message is contained within the `fragment` of a TLS record. The record type for a handshake message is `0x16`. The handshake message has the following structure: a 1-byte `msg_type` field (specifies

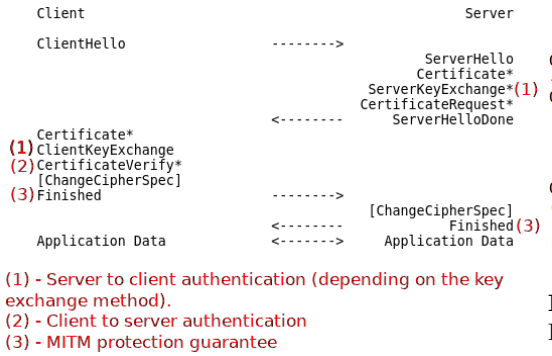


Fig. 3. TLS 1.2 message flow for a full handshake

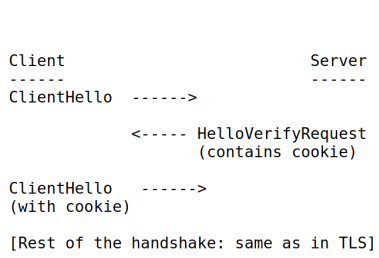


Fig. 4. DTLS handshake with HelloVerifyRequest containing the cookie

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.

The connection begins 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 has similar contents, except that instead of containing a list of supported features, it contains a **single item** containing the one that it chose. The server responds with a single `cipher_suite` and `compression_method` that it chose from the corresponding 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 and it is the one that is goint to be 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"[12]. 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 PubK 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 seldom 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 and 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 that 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 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 security parameters again. 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`, the server generates and sends a `session_id`, which will be associated with the newly negotiated session.

To resume a session, in its `ClientHello` the client includes a `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 will be able to impersonate the compromised party until the corresponding `session ID` is retired.

3.3 TLS Record Processing

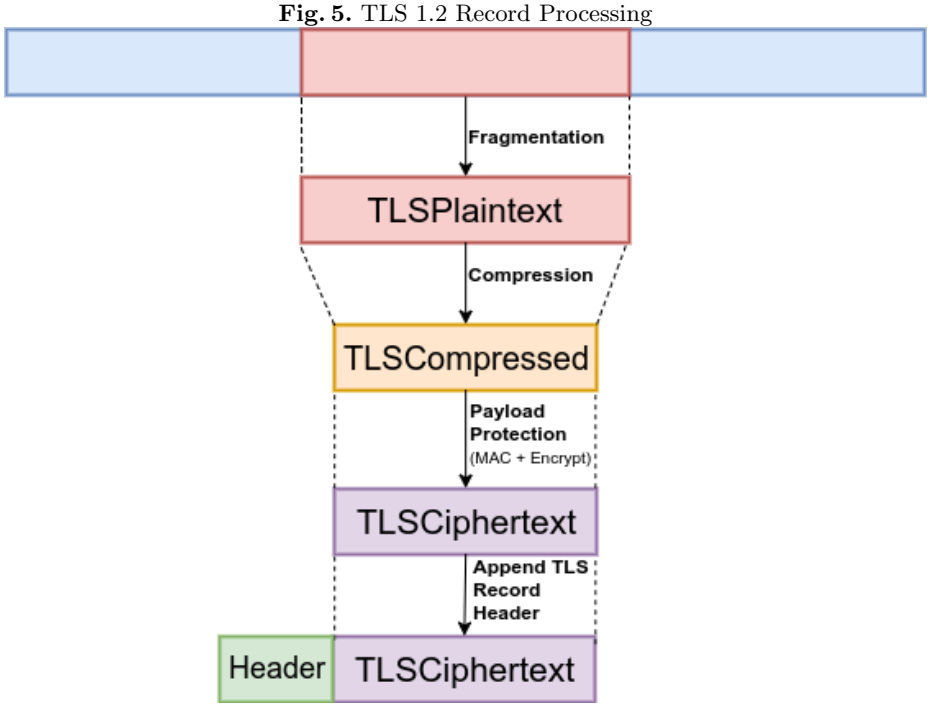
A TLS record must go through some processing before it can be sent over the network. This processing is done by the **TLS Record Protocol** and involves the following steps (`4` for TLS 1.2 and `3` for TLS 1.3):

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 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 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.

Fig. 5. TLS 1.2 Record Processing



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

3.4 TLS Keying Material

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

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

- `client write key` - used by the client to encrypt the data to be sent
- `client read_key` - used by the client to decrypt the incoming data from the server
- `server write key` - used by the server to encrypt the data to be sent
- `server read key` - used by the server to decrypt the incoming data from the server
- `client write IV` - used by the client for implicit nonce techniques with AEAD ciphers
- `server_write_IV` - used by the server for implicit nonce techniques with AEAD ciphers
- `client write MAC key` (TLS 1.2 only) - used by the client to authenticate the data to be sent
- `client write MAC key` (TLS 1.2 only) - used by the client to authenticate the data to be sent

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

3.5 TLS 1.2 Keying Material Generation

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

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

The TLS PRF is used to generate the keying material needed for a connection. 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, you can use it to 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 you need. 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 master and premaster secrets, nor the derivation of the shared keys described in Section **??** is 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`[9]), while others in separate Request For Comment (RFC)s. For example, the ECC based key exchange, specified in `RFC4492` [13]).

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 PubK (which it obtained from the server's `X.509` certificate) and sends it to the server. The server then decrypts it using the corresponding PrivK 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) - a DH key exchange is performed and an **ephemeral** key is generated. 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`) - 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. The public parameters are then digitally signed in some way, usually using the sender's private RSA (`DHE_RSA`) or Private Key (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 (without the leading 0's). Usually, only the server's authenticity is desired, but client's can also be achieved if it supplies the server with its certificate. Whenever the server is authenticated, it is secure against MITM attacks. Table 1 summarizes the security properties offered by each key exchange method.

Table 1. Key exchange methods and security properties

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

In TLS 1.3, static RSA and DH ciphersuites have been removed, meaning that all public key exchange mechanisms now provide PFS. Even though anonymous DH key exchange has been removed, you can still have unauthenticated connections by either using raw public keys[14] or not verifying the certificate chain and any of its contents.

The 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 are defined in RFC4292[15]. 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[16] 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 may 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[17]). 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 "break" 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[18] 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 you have 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 less 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 you can have

0-RTT data, 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, you have multiple encryption keys generated that are 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 [19] 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't 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[20]. There is a draft of DTLS 1.3 [21] that is currently under active development.

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

- two new fields are added to the record layer: an explicit 2 byte sequence number and a 6 byte epoch. The DTLS MAC is the same as in TLS, however, rather than using the implicit sequence number, the 8 byte value formed by concatenation of the epoch number and the sequence number is used.
- stream ciphers must not be used with DTLS.
- a stateless cookie exchange mechanism has been added to the handshake protocol in order to prevent Denial-of-Service (DoS) attacks. To accomplish this, a new handshake message, the HelloVerifyRequest has been added. After the ClientHello, the server responds with a HelloVerifyRequest containing a cookie, which is returned back to the server in another ClientHello that follows it. After this, the handshake proceeds as in TLS. This is depicted in Figure 4. Although optional for the server, this mechanism highly recommended, and the client must be prepared to respond to it. DTLS 1.3 follows

the same idea, but does it differently, namely, the HelloVerifyRequest message has been removed, and the cookie is conveyed to the client via an extension in a HelloRetryRequest message.
– the handshake message format has been extended to deal wit 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, since cryptography is a fundamental part of security and it is fundamental for it to be lightweight in order to run on devices with limited memory and processing capabilities. A lot of the work in IoT incorporates it in one way or another, so this section will begin with the description of the work done in this area.

Alex *et al*[22] 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 here due to 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 of 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 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 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 [23], where the authors were able to recover a secret key used to authenticate updates, via an SCA).

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

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

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

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

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

While there is a high demand lightweight PubK primitives, the required resources for them are much higher than for symmetric ones. As a paper by *Sony Corporation*[24] 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

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

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.

S3K[25] 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[26], 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`[27]. 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`[9].

The second approach consists in requesting an APK (the authors never defined what this acronym stands for, but my assumption is that they mean "Authorization Public Key") 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 PubK. 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` [27]. To achieve this, a new certificate structure is defined, alongside a new `certificate_type`. The new certificate structure is just the `RFC7250` [14] structure, with an additional 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 ChangeCipherSpec message which required a processing time of 17.79ms.

6LoWPAN[28] 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[29]. The compression relies on the shared context between the communicating peers.

[30] 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[31] 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 [30] 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.

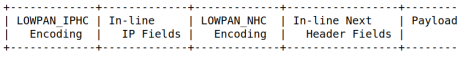


Fig. 6. IPv6 Next Header Compression

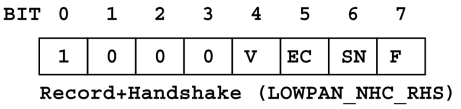


Fig. 7. LOWPAN_NHC_RHS structure

Each DTLS fragment is carried over as a UDP payload. In this case, the UDP payload carries a header-like payload (the DTLS record header). Figure 6 shows the way IPv6 next header compression is done. The authors use the same value for the LOWPAN_NHC Encoding field (defined in RFC 6282[29]) as in RFC7400 and define the format of the In-line Next Header Fields (also defined in [29]), which is the compressed DTLS content. The LOWPAN_IPHC Encoding and In-Line IP Fields fields are used in the IPv6 header compression and are not in the scope of the paper.

All of the cases follow the same basic idea, for this reason only one of them will be exemplified: the case where both, the Record and the Handshake headers are compressed. In this case LOWPAN_NHC Encoding will contain the LOWPAN_NHC_RHS structure (depicted in figure 7), which is the compressed form of the Record and Handshake headers. The parts that are not compressed will be contained in the Payload part. The first four bits represent the ID field and in this case they are fixed to 1000, that way, the decompressor knows what is being compressed (*i.e* how to interpret the structure that follows the ID bits).

If the F field of the LOWPAN_NHC_RHS structure contains the bit 0, it means that the handshake message is not fragmented, so the fragment_offset and fragment_length fields are elided from the Handshake header (common case when a handshake message is not bigger than the maximum header size), meaning that they are not going to be sent at all (*i.e.* they are not going to be present in the Payload part). If the F bit has the value 1, the fragment_offset and fragment_length fields are carried inline (*i.e.* they are present in the Payload part). The remaining two fields define similar behavior for other header fields (some of them assume that some default value is present, when a field is elided). The length field in the Record and Handshake headers are always elided, since they can be inferred from the lower layers.

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

There is also a proposal for TCP header compression for 6LoWPAN[32], 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[33], which is a lightweight security solution for Constrained Application Protocol (CoAP) that uses the same DTLS header compression technique as in [30] with the goal of implementing it as a security support for CoAP. CoAP[34] 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[35].

The authors evaluated their system in a simulated environment in *Contiki OS*[36], which is an open-source operating system for the IoT. They obtained significant gains in terms of packet size (similar numbers to the ones observer in [30]), 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.* [37] 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 guar-

antee. 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[38], 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` [39]), 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[34], 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 [40] 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 [41], which is based on precomputation. 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[42] 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. Some RFCs that are relevant will also be described.

This RFC 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[43] 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[27]. 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[14]. 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[44] 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[45], which enables TLS peers to exchange just the fingerprint (a shorter sequence of bytes used to identify a PubK) of the PubK. 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[45] 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[13]. For certificate revocation, neither the Online Certificate Status Protocol (OCSP)[46], nor the Certificate Revocation List (CRL)[44] 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[16] 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[47] 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[9] 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 [9] 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[11], 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[11].

The use of compression is not recommended for two reasons. First, RFC7525[48] recommends disabling (D)TLS level compression, due to attacks such as CRIME[49]. 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[50] 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 find ways to offer to the generation of quality

random numbers, the guidelines and requirements for which can be found in RFC4086[51].

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[52], nor the truncated MAC[47] extensions are applicable to this specification and must not be used.

The Server Name Indication (SNI) extension[47] 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[47] 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[53] defines an extension that binds the master secret to the log of the full handshake, thus preventing MITM attacks, such as the triple handshake[54]. 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[55], 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[56].

RFC 7925[42] 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[57] 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 frimware update mechanism for IoT devices[58] currently in "Internet-Draft" state.

5 Solution

5.1 Solution Architecture

Most of the work has been centered around DTLS, even though the majority of solutions can be applied to TLS as well. Herein we want to explore TLS optimization more. 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. None of the related work explored (D)TLS 1.3, mainly because the protocol is still in draft stages, however, major design changes are not expected at this point.

The majority of the work done in the area proposes a solution that is either tied to a specific protocol, such as CoAP, or requires an introduction of a third-party entity, such as the trust anchor in the case of S3K[25] or even both. This two main issues: first, if the developer wants to use (D)TLS without using a protocol for which an optimization has been done, he might see himself in some trouble, and 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 who are doing hobby 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 either TLS or DTLS and would work with both versions: (D)TLS 1.2 and the upcoming (D)TLS 1.3.

In order to achieve those goals, the TLS extension mechanism will be heavily relied upon. By using extensions, any deviations from the original protocol can be done, even if they completely change the it. This was confirmed by one of the TLS designers, after I asked this question on the official Internet Engineering Task Force (IETF) mailing list[59], since neither of the existing RFCs made it clear.

Ideas from TLS 1.3, such as `0-RTT data`, which can be done if peers already share a PSK would be interesting to explore for (D)TLS 1.2. For example, this can be very useful in the context of the temperature sensor described in the introduction, where the device sends data to a server every 30 seconds. With `0-RTT data`, it can communicate securely with minimal costs. The use of symmetric cryptography exclusively, has various advantages in the context of IoT. This means that there is value in exploring solutions that revolve around PSKs and that avoid public key cryptography altogether.

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 PubK cryptography, is acceptable, while others are powerful enough to take advantage of certificates and Public Key Infrastructure (PKI), at least up to a point. The solution proposed with this work would be adaptable to any of those scenarios, depending on the context and the security requirements, therefore being similar to a framework. For example,

some scenarios might require integrity, but not privacy. For those, it does not make sense to pay the energy and time overhead of encryption. Special care in the selection of cipher suites and key exchange methods needs to be taken, possibly even developing new ones.

(D)TLS 1.2 is significantly different from (D)TLS 1.3 and there has been no work like RFC 7924[45] done for (D)TLS 1.3. It would be interesting to explore this. (D)TLS 1.3 has many fundamental changes to the way the handshake is done, bringing many new features, whose ideas can be incorporated into the proposed solution and even backported to TLS 1.2.

Fig. 8. LTLS supported

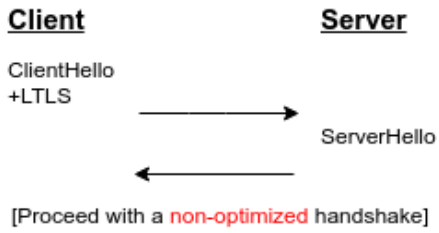


Fig. 9. LTLS not supported

To exemplify the idea of the solution, let us assume that a Lightweight TLS (LTLS) extension is defined. The optimizations that will be described in the second part of this work will be incorporated into it. Let us consider the case of a client that wants to use those optimizations. To achieve this, it will include the LTLS extension in its `ClientHello`. If the server supports the extension, an optimized connection can be established; this case is depicted in Figure 8. If the server does not support the extension, it will not send it back in its `ServerHello`. In that case it's possible to gracefully fall back to regular (D)TLS, as depicted in Figure 9.

In practice the extension will include additional data used to indicate various options, which has been omitted from the discussion for simplicity.

5.2 Evaluation

In order to evaluate how good the result of the work is, 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 used 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 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 few of them will be chosen. During the month of March 2018, the solution will be implemented in code, by modifying the *mbedTLS 2.6.0* library[7], the most important evaluation metrics will be chosen and the related profiling code set up. Testing scenarios will be designed. From early April 2018 until March 2018 the solution will be evaluated and a concentration on writing the dissertation's text will be done.

5.4 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 future work, we want to contribute to this area, by designing a solution that is suitable for the IoT devices, transparent to the programmer and has the provided security services adaptable to the specific context needs.

References

1. SMALT - The Smart Home Device That Elevates Your Dining Experience. Electronic Gadget, 2017. (Accessed on 01/05/2018).
2. Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. 2017.
3. Brian Krebs. Reaper: Calm before the iot security storm? krebs on security. `https://krebsonsecurity.com/2017/10/reaper-calm-before-the-iot-security-storm/`, 10 2017. (Accessed on 01/04/2018).
4. Elaine Barker. Recommendation for Key Management, Part 1: General. *NIST special publication*, 2016.

5. Bundesamt fur Sicherheit in der Informationstechnik (BSI). Kryptographische Verfahren: Empfehlungen und Schlssellngen, Version 2017-01. 2017.
6. C. Bormann, M. Ersue, and A. Keranen. Terminology for Constrained-Node Networks. RFC 7228, RFC Editor, May 2014. http://www.rfc-editor.org/rfc/rfc7228.txt.
7. mbed TLS (formely PolarSSL): SSL/TLS Library For Embedded Devices. (Accessed on 12/19/2017).
8. D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, RFC Editor, May 2008. http://www.rfc-editor.org/rfc/rfc5280.txt.
9. T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, RFC Editor, August 2008. http://www.rfc-editor.org/rfc/rfc5246.txt.
10. A. Freier, P. Karlton, and P. Kocher. The secure sockets layer (ssl) protocol version 3.0. RFC 6101, RFC Editor, August 2011. http://www.rfc-editor.org/rfc/rfc6101.txt.
11. J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077, RFC Editor, January 2008. http://www.rfc-editor.org/rfc/rfc5077.txt.
12. Stephen Checkoway, Jacob Maskiewicz, Christina Garman, Joshua Fried, Shaanan Cohney, Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, and Hovav Shacham. A systematic analysis of the juniper dual ec incident. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 468–479. ACM, 2016.
13. S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492, RFC Editor, May 2006. http://www.rfc-editor.org/rfc/rfc4492.txt.
14. P. Wouters, H. Tschofenig, J. Gilmore, S. Weiler, and T. Kivinen. Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7250, RFC Editor, June 2014.
15. B. Haberman. IP Forwarding Table MIB. RFC 4292, RFC Editor, April 2006.
16. S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport Layer Security (TLS) Extensions. RFC 4366, RFC Editor, April 2006. http://www.rfc-editor.org/rfc/rfc4366.txt.
17. Tim Dierks and Christopher Allen. The TLS Protocol Version 1.0. RFC 2246, RFC Editor, January 1999. http://www.rfc-editor.org/rfc/rfc2246.txt.
18. Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. Internet-Draft draft-ietf-tls-tls13-22, IETF Secretariat, November 2017. http://www.ietf.org/internet-drafts/draft-ietf-tls-tls13-22.txt.
19. H. Krawczyk and P. Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, RFC Editor, May 2010. http://www.rfc-editor.org/rfc/rfc5869.txt.
20. E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, RFC Editor, January 2012. http://www.rfc-editor.org/rfc/rfc6347.txt.
21. Eric Rescorla, Hannes Tschofenig, and Nagendra Modadugu. The Datagram Transport Layer Security (DTLS) Protocol Version 1.3. Internet-Draft draft-ietf-tls-dtls13-22, IETF Secretariat, November 2017. http://www.ietf.org/internet-drafts/draft-ietf-tls-dtls13-22.txt.

22. Alex Biryukov and Léo Paul Perrin. State of the art in lightweight symmetric cryptography. 2017.
23. Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin OFlynn. Iot goes nuclear: Creating a zigbee chain reaction. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 195–212. IEEE, 2017.
24. Masanobu Katagi and Shiho Moriai. Lightweight cryptography for the internet of things. *Sony Corporation*, pages 7–10, 2012.
25. Shahid Raza, Ludwig Seitz, Denis Sitenkov, and Göran Selander. S3k: Scalable security with symmetric keysdtls key establishment for the internet of things. *IEEE Transactions on Automation Science and Engineering*, 13(3):1270–1280, 2016.
26. C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5). RFC 4120, RFC Editor, July 2005. `http://www.rfc-editor.org/rfc/rfc4120.txt`.
27. P. Eronen and H. Tschofenig. Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). RFC 4279, RFC Editor, December 2005. `http://www.rfc-editor.org/rfc/rfc4279.txt`.
28. G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944, RFC Editor, September 2007. `http://www.rfc-editor.org/rfc/rfc4944.txt`.
29. J. Hui and P. Thubert. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. RFC 6282, RFC Editor, September 2011. `http://www.rfc-editor.org/rfc/rfc6282.txt`.
30. Shahid Raza, Daniele Trabalza, and Thiemo Voigt. 6lowpan compressed dtls for coap. In *Distributed Computing in Sensor Systems (DCOSS), 2012 IEEE 8th International Conference on*, pages 287–289. IEEE, 2012.
31. C. Bormann. 6LoWPAN-GHC: Generic Header Compression for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs). RFC 7400, RFC Editor, November 2014.
32. Ahmed Ayadi, David Ros, and Laurent Toutain. TCP header compression for 6LoWPAN. Internet-Draft draft-aayadi-6lowpan-tcphc-01, IETF Secretariat, October 2010. `http://www.ietf.org/internet-drafts/draft-aayadi-6lowpan-tcphc-01.txt`.
33. Shahid Raza, Hossein Shafagh, Kasun Hewage, René Hummen, and Thiemo Voigt. Lithe: Lightweight secure coap for the internet of things. *IEEE Sensors Journal*, 13(10):3711–3720, 2013.
34. C. Bormann and Z. Shelby. Block-Wise Transfers in the Constrained Application Protocol (CoAP). RFC 7959, RFC Editor, August 2016.
35. Carsten Bormann, Simon Lemay, Hannes Tschofenig, Klaus Hartke, Bill Silverajan, and Brian Raymor. CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets. Internet-Draft draft-ietf-core-coap-tcp-tls-11, IETF Secretariat, December 2017. `http://www.ietf.org/internet-drafts/draft-ietf-core-coap-tcp-tls-11.txt`.
36. Contiki: The open source operating system for the internet of things. `http://www.contiki-os.org/`. (Accessed on 01/03/2018); HTTPS link was not used because it is broken.
37. Angelo Capossele, Valerio Cervo, Gianluca De Cicco, and Chiara Petrioli. Security as a coap resource: an optimized dtls implementation for the iot. In *Communications (ICC), 2015 IEEE International Conference on*, pages 549–554. IEEE, 2015.

38. Sye Keoh, Sandeep Kumar, and Zach Shelby. Profiling of DTLS for CoAP-based IoT Applications. Internet-Draft draft-keoh-dtls-profile-iot-00, IETF Secretariat, June 2013. http://www.ietf.org/internet-drafts/draft-keoh-dtls-profile-iot-00.txt.
39. Z. Shelby, K. Hartke, and C. Bormann. The constrained application protocol (coap). RFC 7252, RFC Editor, June 2014. http://www.rfc-editor.org/rfc/rfc7252.txt.
40. Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and rsa on 8-bit cpus. In *CHES*, volume 4, pages 119–132. Springer, 2004.
41. Giuseppe Ateniese, Giuseppe Bianchi, Angelo Capossele, and Chiara Petrioli. Low-cost standard signatures in wireless sensor networks: a case for reviving pre-computation techniques? In *Proceedings of NDSS 2013*, 2013.
42. H. Tschofenig and T. Fossati. Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things. RFC 7925, RFC Editor, July 2016.
43. D. Simon, B. Aboba, and R. Hurst. The EAP-TLS Authentication Protocol. RFC 5216, RFC Editor, March 2008. http://www.rfc-editor.org/rfc/rfc5216.txt.
44. D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, RFC Editor, May 2008. http://www.rfc-editor.org/rfc/rfc5280.txt.
45. S. Santesson and H. Tschofenig. Transport Layer Security (TLS) Cached Information Extension. RFC 7924, RFC Editor, July 2016.
46. S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 6960, RFC Editor, June 2013. http://www.rfc-editor.org/rfc/rfc6960.txt.
47. D. Eastlake. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066, RFC Editor, January 2011. http://www.rfc-editor.org/rfc/rfc6066.txt.
48. Y. Sheffer, R. Holz, and P. Saint-Andre. Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). BCP 195, RFC Editor, May 2015. http://www.rfc-editor.org/rfc/rfc7525.txt.
49. Amichai Shulman Tal Be'ery. A perfect CRIME? Only TIME will tell. https://media.blackhat.com/eu-13/briefings/Beery/bh-eu-13-a-perfect-crime-beery-wp.pdf, 2013. (Accessed on 01/04/2018).
50. R. Seggelmann, M. Tuexen, and M. Williams. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. RFC 6520, RFC Editor, February 2012. http://www.rfc-editor.org/rfc/rfc6520.txt.
51. D. Eastlake, J. Schiller, and S. Crocker. Randomness Requirements for Security. BCP 106, RFC Editor, June 2005. http://www.rfc-editor.org/rfc/rfc4086.txt.
52. P. Gutmann. Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7366, RFC Editor, September 2014. http://www.rfc-editor.org/rfc/rfc7366.txt.
53. K. Bhargavan, A. Delignat-Lavaud, A. Pironti, A. Langley, and M. Ray. Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension. RFC 7627, RFC Editor, September 2015.
54. Karthikeyan Bhargavan, Antoine Delignat Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre Yves Strub. Triple handshakes and cookie cutters: Break-

ing and fixing authentication over tls. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 98–113. IEEE, 2014.
55. E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746, RFC Editor, February 2010. `http://www.rfc-editor.org/rfc/rfc5746.txt`.
56. A. Popov. Prohibiting RC4 Cipher Suites. RFC 7465, RFC Editor, February 2015. `http://www.rfc-editor.org/rfc/rfc7465.txt`.
57. LightweightM2M V1.0 Overview, 2017. (Accessed on 12/30/2017).
58. Brendan Moran, Milosch Meriac, and Hannes Tschofenig. A Firmware Update Architecture for Internet of Things Devices. Internet-Draft draft-moran-suit-architecture-00, IETF Secretariat, October 2017. `http://www.ietf.org/internet-drafts/draft-moran-suit-architecture-00.txt`.
59. Re: [TLS] TLS Extensions: Omitting Handshake Messages. `https://www.ietf.org/mail-archive/web/tls/current/msg24932.html`, 2017. (Accessed on 01/04/2018).
60. M. Badra and I. Hajjeh. ECDHE_PSK Cipher Suites for Transport Layer Security (TLS). RFC 5489, RFC Editor, March 2009.
61. U. Blumenthal and P. Goel. Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport Layer Security (TLS). RFC 4785, RFC Editor, January 2007.
62. F. Cusack and M. Forssen. Generic Message Exchange Authentication for the Secure Shell Protocol (SSH). RFC 4256, RFC Editor, January 2006.
63. Leo Laporte Steve Gibson. Securitynow podcast episodes.

Glossary

AC Asymmetrical Cryptography. 3, 5
AE Authenticated Encryption. 4
AEAD Authenticated Encryption With Associated Data. 4–6, 8, 9, 11, 25

CA Certification Authority. 3, 24
CoAP Constrained Application Protocol. 21–23, 26
CRL Certificate Revocation List. 23

DDoS Distributed Denial-Of-Service. 1, 2
DH Diffie-Hellman. 10, 11, 13, 23, 25
DoS Denial-of-Service. 16
DTLS Datagram TLS. 1, 2, 16, 20–24, 26

EC Elliptic Curve. 22
ECC Elliptic Curve Cryptography. 3, 4, 10, 11, 18, 21, 22, 25
ECDH Elliptic Curve Diffie-Hellman. 11, 22
ECDHE Elliptic Curve Diffie-Hellman Ephemeral. 11
ECDSA Elliptic Curve Digital Signature Algorithm. 11, 22

HKDF HMAC-based Extract-and-Expand Key Derivation Function. 6, 10
HTML Hypertext Markup Language. 7
HTTPS Hypertext Transfer Protocol Secure. 7

IETF Internet Engineering Task Force. 5, 26
IoT Internet Of Things. 1–3, 15–17, 19, 21–26, 28
IV Initialization Vector. 10

MAC Message Authentication Code. 4, 5, 8–10, 14, 16, 19, 25
MITM Man In The Middle. 10, 11, 14, 25

NSA US National Security Agency. 28

OCSP Online Certificate Status Protocol. 23

PFS Perfect Forward Secrecy. 10, 11, 23

PKC Public Key Cryptography. 5
PKI Public Key Infrastructure. 26
PRF Pseudo-Random Function. 6, 9, 10, 13
PrivK Private Key. 3, 10
PSK Pre-Shared Key. 5, 16, 19, 23, 24, 26
PubK Public Key. 3, 4, 10, 13, 18, 19, 23, 26

RFC Request For Comment. 10, 23, 26
RPK Raw Public Key. 19, 23, 26
RSA Rivest-Shamir-Adleman. 3, 10, 11, 13

SC Symmetrical Cryptography. 3, 8
SCA Side-Channel Attack. 18
SNI Server Name Indication. 25
SSL Secure Sockets Layer. 5

TLS Transport Layer Security. I, 1–16, 19, 21, 23–28