Table of Contents

TLS For IoT

Transport Layer Security Protocol For Internet Of Things

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

Instituto Superior Tcnico
Supervisors: Ricardo Chaves, Aleksandar Ilic

Abstract. This paper explores the idea of developing a lightweight Transport Layer Security (TLS) protocol for the Internet Of Things (IoT). I begin by explaining why there is a need for security in the IoT ecosystem and why TLS, as it is, cannot be used in most cases, due to the constrained nature of many IoT devices. I then describe the state of the art in the area, most of which is done for Datagram TLS (DTLS), but can also be applied to TLS, since DTLS is just an adaptation of TLS for unreliable transport protocols, such as UDP. I proceed with a summary of TLS 1.2, outline its main differences from TLS 1.3 and give an overview of the how DTLS differs from TLS. Finally, I present an architecture of the solution that I will be developing, which relies heavily on the use of the TLS extension mechanism.

Keywords: TLS, 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 can be assigned an IP address and provided with the ability to transfer data over a network. Even something such a salt shaker[16] can now be part of the global network.

The IoT technology can be used solve problems and make our lives easier, unfortunately however, the 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[14] when it overhelmed 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[10] however, could be even more devastating, if it's ever put to malicious use and 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. TLS runs 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 mostly used in such devices. The work that I will be doing, however, can be applied to either one of them, so even though I will be talking mostly about TLS, almost everything also applies to DTLS as well, since it's 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's 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 lots 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 with minimal overhead, to conserve RAM and battery life. If that sensor is going to use TLS 1.2, it will need two extra roundtrips before it can send any data, which can be extra hundreds of milliseconds. Besides that, it will need to perform heavy mathematical operations involved in cryptogrpahy, wasting even more battery and taking even more time. There is a clear need for a more lightweight (D)TLS for the IoT.

With my work, I want 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 centered among the (D)TLS versions 1.2 and 1.3. The idea is to make it customizable, depending on the security requirements of the context it's being used in, in that sense, it would be similar to a framework.

In the process of my work on this master thesis, I've 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.

2 Theoretical Background

2.1 Symmetric vs Asymmetric Cryptography (Background Section)

Asymmetrical Cryptography (AC) more expensive than Symmetrical Cryptography (SC) in terms of performance. This is mainly due to two facts: larger key sizes are required for AC system to achieved the same level of security as in a SC system and CPUs are slower at performing the underlying mathematical operations involved in AC, namely exponentiation requires O(log e) multiplications for an exponent e. For example, the 2016 NIST report [11] suggests that an AC would need to use a secret key with size of 15360 bits to have equivalent security to a 256-bit secret key for a SC algorithm. This situation is ameliorated

by Elliptic Curve Cryptography (ECC), which requires keys of 512 bits, it is still slower than SC, though. The 2017 BSI report [4] (from the German federal office for information security) suggests similar numbers.

Another argument for avoiding as much as possible of the AC functionality is that it requires extra storage space to be used and this might be a problem for some IoT devices, like class 1 devices according to the terminology of constrained-code networks [26] which have about 10KB of RAM and 100KB of persistent memory. I measured the resulitng size of the complied mbedTLS 2.6.0 library [17] when compiled with and without the RSA module (located in the rsa.c file), from which I concluded that using that module adds an extra of 32KB.

2.2 Public Certificates

TODO: todo

2.3 AEAD Ciphers

TODO: Already have this written, need to shorten

2.4 Terminology

Before describing the related work, I will briefly introduce some terminology that will be used throughout the document:

– Perfect Forward Secrecy (PFS) - property that preserves the confidentiality of past interactions even if the long-term secret is compromised.

3 Related Work

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

Alex *et al*[18] 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. I won't be including the list here, due to lack of space. The authors also proposed to split the field into two areas: ultra-lightweight and IoT cryptography.

The authors systematized the knowledge in the area of lightweight cryptography in order to define "lightweightness" more precisely. They observed that

the design of lightweight cryptography algorithms vary greatly, the only unifying thread between them being the low computing power of the devices they're designed to run on.

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

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

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

Even though though algorithms can be optimized in implementation: whether it's a software or a hardware now, dedicated lightweight algorithms are still needed. This comes down mainly to two factors: there are limitations to the the extent of the optimizations that you can make and the hardware-accelerated encryption is frequently vulnerable to various Side-Channel Attack (SCA)s (such as the attack done on the Phillips light bulbs [47], where the authors were able to recover a secret key used to authenticate updates, via an SCA).

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

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

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

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

Considering the two definitions above, this work focuses on **IoT Crypto** devices. A summary of differences between the both categories is summarized in table 1.

Table 1. A summary of the differences between ultra-lightweight and IoT crypto

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

TODO: WRITE ABOUT LIGHTWEIGHT PUBLIC CRYPTO HERE

Lightweight cryptography is an important part of my work and there are papers detailing various algorithms. While an important topic for my solution, I do consider that I've done sufficient coverage on it, by specifically choosing recent works that provide an overview of the different areas, rather than focusing on specific implementations, since the length of this section is limited. The remaining text in this seciton will be used to describe work done on the (D)TLS protocol in the context of IoT.

For the reasons specifed above, the main key exchange mechanism used in IoT are Pre-Shared Key (PSK)s. SK3[13] proposes a key management architecture for resource-constrained devices, which allows devices that have no previous, direct security relation to use TLS or DTLS using one of two approaches: shared symmetric keys or raw public keys. The resource-constrained device is a server that offers one or more resources, such as temperature readings. The idea in both approaches is to introduce a third-party `trust anchor (TA)` that both, the client and the server use to establish trust relationships between them.

The first approach is similar to Kerberos[**?**], without requiring any changes to the original protocol. A client can request a PSK `Kc` from the `TA`, which will generate it and send it back to the client via a secure channel, alongside a `psk_identity` which has the same meaning and is used in the same way as defined in `RFC PSK`[**?**]. When connecting to the server, the client will then send the `psk_identity` that it received in the the (D)TLS handshake and the server will derive the `Kc`, using the `P_hash()` function defined in [31].

The second approach consists in the requesting an APK (the authors never defined what this acronym stands for, but I assume they mean "Authorization Public Key") from the TA. In his request, the client includes his Raw Public Key (RPK), which is used for authorization. The TA creates an authorizaton certificate, protects it with a MAC and sends it to the client alongside the server's Public Key (PubK). The client then sends this APK (instead of the RPK) when connecting to the server, which verifies the APK (to authorize the client) and proceeds with the handshake in the RPK mode, as defined in [**?**]. To achieve this, a new certificate structure is defined, alongside a new `certificate_type`. The new certificate strucure is just the RFC7250 [55] structure, with an additonal MAC.

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

6LoWPAN[40] is a protocol that allows devices with limited processing ability and power to transmit information wirelessly using the `IPv6` protocol. The protocol defines IP Header Compression(IPHC) for the IP header and Next Header Compression (NHC) for the IP extension headers and the UDP header in [38]. The compression relies on the shared context between the communicating peers.

[1] uses this same idea, but with the goal of compressing DTLS headers. 6LoWPAN does not provide ways to compress the UDP payload and layers above, there is however, a proposed standard[25] for generic header compression for 6LoWPANs that can be used to compress the UDP payload. The authors propose a way to compress DTLS headers and messages using this mechanism.

The paper [1] defines how the 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.

All of the cases follow the same basic idea, for this reason I'll only exemplify one of them. Each DTLS fragment is carried over in the UDP payload. In this

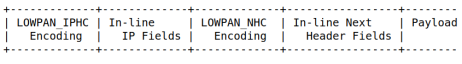

Fig. 1. IPv6 Next Header Compression

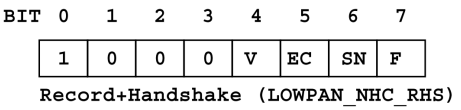

Fig. 2. LOWPAN_NHC_RHS structure

case, the UDP payload carries a header-like payload (the DTLS record header). Figure 1 shows the way IPv6 next header compression is done. The authors use the same value for the `LOWPAN_NHC Encoding` field (defined in [38]) as in `RFC7400` and define the format of the `In-line Next Header Fields` (defined in [38]), which is the compressed DTLS content. The `LOWPAN_IPHC Encoding` and `In-Line IP Fields` fields are used in the IPv6 header compression and are not in the scope of the paper.

I will exemplify the case where both, the record and the Handshake headers are compressed. In this case `LOWPAN_NHC Encoding` will contain the `LOWPAN_NHC_RHS` structure (depicted in figure 2), which is the compressed form of the Record and Handshake headers. The parts that are not compressed will be contained in the `Payload` part. The first four bits represent the ID field and in this case they're fixed to `1000`, so that the decompressor knows what is being compressed (*i.e* how to interpret the structure that follows the ID bits). If the `F` field of the `LOWPAN_NHC_RHS` structure contains the bit `0`, it means that the handshake message is not fragmented, so the `fragment_offset` and `fragment_length` fields are elided from the Handshake header (common case when a handshake message is not bigger than the maximum header size), meaning that they're not going to be sent at all (*i.e.* they're not going to be present in the `Payload` part). If the `F` bit has the value `1`, the `fragment_offset` and `fragment_length` fields are carried inline (*i.e.* they're present in the `Payload` part). The remaining two fields define similar behavior for other header fields (some of them assume that some default value is present, when a field is elided). The `length` field in the Record and Handshake headers are always elided, since they can be inferred from the lower layers.

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

There is also a proposal for TCP header compression for 6LoWPAN[20], 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[6], which is a lightweight security solution for Constrained Application Protocol (CoAP) that uses the same DTLS header compression technique as in [1] with the goal of implementing it as a security support for CoAP.CoAP[27] is a specialized RESTful Internet Application Protocol for constrained devices. It's designed to easily translate to HTTP, in order to simplify its integration with the web, while

also meeting requirements such as multicast support and low overhead. CoAP is like "HTTP for constrained devices". CoAP can run on most devices that support UDP or a UDP-like protocol. CoAP mandates the use of DTLS as the underlying security protocol for authenticated and confidential communication. There is also a specification CoAP running on top of TCP, which uses TLS as its underlying security protocol currently being proposed [**?**], with active work being done in this area.

The authors evaluated their system in a simulated environment in Contiki OS and they obtained significant gains in terms of packet size (similiar numbers to the ones obsever in [1]), energy consumption (in average `15\%` less energy is used to transmitting and receive compressed packets), processing time (the compression and decompression time of DTLS headers is almost negligible) and network-wide response times(up to `50\%` smaller RTT). The gains in the mentioned measures are the largest when the compression avoids fragmentation (in the paper, for payload size of `48 bytes`).

Angelo *et al* [15] proposed to integrate the DTLS protocol inside the CoAP, while also exploiting ECC optimizations and minimizing ROM occupancy. They've implemented their solution on an off-the-shelf mote platform and evaluated its performance. DTLS was designed to protect web application communication, as a result, it has a big overhead in IoT scenarios. Besides that, it runs over UDP, so additional mechanisms are needed to provide the reliability and ordering guarantee. With this in mind, the authors wanted to design a version of DTLS that's both: minimizes the code size and the number of exchanged messages, resulting in an optimized Handshake protocol.

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

To achieve this they proposed the use of a RESTful DTLS connection as a CoAP resource, which is created when a new secure session is requested. The authors exploit the the CoAPs capability to provide connection-oriented communication offered by its message layer. In particular, each `Confirmable` CoAP message requires an `Acknowledgement` message[12], 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[27], which was developed to support transmission of large payloads. This approach has two advantages: first, the code in the DTLS layer responsible for this function can be removed, thus reducing ROM occupancy and second, the fragmentation/reassembly process burdens the lower Layers with state that is better managed in the application layer.

The authors also optimized the implementation of basic operations on which many security protocols, such as Elliptic Curve Diffie-Hellman (ECDH) and Elliptic Curve Digital Signature Algorithm (ECDSA) rely upon. The first optimization had to do with modular arithmetic on large integers. A set of optimized assembly routines based on [2] allow the improved use of registers, reducing the number of memory operations needed to perform operations such as multiplications and square roots on devices with `8-bit` registers.

Scalar multiplication is often the most expensive operation in Elliptic Curve (EC) based cryptography, therefore optimizing it is of high interest. The authors used a technique called *IBPV* described in [7], which is based on precumputation of a set of Discrete Log pairs. I will refrain from going into the mathematical details, since they're not relevant for this description. The *IBPV* technique was used to improve the performance of the ECDSA signature and was also extended to the ECDH protocol. In order to reduce the time it takes to do the ECDSA signature verification, the *Shamir trick* was used, which allows to perform the sum of two scalar multiplications (frequent operation in EC cryptography) faster than performing two independent scalar multiplications.

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

`RFC 7925`[54] describes a TLS and DTLS 1.2 profile for IoT devices that offers communication security services for IoT applications and is reasonably implementable on many constrained devices. 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, so I'll summarize the most important parts. I will also provide a brief description of some RFCs that I consider to be relevant for my work.

The 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[53] can reuse most parts of the code for (D)TLS at the application layer.

For the credential types the profile considers 3 cases:

– PSK - authentication based on PSKs is described in `RFC 4249`[35]. When using PSKs, the client indicates which key it uses 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 ciphersuite to be used with PSKs, just like CoAP does. If a

PFS ciphersuite is used, ephemeral Diffie-Hellman (DH) keys should not be reused over multiple protocol exchanges.

- RPK - the use of RPKs in (D)TLS is described in RFC 7250[**?**]. With RPKs, only a subset of the information 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[29] 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 the RPK use, two new extensions have been defined: one for the client to list the certificate types that it supports (sorted by order of preference) and one for the indicate which one it chose. To further reduce the size of the implementation, the RFC 7250 recommends the use of the TLS Cached Information extension[50], which enables the TLS peers to exchange just the fingerprint (a shorter sequence of bytes used to identify a PubK) of the PubK. The only mandatory-to-implement ciphersuite to be used with RPKs is TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8, just like with CoAP. Note that the ciphersuite makes the use of Authenticated Encryption With Associated Data (AEAD).
- certificate - conventional certificates can also be used. The support for the Cached Information extension[50] and the TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8 ciphersuite is required. The profile restricts the use of named curves to the ones defined in [22]. For certificate revocation, neither the Online Certificate Status Protocol (OCSP)[49], nor the Certificate Revocation List (CRL)[29] 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[23] 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[33] 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 Certification Authority (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 recommended.

The signature algorithms extension[31] allows the client to indicate to the server which signature/hash pairs it supports to be used in digital signatures. The client must send this extension to indicate the use of SHA-256, otherwise the defaults defined in [31] are used. This extension is not applicable when the PSK-based ciphersuites are used.

The profile mandates that constrained clients must implement the session resumption to improve the performance of the handshake since this will lead to

less exchanged messages, lower computational overhead, since only symmetric cyrptography is used, and it requires less bandwidth. In case the server is constrained, but the client is not, the client must implement the Session Resumption Without Server-Side State mechanism[48], 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[48].

The use of compression is not recommended for two reasons. First, RFC7525[52] recommends disabling (D)TLS level compression, due to attacks such as CRIME[**?**]. RFC7525 provides recommendations for improving the security of deployed services that use TLS and DTLS and was published as a response to the various attacks on (D)TLS that have emerged over the years. Second, for IoT applications, the (D)TLS compression is not needed, since application-layer protocols are hightly optimized and compression at the (D)TLS layer increases the implementation's size and complexity.

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

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 generate quality random numbers, the guidelines and requirements for which can be found in RFC4086[34], since they play an essential role in the overall security of the protocol.

Implementations compliant with this profile must use AEAD ciphers, this means that encryption and Message Authentication Code (MAC) computation are no longer independent steps, which means that neither encrypt-then-MAC[36], nor the truncated MAC[33] extensions are applicable to this specification and must not be used.

The Server Name Indication (SNI) extension[33] defines a mechanism for a client to tell a (D)TLS server the name of the server it wants to connect to. 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[33] 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's able to buffer, allowing the client implementations to lower their RAM requirements, since it doesn't not need to accept packets of large size, such as the 16K packets required by plain (D)TLS. For that reason, client implementations must support this extension.

The Session Hash Extended Master Secret Extension [**?**] defines an extension that binds the master secret to the log of the full handshake, thus preventing

Man In The Middle (MITM) attacks, such as the tripple handshake[19]. Even though the ciphersuites recommended by the profile are not vulnerable to this attack, the implementation of this extension is advised. In order to prevent the renegotiation attack[44], 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 ciphersuites, may be shorter, dependent on the sensitivity of transmitted data.

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

The `RFC7925`[54] points out the importance of 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`[8] 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. Namely, there is a document specifying an architecture for a frimware update mechanism for IoT devices[**?**] currently in "Internet-Draft" state.

4 Background

5 The TLS Protocol

TLS stands for Transport Layer Security, it's a **client-server** protocol that runs on top a **connection-oriented and reliable transport protocol**, such as **TCP**. Its main goal is to provide **privacy** and **integrity** between the two communicating peers. Privacy implies that a third party will not be able to read the data, while integrity means that a third party will not be able to alter the data.

In the TCP/IP Protocol Stack, TLS is placed between the **Transport** and **Application** layers. It's designed to make the application developer's life easier: all the developer has to do is create a "secure" connection, instead of a "normal" one.

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

1. **Negotiate security parameters** - the communicating peers agree on a set of security parameters to be used in a TLS connection, such as the algorithm used for bulk data encrytion, as well as the secret keys.

2. **Authenticate one to another** - usually only the server authenticates to the client.
3. **Communicate securely** - use the negotiated security parameters to encrypt and authenticate the data, communicating securely one with another.

SSL vs TLS: What's The Difference? You will find the names Secure Sockets Layer (SSL) and TLS used interchangeably in the literature, so I think it's important to distinguish both. TLS is an evolution of the SSL protocol. The protocol changed its name from SSL to TLS when it was standardized by the Internet Engineering Task Force (IETF).SSL was a proprietary protocol owned by Netscape Communications, and The IETF decided that it was a good idea to standarize it, which resulted in `RFC 2246` [32], specifying TLS 1.0, which was nothing more than a new version SSL 3.0, very few changes were made. In this document, I'll be concentrating on TLS 1.2 and TLS 1.3 protocols. The first one is the most recently standardized version of TLS and the latter is currently and in-draft version with many improvements and optimizations relevant for the topic of this dissertation. Despite the protocol name not suggesting it TLS 1.3 is very different from TLS 1.2, in fact, it should've probably been called TLS 2.0 instead. For this reason, I will first describe what is common to both protocols and then go into the relevant details about each one.

5.1 Security Services

TLS provides the following 3 security services:

- **authentication** - both, **peer entity** and **data origin** (or **integrity**) authentication.
  - **peer entity authentication** - we can be sure that were talking to certain entity, for example, `www.google.com`. This is achieved thought the use of **asymmetrical** or Public Key Cryptography (PKC) (for example, `RSA` and `DSA`) or **symmetric key cryptography**, using a PSK.
- **confidentiality** - the data transmitted between the communicating entities (the client and the server) is encrypted. Symmetric cryptography is used of data encryption (for exmaple, `AES`).
- **integrity** (also called **data origin authentication**) - we can be sure that the data was not modified or forged, *i.e.*, be sure that the data that were receiving is coming from the expected entity (for example, we can be sure that the `index.html` file sent to us when we connected to `www.google.com` in fact came from `www.google.com` and that it was not modified (i.e tampered with) en route by an attacker (**data integrity**). This is achieved through the use of a keyed 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 user denying having sent a message, not **non-repudiation with proof of delivery**, which addresses the user denying the receipt of a message. This is due to the fact, that

instead of using **digital signatures**, either a keyed MAC or an AEAD cipher is used, both of which require a **shared secret** to be used.

You are not required to use all of the 3 security services in every situation. You can think of TLS as a framework that allows you to select which security services you want to use for a communication session. As an example, you might ignore certificate validation, which means you're ignoring the **authentication** guarantee. There are some differences regarding this claim between TLS 1.2 and TLS 1.3, for example, while in the first you have a `null` cipher (no authentication, no confidentiality, no integrity), in the latter this is not true, since it deprecated all non-AEAD ciphers in favor of AEAD ones.

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

5.2 TLS (Sub)Protocols

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

- **TLS Record Protocol** - the lowest layer in TLS. It's the layer that runs directly on top of **TCP/IP** and it serves as an **encapsulation for the remaining sub-protocols** (`4` in case of TLS 1.2 and `3` in case of TLS 1.3). To the **Record Protocol**, the remaining sub-protocols are what `TCP/IP` is to `HTTP`. A TLS Record is comprised of 4 fields, with the first 3 comprising the TLS Record header: a 1-byte record `type`, specifying the type of record that's encapsulated (ex: value `0x16` for the handshake protocol), a 2-byte `TLS version` field, a 2-byte `length` field (which means that a maximum TLS Record size is of `16384` bytes), specifying the length of the data in the record, excluding the header itself and a `fragment` field whose size in bytes is specified by the `length` field, which contains data that's transparent to the Record layer and should be dealt by a higher-level protocol, specified by the `type` field. This is illustrated in figure **??**.
- **TLS Handshake Protocol** - the core protocol of TLS. Allows the communicating peers to **authenticate** one to another and negotiate a **cipher suite** (**cipher suite** and key exchange algorithm in case of TLS 1.3) which will be used to provide the security services. For TLS 1.2, **compression** method is also negotiated here.

- **TLS Alert Protocol** - allows the communicating peers to signal potential problems.
- **TLS Application Data Protocol** - used to transmit data securely.
- **TLS Change Cipher Spec Protocol** (removed in TLS 1.3) - used to activate the initial **cipher spec** or change it during the connection.

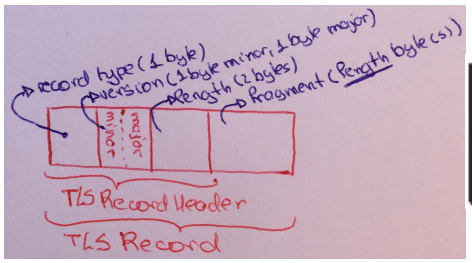

Fig. 3. TLS Record header

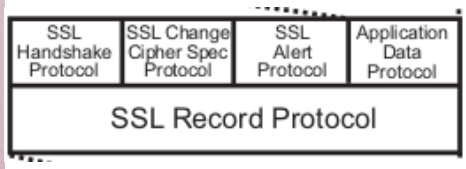

Fig. 4. TLS (Sub)protocols and Layers

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

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

- **TLS sesion** - assosciation between two communicationg peers that's created by the **TLS Handshake Protocol**, wich defines a set of negotiated paramters (cyrptographic and others, depending on the TLS version, such as the compression algorithm) that are used by the **TLS connections associated with that session**. A single **TLS session** can be shared among multiple **TLS connections** and its main purpuse is to avoid the expensive negotiation of new parameters for each **TLS connection**. For example, let's say you download an Hypertext Markup Language (HTML) page over Hypertext Transfer Protocol Secure (HTTPS) and that page referrences some images from that same server, also using HTTPS, instead of your web browser negotiating a new TLS session again, it can re-use the the one you established to download the HTML page in the first place, saving time and computational resources. Session resumption can be done using various approaches, such as **session identifiers**, described throughout `Section 7.4` of `RFC 5246` [31], **session tickets**, defined in `RFC 5077` [48]. TODO: Re-write example better.
- **TLS connection** - used to actually transmit the cryptographically protected data. For the data to be cryptographically protected, some parameters, such as the `secret keys` used to encrypt and authenticate the transmitted data need to be established; this is done when a **TLS session** is created, during the **TLS Handshake Protocol**.

TLS Record Processing A TLS record must go through some processing before it can tbe sent over the netwrok. This processing involves the following steps (4 for TLS 1.2 and 3 for TLS 1.3):

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

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

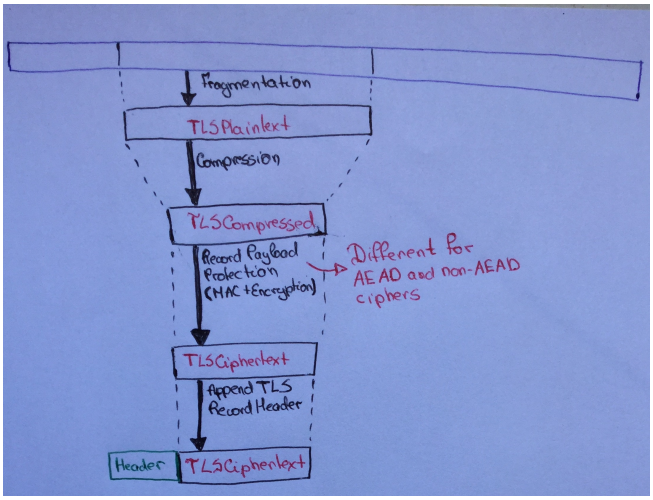

Fig. 5. TLS Record Processing

5.3 TLS Keying Material

Secret keys are at the base of most cryptographic operations. In order for both communicating peers to be able to encrypt and decrypt data using symmetric cyrpto aglorithms, they need to **share** the same key somehow. In TLS, both, the client and a server derive the **same set of keys** independetely, through the exchanged messages in the TLS Handshake Protocol.

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

by the client to encrypt the data to be sent), `client_read_key`(used by the client to decrypt the incoming data from the server), `server_write_key`(used by the server to encrypt the data to be sent) and `server_read_key`(used by the server to decrypt the incoming data from the client). Note, that the following relationships must hold: `client_write_key == server_write_key` and `client_read_key == server_write_key`.

Besides the secret keys mentioned previously, in TLS 1.2 you might also have other ones, depending on the cipher suite in use, namely the `client/server_wirite_IV` that is only generated for implicit nonce techniques used with AEAD ciphers and the `client/server_write_MAC_key`, which is the input secret to the MAC function; this key is not present when AEAD ciphers are in use.

5.4 TLS 1.2 Keying Material Generation

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

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

In TLS 1.2, the TLS PRF is used to generate the keying material needed for a connection, which 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, meaning that you can use to to generate anywhere from 1 to an infinite number of bits of output. The `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 you will actually obtain final keys the `secret` input is the `master secret`.

The cryptographic hash function used in `P_hash(secret, label, seed)` is a hash function 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.

5.5 TLS 1.3 Keying Material

TLS 1.3's keying material generation is a little more complex, 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, with Handshake messages besides the `Finished` being encrypted, as well as features such as **early client data**, also known as **0-RTT data**, where data comes encrypted in the first flight. As a result of those properties, you have multiple

encryption keys generated and used to encrypt different data throughout the handshake.

The way the keying material is derived is also different, since the PRF construction described above has been replaced. The new design allows easier analysis by cryptographers due to the improved key separation properties. In TLS 1.3, key derivation uses the HKDF function defined in RFC 5869[?] and its two components, the HKDF-Extract and HKDF-Expand.

6 TLS 1.2 Key Exchange Methods

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

You have quite a few choices when it comes to key exchange methods. Some of them are defined in the base spec (RFC5246 [31]), while others in separate RFCs (such as the ECC based key exchange, specified in RFC4492 [22]).

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

- static RSA (RSA) [removed in TLS 1.3] - the client generates the premaster secret (PMS), encrypts it with the server's PubK (which it obtained from the server's X.509certificate), sending it to the server, which decrypts it using the corresponding Private Key (PrivK). This key exchange method offers authenticity, but does not offer PFS.
- anonymous DH (DH_anon) [removed in TLS 1.3] - a DH key exchange is performed and an **ephemeral** key is generated, but the exchanged DH parameters are **not authenticated**, making the resulting key exchange vulnerable to MITM attacks. TLS 1.2 spec states that cipher suites using DH_anon **must not** be used, unless the application layer explicitly requests so. This key exchange offers PFS, but no authenticity.
- fixed/static DH (DH) [removed in TLS 1.3] - the server's/client's public DH parameter is embedded in its certificate. This key exchange method offers authenticity, but does not offer PFS.
- epehemeral DH (DHE) - each run of the protocol, uses different pubic DH parameters, which are generated dynamically. This results in a different, epehemeral key being generated every time. The public parameters are then digitally signed in some way, usually using the sender's private RSA (DHE_RSA)) or \gls{dsa} DHE_DSS) key. This key exchange offers both authenticity and PFS.

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

Table 2. Key exchange methods and security properties

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

Note that in TLS 1.3, all of static RSA and DH cipher suites have been removed: all of the PubK exchange methods now provide PFS. Even though, anonymous DH has also been removed from TLS 1.3, you can still have unauthenticated connections by either using **raw public keys** [55] or by not verifying the certificate chain and any of it's contents.

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

The ECC-based key exchange (ECDH and Elliptic Curve Diffie-Hellman Ephemeral (ECDHE)) and authentication (ECDSA) algorithms are defined in `RFC4292` [37], which is also referenced in `RFC5246` [30]. The document introduces five new ECC-based key exchange algorithms, all of which use ECC to compute the **premaster secret**, differing only in whether the negotiated keys are epehemeral (ECDH) or long-term (ECDHE), as well as the mechanism (if any) used to authenticate them. Three new ECDSA **client authentication** mechanisms are also defined, differing in the algorithms that the certificate must be signed with, as well as the key exchange algorithms that they can be used with. Those features are negotiated through the TLS Extension Mechanism.

6.1 TLS 1.2 Handshake Protocol

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

As I explained before every TLS Handshake message is encapsulated within a TLS Record. The actual Handshake message is contained within the `fragment` of the TLS Record. The Record type for a Handshake message is `0x16`. The Handshake message has the following structure: a 1-byte `msg_type` field (specifies

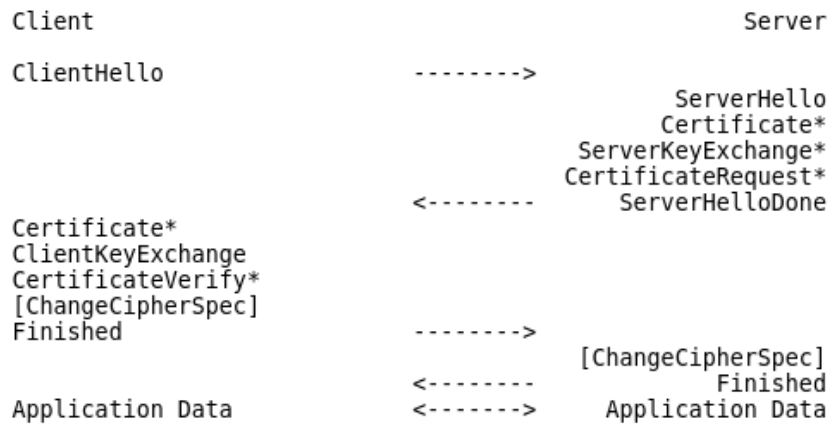

Fig. 6. TLS 1.2 message flow for a full handshake

the Handshake message type), a 2-byte `length` field (specifies the length of the `body`) and a `body` field, which contains a structure depending on the `msh_type` (similar to `fragment` field in a TLS Record).

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

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

The connections begins with the client sending a `ClientHello`, containing `.random`, `cipher_suites` and `compresison_methods`, among other fields. `cipher_suites` contains a **list** of cipher suites and `compression_methods` contains a list of compression methods (`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` message, which is similar in its contents, except that instead of containing a list of supported features, it contains a single item containing the one that it chose, *i.e.*, the server responds with the chosen `cipher_suite` and `compression_method` (note the **singular** form) that it chose from the corresponding list sent by the client. Just like in the client's case, a `random` is also present. The `version` field in the TLS Record indicates the TLS version chosen by the server and it's the one that's gonna be used for that connection.

TLS requires cryptographically secure pseudorandom values to be generated by both of the parties independetely. Those random number (or nonces) are essential for freshness (protection against replay attacks; we do need both randoms, otherwise the messages could be replayed) and session uniqueness, since both of the random values are inputs to the **master secret** generation, meaning that a new keying material is generated with every session. If the output

of the pseudorandom numbers can be predicted by the attacker, he can predict the keying material, as described in "A Systematic Analysis of the Juniper Dual EC Incident"[3]. 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 (a certificate chain): the server's certificate, every intermediate certificate and the root certificate. 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 asks the client for certificate with the CertificateRequest message. Note, that 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: only the server knows the corresponding private key, so if both of the sides generate the same keying material, then the server must be who he claims to. 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, 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 DH parameters that will allow each side to generate the same premaster secret independently.

The CertificateVerify message is sent by the client to verify its certificate, if it 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's sent by both parties to notify the reciever that subsequent records will be protected under the newly negotiated CipherSpec and keys.

The Finished message is an essential part of the protocol. It's 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 is when they can be sure that the Handshake has not been tampered with by a 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, the result will be a mismatch in computed and 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 an point in the handshake, the Alert protocol may be used by any of the peers to signal any problems or even abort the process by using an appropriate message type.

Besides the full handshake, the TLS 1.2 specification also specifies an abbreviated handshake mechanism, which can be used to either resume a previous session or duplicate one, instead of negotiating new security parameters (for example, this is useful in the context of multiple HTTPS requests for various resources, when loading a typical website). The advantage of this mechanism is that the handshake is reduced to 1 RTT, instead of the usual 2 RTT as it's the case in the full handshake. In order to do the abbreviated handshake, the client and the server must have established a session previously, by performing the full handshake. To do this, the clients sends a session ID of the session it wants to resume in its ClientHello and its up to the server to decide if he wants to resume that session, by responding with a ServerHello containing that same session ID value, or if it wants to establish a new session by sending a session ID with a different value. 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, which means that provided that the master secret has not been compromised and that the secure hash operations are secure, the new connection will be secure and independent from previous connections. The TLS 1.2 spec, suggests and upper limit of 24 hours for session ID lifetimes, since an attacker who obtains the master secret will be able to impersonate the compromised party until the corresponding session ID is retired. Note that this mechanism requires state to be maintained in both peers.

6.2 DLTS

The design of DTLS is intentionally very similar to TLS, in fact, its specification is written in terms of differences from TLS. The changes are mostly done on the lower level, and even extensions that have been defined before DTLS has even existed can be used with DTLS, The latest version of DTLS is 1.2 and it's defined in RFC 6347[43]. There a draft of DTLS 1.3 [**?**] is currently in 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 values are added to the record layer: an explicit `2 byte` sequence number and a `6 byte` epoch fields. The DTLS MAC is the same as of 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 handahke 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 which the handshake proceeds as in TLS. Although optional for the server, this mechanism highly recommended, and the client must be prepared to respond to it.
- 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.

6.3 TLS Extensions

TLS extensions were originally defined in `RFC 4366`[23] 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 TLS a 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 are not "extensions-aware" ignore the data added to the `ClientHello`s that they don't understand (section `7.4.1.2` of `RFC 2246`[**?**]), meaning that even older servers that don't support extensions, namely the ones with version of TLS prior to 1.2 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 "normal" `ServerHello` with an additional block of data following the `compression_method` field that contains 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 clients must abort the handshake.

6.4 TLS 1.3

Due to limited space, I won't be able to describe TLS 1.3 in detail. I decided to concentrate on TLS 1.2 instead, because TLS 1.3 is still in draft mode and 1.2 is the latest and the recommended version in use.

Despite the protocol name not suggesting it TLS 1.3 is very different from TLS 1.2, in fact, it should've probably been called TLS 2.0 instead. I've studied TLS 1.3 in great detail, and as I already mentioned, I have been formally recognized as a TLS 1.3 contributor 1.3. I have also participated in the mailing lists, as part of my work on the architecture of the solution.

Throughout this document, in various parts I mentioned how TLS 1.3 differs from 1.2. A lot of the changes brought in TLS 1.3 make it more suitable in the context of IoT. Some of those changes I've already mentioned previously in this document, here I will add a few more. I will not go into great detail, but will give a general overview.

The first important difference is that in TLS 1.3 extensions are required, since some of the functionality has been moved into extensions, in order to preserve backwards-compatibility with the previous versions of the ClientHellos. In fact, the way a server distinguishes if a client is requesting a 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 encrytion starts earlier. For example, on 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 and it contains any extensions that are not needed to establish the cryptographic context.

In TLS 1.3, non AEAD ciphersuites are not supported anymore. 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 or not verifying the certificate and any of its contents.

One of the main problems with using TLS in IoT is that while IoT traffic need 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, when compared to TLS 1.2 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 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 essentially encrypting it with the previously shared PSK. Session resumption via identifiers and tickets has been

obsoleted in TLS 1.3, and both methods have been replaced by a PSK mode. The PSK is established on a previous connection after the handshake is completed and can be presented by the client on the next visit.

7 Proposed Solution

Most of the work that has been done has been centered around DTLS, even though most of the solutions can be applied to both. I want explore TLS optimization more. There is clearly a need for that, specially with CoAP over TCP and TLS standard being currently developed. The CoAP protocol is the HTTP for constrained devices and the mentioned standard does not explore any TLS optimizations, and since any IoT device using it in the future would benefit them, this is an important area to explore. None of the related work centered around (D)TLS 1.3, mainly this is because the protocol is still in draft stages, I wouldn't expect however, major changes to its design at this point.

Most of the work done in this area proposes a solution that's 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 SK2[**?**] or even both. This two main issues: first, if the developer wants to use (D)TLS without using any specific protocol for which the solution was tuned for (such as CoAP), 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 that 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. My goal is to design a solution that can be used out of the box and is not tailored towards any specific protocol, that's 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 the goals above, in my solution I will heavily rely on the extension mechanism. By using extensions, I can make any deviations from the original protocol that I want, even if they completely change the protocol. This was confirmed by one of the TLS designers, after I sent this question to the official IETF mailing list[9], since none of the TLS specifications or RFC 4366 made it clear.

I want to incorporate some of the ideas from TLS 1.3, such as the 0-RTT data, which can be done if peers already share a PSK. For example, this can be very useful in the context of the temperature sensor described in the introduction, where the sensors sends data to a server every 30 seconds. With 0-RTT data, it can communicate securely with minimal costs. The use of symmetric crytpography only, has various other advantages in the context of IoT that I described above, so I will look into 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 crytpography at all. For some of them, the use of something like raw public keys, which is considered the first entry point into the area of PubK crytpography,

is acceptable, while others are powerful enough to take advantage of certificates and Public Key Infrastructure (PKI), at least up to some point. My solution 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 not require integrity, but not privacy. For those, it makes no sense to pay the energy and time overhead of encryption. I will take special care in the selection of cipher suites and key exchange methods, possibly even developing my own.

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

References

1. 6lowpan compressed dtls for coap - ieee conference publication. http://ieeexplore.ieee.org/document/6227754/, (Accessed on 12/20/2017)
2. Comparing elliptic curve cryptography and rsa on 8-bit cpus — springerlink. https://link.springer.com/chapter/10.1007/978-3-540-28632-5$_9$, ($Accessedon12/24/2017$)
3. Dualecjuniper-draft.pdf. http://dualec.org/DualECJuniper-draft.pdf, (Accessed on 12/31/2017)
4. Kryptographische verfahren: Empfehlungen und schlssellngen, version 2017-01. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf?$_blob=publicationFile,(Accessedon12/19/2017)$
5. Lightweight cryptography for the internet of things. https://pdfs.semanticscholar.org/9595/b5b8db9777d5795625886418d38864f78bb3.pdf, (Accessed on 12/26/2017)
6. Lithe: Lightweight secure coap for the internet of things - ieee journals & magazine. http://ieeexplore.ieee.org/document/6576185/, (Accessed on 12/20/2017)
7. Low-cost standard signatures in wireless sensor networks: A case for reviving pre-computation techniques? (pdf download available). https://www.researchgate.net/publication/259811495$_Low-cost_Standard_Signatures_in_Wireless_Sensor_Networks_A_Case_for_Reviving_Pre-computation_Techniques$, ($Accessedon12/24/2017$)
8. Open mobile alliance - lightweightm2m v1.0 overview. http://www.openmobilealliance.org/wp/Overviews/lightweightm2m$_overview.html$, ($Accessedon12/30/2017$)
9. Re: [tls] tls extensions: Omitting handshake messages. https://www.ietf.org/mail-archive/web/tls/current/msg24932.html, (Accessed on 01/04/2018)
10. Reaper: Calm before the iot security storm? krebs on security. https://krebsonsecurity.com/2017/10/reaper-calm-before-the-iot-security-storm/, (Accessed on 01/04/2018)
11. Recommendation for key management, part 1: General. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf, (Accessed on 12/19/2017)
12. Rfc 7252 - the constrained application protocol (coap). https://tools.ietf.org/html/rfc7252page-8, (Accessed on 12/24/2017)

13. S3k: Scalable security with symmetric keysdtls key establishment for the internet of things - soda. http://soda.swedishict.se/5920/, (Accessed on 12/30/2017)
14. sec17-antonakakis.pdf. https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-antonakakis.pdf, (Accessed on 01/04/2018)
15. Security as a coap resource: An optimized dtls implementation for the iot - ieee conference publication. http://ieeexplore.ieee.org/document/7248379/, (Accessed on 12/24/2017)
16. Smalt: The world's first interactive centerpiece — indiegogo. https://www.indiegogo.com/projects/smalt-the-world-s-first-interactive-centerpiece-health/, (Accessed on 01/04/2018)
17. Ssl library mbed tls / polarssl: Download for free or buy a commercial license. https://tls.mbed.org/, (Accessed on 12/19/2017)
18. State of the art in lightweight symmetric cryptography. https://eprint.iacr.org/2017/511.pdf, (Accessed on 12/27/2017)
19. Triple handshakes and cookie cutters: Breaking and fixing authentication over tls - ieee conference publication. http://ieeexplore.ieee.org/document/6956559/, (Accessed on 12/30/2017)
20. Ayadi, A., Ros, D., Toutain, L.: TCP header compression for 6LoWPAN. Internet-Draft draft-aayadi-6lowpan-tcphc-01, IETF Secretariat (October 2010), http://www.ietf.org/internet-drafts/draft-aayadi-6lowpan-tcphc-01.txt, http://www.ietf.org/internet-drafts/draft-aayadi-6lowpan-tcphc-01.txt
21. Badra, M., Hajjeh, I.: $\mathrm{ECDHE}_{P}SKCipherSuitesforTransportLayerSecurity(TLS).RFC5489, RFCEditor(March20$
22. Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., Moeller, B.: Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492, RFC Editor (May 2006), http://www.rfc-editor.org/rfc/rfc4492.txt, http://www.rfc-editor.org/rfc/rfc4492.txt
23. Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., Wright, T.: Transport Layer Security (TLS) Extensions. RFC 4366, RFC Editor (April 2006), http://www.rfc-editor.org/rfc/rfc4366.txt, http://www.rfc-editor.org/rfc/rfc4366.txt
24. Blumenthal, U., Goel, P.: Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport Layer Security (TLS). RFC 4785, RFC Editor (January 2007)
25. Bormann, C.: 6LoWPAN-GHC: Generic Header Compression for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs). RFC 7400, RFC Editor (November 2014)
26. Bormann, C., Ersue, M., Keranen, A.: Terminology for Constrained-Node Networks. RFC 7228, RFC Editor (May 2014), http://www.rfc-editor.org/rfc/rfc7228.txt, http://www.rfc-editor.org/rfc/rfc7228.txt
27. Bormann, C., Shelby, Z.: Block-Wise Transfers in the Constrained Application Protocol (CoAP). RFC 7959, RFC Editor (August 2016)
28. Bormann, C., Lemay, S., Tschofenig, H., Hartke, K., Silverajan, B., Raymor, B.: 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, http://www.ietf.org/internet-drafts/draft-ietf-core-coap-tcp-tls-11.txt
29. Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., Polk, W.: 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, http://www.rfc-editor.org/rfc/rfc5280.txt
30. Cusack, F., Forssen, M.: Generic Message Exchange Authentication for the Secure Shell Protocol (SSH). RFC 4256, RFC Editor (January 2006)

31. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, RFC Editor (August 2008), http://www.rfc-editor.org/rfc/rfc5246.txt, http://www.rfc-editor.org/rfc/rfc5246.txt
32. Dierks, T., Allen, C.: The TLS Protocol Version 1.0. RFC 2246, RFC Editor (January 1999), http://www.rfc-editor.org/rfc/rfc2246.txt, http://www.rfc-editor.org/rfc/rfc2246.txt
33. Eastlake, D.: Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066, RFC Editor (January 2011), http://www.rfc-editor.org/rfc/rfc6066.txt, http://www.rfc-editor.org/rfc/rfc6066.txt
34. Eastlake, D., Schiller, J., Crocker, S.: Randomness Requirements for Security. BCP 106, RFC Editor (June 2005), http://www.rfc-editor.org/rfc/rfc4086.txt, http://www.rfc-editor.org/rfc/rfc4086.txt
35. Eronen, P., Tschofenig, H.: Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). RFC 4279, RFC Editor (December 2005), http://www.rfc-editor.org/rfc/rfc4279.txt, http://www.rfc-editor.org/rfc/rfc4279.txt
36. Gutmann, P.: 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, http://www.rfc-editor.org/rfc/rfc7366.txt
37. Haberman, B.: IP Forwarding Table MIB. RFC 4292, RFC Editor (April 2006)
38. Hui, J., Thubert, P.: 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, http://www.rfc-editor.org/rfc/rfc6282.txt
39. Keoh, S., Kumar, S., Shelby, Z.: Profiling of DTLS for CoAP-based IoT Applications. Internet-Draft draft-keoh-dtls-profile-iot-00, IETF Secretariat (June 2013), http://www.ietf.org/internet-drafts/draft-keoh-dtls-profile-iot-00.txt, http://www.ietf.org/internet-drafts/draft-keoh-dtls-profile-iot-00.txt
40. Montenegro, G., Kushalnagar, N., Hui, J., Culler, D.: Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944, RFC Editor (September 2007), http://www.rfc-editor.org/rfc/rfc4944.txt, http://www.rfc-editor.org/rfc/rfc4944.txt
41. Moran, B., Meriac, M., Tschofenig, H.: 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, http://www.ietf.org/internet-drafts/draft-moran-suit-architecture-00.txt
42. Popov, A.: Prohibiting RC4 Cipher Suites. RFC 7465, RFC Editor (February 2015), http://www.rfc-editor.org/rfc/rfc7465.txt, http://www.rfc-editor.org/rfc/rfc7465.txt
43. Rescorla, E., Modadugu, N.: Datagram Transport Layer Security Version 1.2. RFC 6347, RFC Editor (January 2012), http://www.rfc-editor.org/rfc/rfc6347.txt, http://www.rfc-editor.org/rfc/rfc6347.txt
44. Rescorla, E., Ray, M., Dispensa, S., Oskov, N.: Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746, RFC Editor (February 2010), http://www.rfc-editor.org/rfc/rfc5746.txt, http://www.rfc-editor.org/rfc/rfc5746.txt
45. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. Internet-Draft draft-ietf-tls-tls13-21, IETF Secretariat (July 2017), http://www.ietf.org/internet-drafts/draft-ietf-tls-tls13-21.txt, http://www.ietf.org/internet-drafts/draft-ietf-tls-tls13-21.txt
46. Rescorla, E., Tschofenig, H., Modadugu, N.: 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, http://www.ietf.org/internet-drafts/draft-ietf-tls-dtls13-22.txt

47. Ronen, E., OFlynn, C., Shamir, A., Weingarten, A.O.: Iot goes nuclear: Creating a zigbee chain reaction. Cryptology ePrint Archive, Report 2016/1047 (2016), https://eprint.iacr.org/2016/1047
48. Salowey, J., Zhou, H., Eronen, P., Tschofenig, H.: Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077, RFC Editor (January 2008), http://www.rfc-editor.org/rfc/rfc5077.txt, http://www.rfc-editor.org/rfc/rfc5077.txt
49. Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., Adams, C.: 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, http://www.rfc-editor.org/rfc/rfc6960.txt
50. Santesson, S., Tschofenig, H.: Transport Layer Security (TLS) Cached Information Extension. RFC 7924, RFC Editor (July 2016)
51. Seggelmann, R., Tuexen, M., Williams, M.: 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, http://www.rfc-editor.org/rfc/rfc6520.txt
52. Sheffer, Y., Holz, R., Saint-Andre, P.: 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, http://www.rfc-editor.org/rfc/rfc7525.txt
53. Simon, D., Aboba, B., Hurst, R.: The EAP-TLS Authentication Protocol. RFC 5216, RFC Editor (March 2008), http://www.rfc-editor.org/rfc/rfc5216.txt, http://www.rfc-editor.org/rfc/rfc5216.txt
54. Tschofenig, H., Fossati, T.: Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things. RFC 7925, RFC Editor (July 2016)
55. Wouters, P., Tschofenig, H., Gilmore, J., Weiler, S., Kivinen, T.: Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7250, RFC Editor (June 2014)

Glossary

AC Asymmetrical Cryptography. 2, 3
AEAD Authenticated Encryption With Associated Data. 10, 11, 13, 14, 16, 17, 24

CA Certification Authority. 10
CoAP Constrained Application Protocol. 7–10, 25
CRL Certificate Revocation List. 10

DDoS Distributed Denial-Of-Service. 1
DH Diffie-Hellman. 10, 12, 18, 19, 21, 24
DoS Denial-of-Service. 23
DTLS Datagram TLS. 1, 2, 6, 8, 9, 11, 22, 23, 25

EC Elliptic Curve. 9
ECC Elliptic Curve Cryptography. 3, 8, 9, 12, 18, 19
ECDH Elliptic Curve Diffie-Hellman. 9, 19
ECDHE Elliptic Curve Diffie-Hellman Ephemeral. 19
ECDSA Elliptic Curve Digital Signature Algorithm. 9, 19

HKDF HMAC-based Extract-and-Expand Key Derivation Function. 14, 18
HTML Hypertext Markup Language. 15
HTTPS Hypertext Transfer Protocol Secure. 15

IETF Internet Engineering Task Force. 13, 25
IoT Internet Of Things. 1–3, 5, 8, 9, 11, 12, 24, 25
IV Initialization Vector. 18

MAC Message Authentication Code. 11, 13, 14, 16–18, 22, 23
MITM Man In The Middle. 12, 18, 19, 21, 22

OCSP Online Certificate Status Protocol. 10

PFS Perfect Forward Secrecy. 3, 10, 18, 19, 24
PKC Public Key Cryptography. 13
PKI Public Key Infrastructure. 26
PMS premaster secret. 18, 19
PRF Pseudo-Random Function. 14, 17, 18

PrivK Private Key. 18
PSK Pre-Shared Key. 5, 6, 9, 10, 13, 24, 25
PubK Public Key. 6, 10, 18, 19, 21, 25

RPK Raw Public Key. 6, 10
RSA Rivest-Shamir-Adleman. 18, 19

SC Symmetrical Cryptography. 2, 3
SCA Side-Channel Attack. 4
SNI Server Name Indication. 11
SSL Secure Sockets Layer. 13

TLS Transport Layer Security. 1, 2, 8–26