POLITECNICO DI TORINO

Master's Degree Course in Computer Engineering

Master's Degree Thesis

Performance and Security Evaluation of TLS, DTLS and QUIC Security Protocols

Supervisor
Prof. Antonio Lioy
Ing. Diana Gratiela Berbecaru

Candidate
Andrea Gaminara

Academic Year 2021-2022

Summary

During the last decades some protocols to implement a secure communication between a client and a server have been designed and standardized. The most used and known protocol to provide security to a communication is Transport Layer Security (TLS). The most important competitor of TLS turned out to be Quick UDP Internet Connection (QUIC), which in recent years has gained significant diffusion and use. If we consider the IoT scenario, another standard very much used is represented by Datagram Transport Layer Security (DTLS), a protocol very similar to TLS, but that is built on top of UDP instead of TCP as a difference.

The main focus of this thesis is to analyse and compare TLS, DTLS and QUIC protocols from the performance and the security points of view. After a brief introduction in which all these protocols are described in detail, in particular keeping the attention on the implementation of the security properties, such as authentication, confidentiality and integrity, different works of the past regarding the analysis of TLS, DTLS and QUIC are presented. These works have analysed the protocols using different metrics and tools, conducting experiments in different environment and scenarios. This analysis is useful in particular to understand which are the metrics and the methods used to compare security protocols' performance. Subsequently all three protocols are measured in terms of performance following some of the past approaches studied and also executing some new experiments. The target is to understand in different situations which is the best choice to adopt among all the protocols analysed. In particular the time is a recurrent metric that is measured during the experiments: from the time needed to establish a secure connection, passing through the time needed to download a page through the browser, till the time needed to exchange different sized files between client and server in different network conditions, varying bandwidth and delay values for the links connecting the two endpoints.

From the security point of view, first of all, for each protocol the evolution during years passing through different versions is described. Then all the security properties that are implemented are compared to understand which one between TLS, DTLS and QUIC offers greater data protection and security. As last point, a description is made of a series of attacks that can be conducted against the protocols analysed. For each of them, how the attack works is analysed, as well as the resistance of the protocol and possible countermeasures to be taken.

Acknowledgements

I would like to thank my supervisor Prof. Antonio Lioy, as well as Ing. Diana Berbecaru, for allowing me to participate in this project and for their guidance through the whole of it.

I am also grateful to my colleagues and all the people that I met and knew during these 5 years at Politecnico for their help and moral support.

I must express my very profound gratitude to my parents, my sister Giulia and my friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Contents

Chapter 1

Introduction

When browsing the Internet, users and web applications regularly encounter multiple possible security problems, such as authenticating the identity of the other party, data tampering, and third-party monitoring. In addition, a lot of different attacks against a communication are possible in order to retrieve some private or secret data or to simply disrupt the communication channel preventing one of the parties from using a service. For this reason it's really important to use a security protocol in order to protect the communication between the two parties. A security protocol is in charge of ensuring some security features for the channel between two peers, such as authentication of the parties, integrity and confidentiality of data in transit and authentication of the messages sent during the communication. During the last decades different protocols to protect a channel were designed and developed. They differ from each other for the security guarantees they provide and for their position in the protocol stack, that defines also which part of packet they protect. Transport Layer Security (TLS) is a common protocol used to establish a secure session between two endpoints, preventing eavesdropping, tampering, and message forgery. It's a standard in the protection of the communication over Internet and it's the most widely used security protocol. Datagram Transport Layer Security (DTLS) is another solution to protect a communication and is based on TLS, but it's designed to run over unreliable datagram protocols like UDP differently from TLS that runs over TCP. DTLS modifies the protocol to make sure it can still provide equivalent security guarantees to TLS with the exception of order protection/non-replayability. TLS and DTLS have become also a typical solution in securing the communication to/from IoT devices, that represent a really important reality nowadays and have had a huge spread in the last recent years. In 2012 Google proposed a new protocol named Quick UDP Internet Connections (QUIC) with the purpose of reducing latency in the loading of a web page and improving the performance. This protocol runs on top of UDP and implements its own encryption system (QUIC Crypto), comparable to TLS, which combines connection establishment and key agreement. In 2015 the IETF started the work to standardize the protocol, that is based on the original version but with some differences. For example the IETF version of QUIC requires the use of TLS 1.3 for key negotiation and message protection. In the following sections all the protocols will be described in detail.

Subsequently all three protocols are measured in terms of performance following some of the past approaches studied and also executing some new experiments. There are also different metrics and parameters that can be measured to compare protocols' performance and in the following sections the choice for each selected parameter to measure is explained. The target is to understand in different situations which is the best choice to adopt among all the protocols analysed. TLS and QUIC performance are compared in terms of performance in a desktop scenario when they are employed by the browser to download some known and popular webpages. Subsequently all three protocols are tested from the performance point of you, measuring the time necessary to establish a connection between client and server and the time needed to complete a file exchange between the two endpoints.

The last part of the thesis contains an analysis of TLS, DTLS and QUIC protocols from the security point of view. First of all the history and evolution of each of them is analysed, including the enhancements and improvements introduced in the design during years to provide better and

stronger communication protection. Then for each one of the analysed protocols, a description of possible and known attacks is made. Moreover, possible countermeasures for each one of the introduced vulnerabilities are explained. This section is ended with some recommendations, that include for each protocol the best version to use and some best practices to adopt in order to avoid attacks.

Chapter 2

Security Protocols

2.1 TLS 1.2

Transport Layer Security (TLS) was designed by Netscape in 1994 as the successor of Secure Socket Layer (SSL) and it is currently the most widely adopted network security protocol. TLS 1.2 [1] was standardized in August 2008 with RFC 5246 in order to introduce improvements and enhancements with respect to its predecessor (TLS 1.1 [2]), in particular from the security point of view.

TLS creates a secure transport channel on top of the layer 4, providing some important security features: peer authentication, message confidentiality, message authentication and integrity and protection against replay and filtering attacks. On top of the reliable transport protocol, e.g. TCP, there is the TLS record protocol, which is the protocol that encapsulates the data being transmitted. The TLS record is used to transmit four different kind of protocols: application protocols (like HTTP), *TLS Handshake* protocol, *TLS Change Cipher Spec* protocol and *TLS Alert* protocol.

The TLS record contains a 5-byte record header with a field, named *Content type*, that indicates which is the protocol carried by the TLS record protocol, as reported in the Fig. 2.1. The other fields of the header are *Version*, that is the version of TLS used and *Length*, that indicates the length of the data in the record excluding the header itself. Then in the TLS record there are the payload, that contains encrypted data, the MAC computed over the data and the padding, that is necessary only in case a block cipher is used to encrypt the application data.

A typical workflow for delivering application data is composed by the following steps:

- Record protocol receives application data;

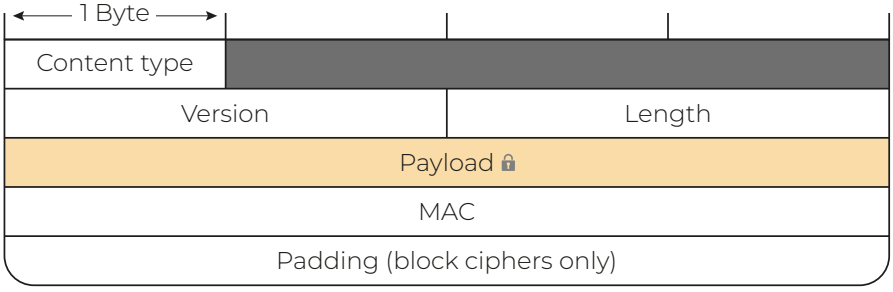


Figure 2.1. TLS 1.2 Packet

- Received data is divided into blocks (maximum 16 KB per record);
- Message authentication code (MAC) or HMAC is computed and added to each record;
- Data within each record is encrypted using the negotiated cipher.

It's a mechanism of authenticate-then-encrypt that is vulnerable to DoS attacks, especially in presence of big records. Once these steps are complete, the encrypted data is passed down to the TCP layer for transport. At the receiver these steps are followed in reverse order: decrypt record using negotiated cipher, verify MAC, extract and deliver the data to the application above it.

2.1.1 Handshake

The most important part of the protocol from the security point of view is the handshake, which is used to establish the secure TLS channel between the parties. During the handshake phase the client and the server authenticate themselves, agree on which algorithms will be used to ensure confidentiality and integrity and establish the key that will be used to protect the channel.

The first message of the handshake is called the *ClientHello*, as depicted in Fig. 2.2. This message includes the version of the protocol preferred by the client, a session identifier (0 for a new handshake, different from 0 to resume a previous session), 28 pseudo-random bytes called *client random* and the list of ciphersuites, which is the set of algorithms that client is able to use for encryption, key exchange and integrity. The server responds with a *ServerHello* message, that contains the connection parameters it has selected from the provided list, the protocol version preferred by the server (the one proposed by the client or a lower one), its own 28 pseudo-random bytes, called *server random*, and a session id (a new one for a new connection or the one proposed by the client if a session has been resumed). If the client and server do not share any capabilities, the connection terminates unsuccessfully.

Once the initial parameters are established, since server authentication is compulsory, an important message is the *Certificate* one sent by the server. In this message, the server sends its certificate chain (which includes its leaf certificate and intermediate certificates) to the client. The X.509 certificate must contain in the *Subject* field or in the *SubjectAltName* field the exact name of the server. Upon receipt, the client performs several checks to authenticate the certificate, that include checking the certificate's digital signature, verifying the certificate chain, and checking for any other potential problems with the certificate data (expired certificate, wrong domain name, etc). The client will also make sure the server has possession of the certificate's private key during the key exchange process. If the server is requesting client authentication, then it will send an explicit message to the client which is the *CertificateRequest*. In this message the server tells also from which CA the certificate must be issued, because server trusts only some CAs.

The *ServerKeyExchange* is an optional message, only needed for certain key exchange methods (Diffie-Hellman) that require the server provides additional data. In case mutual authentication is requested, then the client sends the *Certificate* message which contains the certificate for client authentication. With the *ClientKeyExchange* message the client then provides its contribution to the session key. The specifics of this step depend on the key exchange method that was decided in the initial phase of the handshake. After the reception of this message also the server can generate the pre-master secret.

The *ChangeCipherSpec* message lets the other party know that it has generated the session key and it triggers the change of the algorithms to be used for message protection, allowing to pass from the previous unprotected channel to the protection of the next messages with algorithms and keys just negotiated. The *Finished* message is then sent to indicate that the handshake is complete on the client side and it's the first data protected by the session key. The message contains a MAC computed over all the previous handshake messages with the exclusion of *ChangeCipherSpec* (since it has no information) that allows each party to make sure the handshake was not tampered with. Then the server, in its turn, decrypts the pre-master secret and computes the session key. Then it sends its *ChangeCipherSpec* message to indicate it is switching to encrypted communication. The server sends its *Finished* message using the symmetric session key it just generated and verifies

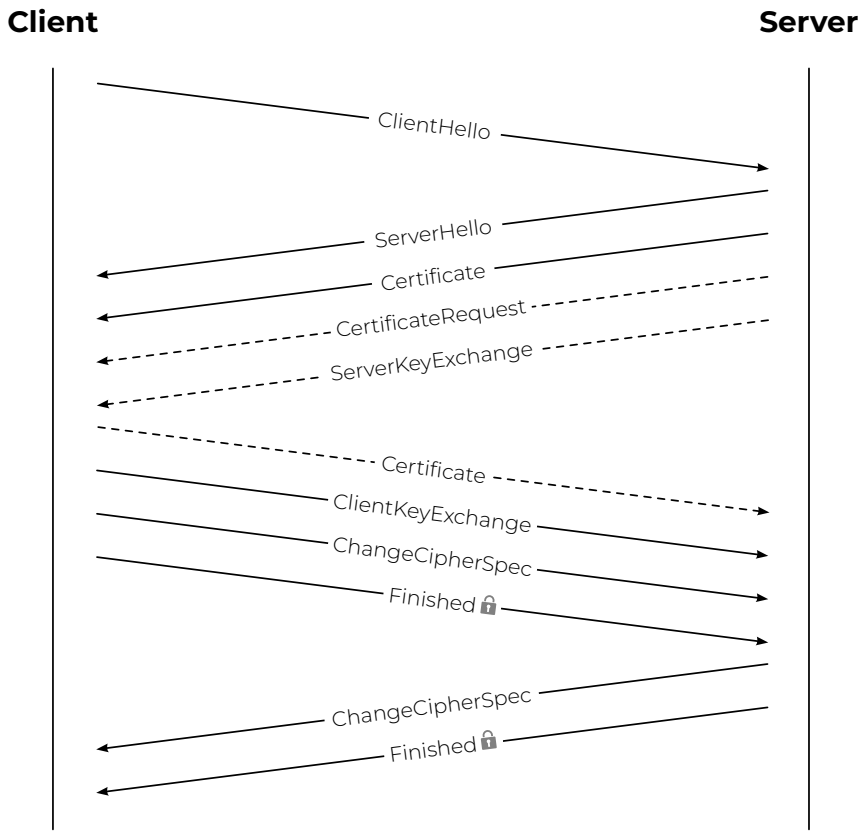


Figure 2.2. TLS 1.2 Handshake

the integrity of the handshake thanks to the *Finished* message sent previously by the client. After these steps the TLS handshake is complete. Both parties now have a session key and will begin to communicate with an encrypted and authenticated connection.

A client can ask to a server to resume a previous session without having to create a new one. In this case the handshake phase is shorter and includes the following messages: the client sends a *ClientHello* using the Session ID of the session to be resumed. If the server succeeds in finding a match of the Session ID sent by the client in its cache the session can successfully be resumed and the server responds with a *ServerHello* message containing the same Session ID value. At this point, both client and server can conclude the handshake with *ChangeCipherSpec* and *Finished* messages and then start exchanging application layer data. If the Session ID sent by the server doesn't match, the server generates a new session ID and a full handshake must be performed.

2.1.2 Authentication

In TLS 1.2 we have two possibilities for the authentication that are related to the choice made for the key exchange (Diffie-Hellman or RSA). In RSA the public/private key pair is used to both authenticate the server, as well as exchange the symmetric session key. In fact, the server is authenticated if it can successfully decrypt the pre-master secret using it's private key. With Diffie-Hellman, the public/private key pair is not used to exchange the symmetric session key. When Diffie-Hellman is involved, the private key is actually associated with the accompanying signature algorithm (ECDSA or RSA).

The RSA authentication process is related with its key exchange process. When the client is

presented with a server's TLS certificate, it performs a series of checks to verify that the certificate is valid and trusted.

- It checks the digital signature using the public key;
- It checks the certificate chain, ensuring the certificate comes from one of the trusted root certificates;
- It checks the validity period of the certificate to make sure it's not expired;
- It checks the revocation status of the certificate.

Assuming all of this is satisfactory, the authentication process terminates when the client uses the public key of the server to encrypt the pre-master secret and send it to the server. If the server can decrypt the pre-master secret and use it to compute the master secret, it's authenticated. That verifies that the server is the owner of the public/private key pair being used.

When Diffie-Hellman and ECDSA/RSA are in use, the authentication and key exchange unfold side-by-side. The asymmetric key pair is only used for the digital signature/authentication process. When the client receives the certificate it, still runs through the standard checks:

- Checks the signature on the certificate;
- Checks the certificate chain;
- Checks validity status;
- Checks revocations status.

But the proof of possession check for the private key is different. The server uses its private key to encrypt the client and server randoms, as well as its Diffie-Hellman parameter, and sends them in the *ServerKeyExchange* message during the handshake. The client can use the associated public key to verify that the server is the rightful owner of the key pair.

2.1.3 Key Exchange

The goal of the key-exchange phase is to make client and server agree on a master secret, that will be used later as the basis to compute all the cryptographic material to protect the communication: encryption keys, MAC keys per HMAC computation of TLS records and IVs (if the ciphersuite chosen by the server to protect the communication requires them). There are two possible solution for key-exchange in TLS 1.2. If the key exchange happens through RSA, it's composed by the following phases:

- The client and server exchange two prime numbers (x and y) called randoms;
- The client generates a pre-master secret (a), then uses the server's public key to encrypt it and send it to the server;
- The server decrypts the pre-master secret with the corresponding private key. Both parties now have all three inputs (client random, server random and pre-master secret) and combine them with some pseudo-random functions (PRFs) to produce a master secret;
- Both parties combine even more PRFs with the master secret and derive matching session keys.

If Diffie-Helmann is used for key exchange instead, the phases are the following:

- The client and server exchange two prime numbers (x and y) called randoms;
- One party picks a secret number called a and computes: $x^a \bmod y$. Then it sends the result (A) to the other;

- The other party does the same thing, picking a secret number b and computing x^b mod y, then sending its value (B) back;
- The two parties finish the key exchange phase by taking the given values and repeating the operation. One computes B^a mod y, the other computes A^b mod y.

  In this way both client and server are provided with the same pre-master secret. At this point all that remains is to combine the pre-master secret with the client random and the server random through the PRF to obtain the master secret.

2.2 TLS 1.3

2.2.1 Improvements with respect to TLS 1.2

TLS 1.3 [3] is the newest version of the protocol released in August 2018 and it has been designed to introduce several improvements, such as reducing handshake latency, encrypting more of the handshake in order to avoid some possible attacks and also for privacy reason (the client certificate identifies a person using a browser), improving resiliency to cross-protocol attacks and removing legacy features that are no longer needed.

TLS 1.3 avoids the use of CBC mode and authenticate-then-encrypt mechanism, that were source of possible attacks (such as Lucky13, Lucky microseconds and POODLE) in the previous version of the protocol, and it uses only AEAD (Authenticated Encryption with Associated Data) algorithms. To this end the list of supported symmetric encryption algorithms has been pruned and also ciphersuites that involve static RSA and/or Diffie-Hellman have been excluded. The use of elliptic curve algorithms is encouraged and new signature algorithms are included, such as EdDSA. Other important cryptographic improvements introduced are the redesign of the key derivation functions and the removal of of compression, the Digital Signature Algorithm (DSA), and custom Ephemeral Diffie-Hellman (DHE) groups.

With TLS 1.3 the handshake is reduced from 2-RTT to 1-RTT. Specifically, the *ClientHello* message is combined with the list of ciphersuites and the client part of the key share and the same happens for the server side, in which the *ServerHello* message is combined with the key exchange part from the server and the selected ciphersuite, as can be noticed looking at Fig. 2.3. In addition all handshake messages after the *ServerHello* are encrypted. The new EncryptedExtensions message provides confidentiality for various extensions sent in the *ServerHello* message. It's also possible to resume a previous session and further reduce the handshake time to 0-RTT using PSK (Pre Shared Key), at the cost of certain security properties. The client in this case can send early data along with the first handshake message, protected with a key derived from the PSK, without having to wait for the completion of a new handshake.

2.2.2 Handshake

In TLS 1.3 the *ServerHelloDone*, *ChangeCipherSpec*, *ServerKeyExchange* and *ClientKeyExchange* messages have been removed with respect to TLS 1.2 handshake. In particular the content of the *ServerKeyExchange* and *ClientKeyExchange* messages has been moved to extensions in the *ClientHello* and *ServerHello* messages for backward compatibility with TLS 1.2. For what concerns the authentication mode the one based on public key is the most used and provides always perfect forward secrecy, using an ephemeral Diffie-Hellman exchange.

The TLS 1.3 handshake starts with a *ClientHello* message that contains the *client random*, the list of supported protocol versions and the list of the ciphersuites (composed as pairs of symmetric cipher and HKDF hash). In addition a part of the message is reserved to extensions for key exchange, that involve the "key_share" extension for the portion of the key of the client generated using (EC)DHE and/or eventually the "pre_shared_key" extension containing a set of pre-shared key labels. The server answers with a *ServerHello* message that contains the *server random*, the selected protocol version and the selected ciphersuite. It also includes an extension for key agreement with the client: if (EC)DHE key establishment is used, then the *ServerHello* contains

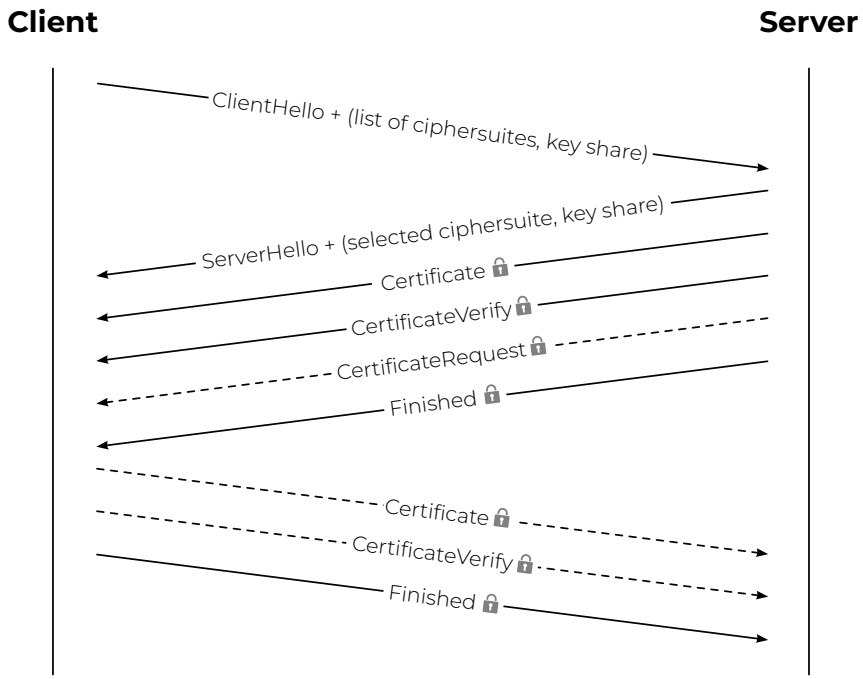


Figure 2.3. TLS 1.3 Handshake

a "key_share" extension with the server's ephemeral Diffie-Hellman portion of the key compatible with the client one, otherwise if PSK key establishment is preferred, then the *ServerHello* contains a "pre_shared_key" extension containing the selected PSK from the ones proposed by the client.

The server can optionally send to the client an *EncryptedExtensions* message in response to *ClientHello* extensions and also a *CertificateRequest* message, needed only in case of mutual authentication. In addition the server send a *Certificate* message, containing the server X.509 certificate, a *CertificateVerify* message, that is a signature computed over the entire handshake with the private key corresponding to the public key in the certificate, and the *Finished* message, that is a MAC over the entire handshake. All these last four messages are encrypted with a temporary handshake key. After that the server can send application data encrypted with the session key (long term key), but all these messages are sent to an unauthenticated peer, since all the information to authenticate the client (if requested) are not received yet.

The client completes the handshake with *Certificate* and *CertificateVerify* messages if client authentication is requested and its *Finished* message. Also these messages for the client are protected with the temporary key. At this point the handshake is completed, so client and server can derive the keying material and start communicating in a protected way.

2.2.3 Session Resumption, PSK and 0-RTT Data

The pre-shared key (PSK) authentication mode can be used to resume a session and it optionally uses a Diffie-Hellman exchange for key agreement. PSKs can be established in a previous connection and then used to resume the previous session. After a full handshake process, the server can send the client a PSK identity that corresponds to a unique key derived from the initial handshake. PSK could also be OOB (e.g., generated out-of-band, from a passphrase), which is risky if the passphrase is not sufficiently random so that a brute-force attack could be possible. In general, OOB-PSK is present but discouraged. The PSK identity can be used by the client in future handshakes to negotiate the use of the associated PSK. The client uses the "pre_shared_key" extension in the *ClientHello* message to negotiate the use of the PSK mode. This extension contains the

PSK identity and the so called binder. The PSK identity identifies the shared secret and the binder is a structure that contains one or multiple MAC values. If the server accepts the PSK, then the security context is related to the original connection and doesn't need to be negotiated from scratch. Consequently a lower number of messages during the handshake is needed.

If PSKs are used alone they don't provide perfect forward secrecy, while this property can be obtained combining PSKs and (EC)DHE key exchange. As the server is authenticating via a PSK, the *Certificate* and the *CertificateVerify* messages are no more needed. When a client tries to negotiate the use of PSK, it should also provide a "key_share" extension to the server in order to enable anyway a full handshake if the server refuses the client proposal. If the server accepts, it must respond with a "pre_shared_key" extension to select a PSK and can also add a "key_share" extension for (EC)DHE key establishment. When PSKs are provisioned out of band, the agreement about the PSK to use includes also a KDF hash algorithm.

When clients and servers share a PSK (either obtained externally or via a previous handshake), TLS 1.3 allows a client to send application data, called "early data", protected with a key derived from the PSK, already with the first message. This mode represents a new feature in TLS 1.3 and does not have an equivalent in earlier TLS versions. There are two problems related to this solution: the key used to protect early data doesn't provide perfect forward secrecy, since it comes from something (the PSK) that remains the same for all the connections within the same session and there is the risk of some kind of replay attack, because the key is created only at the client side, without any contribution from the server.

2.2.4 Authentication

In TLS 1.3 the authentication and digital signatures algorithms have been removed from the cipher suites to simplify negotiation. The three main signature algorithms for authentication allowed in TLS 1.3 are:

- RSA (signature only);
- Elliptic Curve Digital Signature Algorithm (ECDSA);
- Edwards-curve Digital Signature Algorithm (EdDSA).

EdDSA is an elliptic curve-based algorithm. Unlike in the TLS 1.2 handshake, the authentication portion of the TLS 1.3 handshake isn't coupled with the actual key exchange itself. Rather, it's handled alongside the key exchange and message authentication. In TLS 1.3 the server signs the hash of all the handshake messages when it returns the *ServerHello* in order to verify the integrity of the handshake. The client receives all of the information contained alongside the *ServerHello* and runs the standard series of checks to authenticate the TLS certificate it was just presented with. That includes checking the signature and the validity on the certificate and also verifying the signature that was added to the hash sent by the server. A match proves possession of the private key by the server.

2.2.5 Key Exchange

In TLS 1.3 the key exchange part has been heavily changed with respect to TLS 1.2: RSA key exchange is no more permitted, so RSA keys can still be used but only for server authentication not for key exchange. The problem is that RSA key exchange is not enabling perfect forward secrecy and it is also difficult to implement correctly. In this way we are automatically protected against Bleichenbacher and ROBOT attacks that exploits RSA key exchange. Rather than using it TLS 1.3 uses Diffie Hellman ephemeral, but arbitrary parameters are not permitted because there are attacks such as LogJam and WeakDH that force the server to use small numbers for DH parameters (just 512-bits), so the attacker is able to attack DH exchange. To avoid those kind of problem TLS 1.3 uses Diffie Hellman ephemeral with few predefined groups so clients and servers can only choose among these groups from a list (and not create parameters on their own).

Because of the limited selection of key exchange schemes, the client can successfully guess the scheme and send its part of the key share during the opening portion (*ClientHello*) of the handshake. At the start of the TLS 1.3 handshake, knowing that a DHE key agreement scheme will be used, the client includes a key share based on the guessed key exchange scheme with its *ClientHello*. The server receives this information and, provided the client guessed correctly, returns its own portion of key share with the *ServerHello*. Then client and server can compute the session keys.

2.3 DTLS 1.2

The Datagram Transport Layer Security (DTLS) protocol has been designed by the IETF in order to provide secure communication for datagram protocols, such as UDP. Version 1.2 of the protocol has been standardized in 2012 (RFC 6347 [4]). It provides security guarantees similar to TLS, in fact it allows client and server applications to communicate with one another preventing eavesdropping, tampering, and message forgery. However, DTLS introduces some minimal changes with respect to TLS, in order to deal with the unreliable nature of the underlaying datagram transport protocols. First, stream ciphers, such as RC4, cannot be adopted, and an explicit sequence number is included in every DTLS message. If datagrams always arrived in order, it would be sufficient for a DTLS end point to keep track of the most recent record seen in order to detect replays. But since datagrams may also arrive out of order, a replay window mechanism is required. From the security point of view DTLS does not offer any improvements over TLS. In order to deal with packet loss DTLS uses local timeouts and message retransmission policies. Finally, when an invalid message is received it can be discarded silently and the associated DTLS connection may not be terminated. Communication among two DTLS peers relies on secure sessions, identified by a unique session ID chosen by the DTLS server. Messages are transmitted as a series of DTLS records, that can carry four DTLS upper protocols:

- Handshake protocol;
- Alert protocol;
- Change Cipher Spec protocol;
- Application Data protocol.

The Handshake protocol negotiates algorithms and key material to establish a secure authenticated session between two peers. The Alert protocol is used to signal a session closure or eventual errors, while the Change Cipher Spec protocol signals the change in the algorithms used to protect the channel.

The DTLS record is prepended with a 13 bytes long header, that composed by various fields, as depicted in Fig. 2.4. The *Type* field indicates the higher level protocol used to process the enclosed data, while the *Version* field states the employed version of DTLS protocol. The *Length* field represents the size of the actual application data conveyed in the record. Then with respect to TLS, two additional fields are present, namely *Epoch* and *Sequence Number*. The Epoch number is used by endpoints to determine which encryption parameters have been used to protect the record payload and is required to resolve ambiguity that arises when data loss occurs during a session renegotiation. Instead, the *Sequence Number* is incremented for every new message transmitted by the same peer over the same DTLS connection. The concatenation of the *Epoch* and *Sequence Number* fields is considered as a single 64 bit fresh value, which is used to compute a Message Authentication Code for assuring integrity of protected DTLS records. The record header is either followed by the plaintext, if no security has been negotiated yet, or by an encrypted message. If encryption has been performed using a block cipher, the message is prepended by a random Initialization Vector (IV), which has the size of the block. The data are followed by an HMAC which allows the receiver to detect if the DTLS record (including the DTLS header) has been altered. Finally, the message is padded to a multiple of the block size if a block cipher has been used.

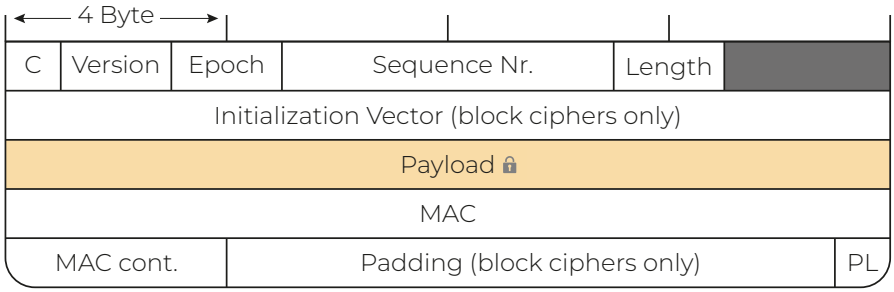


Figure 2.4. DTLS 1.2 Packet

2.3.1 Handshake

The handshake phase, as in TLS, establish a new secure session between a client and a server. It's possible to reuse a previous established session in order to reduce the number of messages exchanged during the handshake. The client starts the handshake process sending the *ClientHello* message to the server. Multiple handshake messages can be grouped together in a single *Flight*. The DTLS handshake is nearly identical to the TLS one. There are two major changes:

- Stateless cookie exchange to prevent denial of service;
- Message fragmentation and re-assembly.

Because the DTLS handshake takes place over datagram transport that is unreliable it is vulnerable to two denial of service attacks: the standard resource consumption attack and an amplification attack where an adversary can repeatedly transmit *ClientHello* messages to a DTLS server, so triggering the establishment of new DTLS sessions. These attacks lead the server to allocate and waste a lot of resources and also to send a large message, the *Certificate* one, to the victim.

To mitigate these attacks, DTLS uses the cookie exchange technique. It's based on a *HelloVerifyRequest* message that includes a locally generated cookie, that is sent by the server to the client after it has received the *ClientHello* message. The cookie generation should not require keeping state on the server, in order to avoid memory consumption denial of service attacks. One possible solution is to create the cookie from a keyed hash of the client IP address, using a global key. Upon its reception, the client must reply with a second *ClientHello* message, including the same cookie received from the server. Then, the server proceeds with the rest of the handshake only if it successfully verifies the cookie received in the second *ClientHello* message. With this solution attacks performed with spoofed IP addresses are more complicated, because the client has to receive the cookie in order to continue with the handshake. It's possible to avoid the cookie exchange phase if the client is trying to resume a previous session and it is providing the server a valid session ID. In this case the identity of the client must have been previously established.

The *ClientHello* message contains the protocol version supported by the client as well as the cipher suites. The server answers with its *ServerHello* message that contains the cipher suite chosen from the list offered by the client. The server also sends a X.509 certificate to authenticate itself followed by a *CertificateRequest* message if mutual authentication is requested. If requested the client sends its own *Certificate* message at the beginning of Flight 5.

The *ClientKeyExchange* message contains half of the pre-master secret encrypted with the server's public RSA key from the server's certificate. The other half of the pre-master secret was created and transmitted unprotected by the server in the *ServerHello* message. Since the first half of the pre-master secret is encrypted with the server's public key, it's a proof of possession of the corresponding private key. If the server doesn't control the correct private key the handshake cannot be completed. All the keys used to protect the channel established during the handshake are derived from the pre-master secret. In a similar way the client proves the possession of the

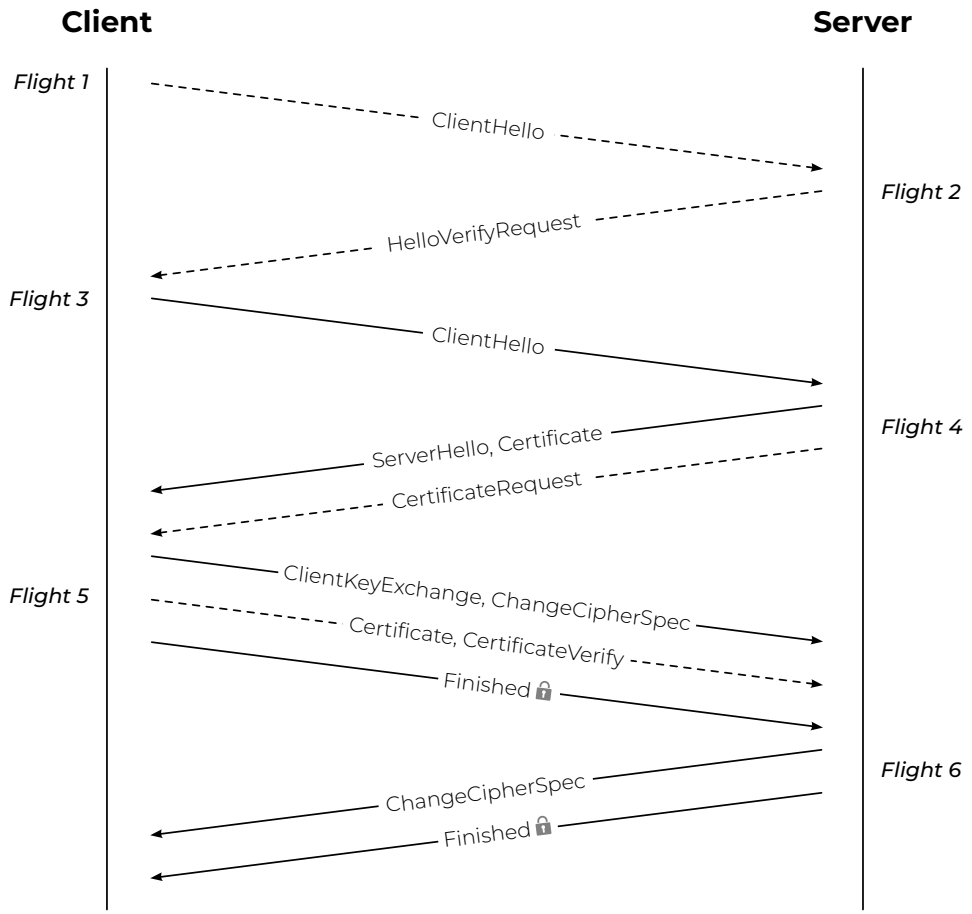


Figure 2.5. DTLS Handshake

private key that matches the public key in its certificate through the *CertificateVerify*. It does this by signing a hashed digest of all previous handshake messages with its private key. The server can verify this through the public key of the client received within the client certificate.

The *ChangeCipherSpec* message indicates that all following messages by the client will be encrypted with the negotiated cipher suite and keys. The *Finished* message contains an encrypted message digest of all previous handshake messages in order to verify the integrity of the handshake. The server answers with its own *ChangeCiperSpec* and *Finished* message to complete the handshake. Some handshake messages are too large to fit in a single DTLS record and therefore must be fragmented across multiple records. The DTLS handshake layer is responsible for reassembling these records into a coherent stream of complete handshake messages. This solution requires retransmission as well as a more complicated message format.

2.3.2 Timeout and restransmission

DTLS needs a mechanism for retransmission because it's built on top of UDP and so DTLS handshake messages may be lost. Each DTLS end-point has a timer and keeps retransmitting its last message every time the timer expires until a reply is received. If the data fragment that arrives is the expected next handshake message then the fragment is returned to the higher layers and the timer is cancelled. Otherwise, the fragment is buffered or discarded as appropriate and the timer keeps going. If the timer expires before the reply expected arrives, the last flight of messages is retransmitted.

Three different states are possible for client and server. In the PREPARING state, the endpoint is preparing the next flight of messages. When the setting phase is finished, the peer enters the SENDING state and transmits the buffered flight of messages. Once the messages have been sent, if this is the last flight in the handshake the endpoint enters the FINISHED state, otherwise enter again the WAITING state setting a retransmit timer.

When an endpoint is in the WAITING state there are two possibilities: the timer expires without receiving any messages or the endpoint receives a flight of messages. In the first case the peer has to send again the flight (entering the SENDING state) and then it returns to the WAITING state. The same thing happens if the endpoint reads a retransmitted flight from the peer, that likely means that the timer has expired on the peer and a part of a previous flight was lost. If the endpoint receives the correct flight of messages of the sequence and it is the final flight, it can enter the FINISHED state. If a new flight must be sent, the endpoint transitions to the PREPARING state.

A DTLS client starts the handshake in the PREPARING state, while a DTLS server starts in the WAITING state, with empty buffers and no retransmit timer.

2.4 Google QUIC

QUIC [5] is a transport protocol developed by Google that works on top of UDP. QUIC support was initially added to Chrome in June 2013 for the development team, and in early 2014 a tiny number of users were allowed early access. Gradually the number of users increased and QUIC was enabled for all users of Chrome and the Android YouTube app by January 2017. Many emerging protocols are using UDP (and even QUIC) as a substrate to target specific application use-case. QUIC is a user-space protocol that allows for faster development and deployment and it replaces the traditional (TCP, TLS and HTTP/2) HTTP(S) stack.

QUIC is designed to meet several goals, including deployability, security, and reduction in handshake and head-of-line blocking delays. The QUIC protocol combines its cryptographic and transport handshakes to minimize connection setup time. It's possible to have multiplexing of different requests/responses over a single connection by providing each with its own stream, so that no response can be blocked by another. Loss recovery is improved by using unique packet numbers to avoid retransmission ambiguity and by using explicit signaling in ACKs for accurate RTT measurements. A QUIC connection is uniquely identified by a CID (Connection IDentifier) at the application layer and not by the pairs of IP addresses and port number. The problem with IP address-related identifier is that when an handover between two networks has to be managed the connection must be re-established. With it's own CID, QUIC solves this problem and connections can migrate across IP address changes. Another advantage of using CID is that QUIC can provide native support to multi-path and so a mobile device could use all the network connections for the same CID. QUIC employs flow control both at the connection level as well as the stream level to limit the amount of data buffered at a slow receiver and ensures that a single stream does not consume all the receiver's buffer by using per-stream flow control limits.

QUIC packets are fully authenticated and mostly encrypted, except for a few early handshake packets and reset packets. The parts of the packet header that are not encrypted are useful for routing and decryption of the packet. The QUIC packet header is composed by various fields: *Flags*, *Connection ID*, *Version*, *Diversification Nonce*, *Packet Number*, as shown in Fig. 2.6. *Flags* encodes the presence of the *Connection ID* field and length of the *Packet Number* field and must be visible to read subsequent fields. The *Connection ID* is useful for routing and identification purposes. The *Version* and *Diversification Nonce* fields are only present in early packets. The server generates the diversification nonce and sends it to the client in the *SHLO* packet to add entropy into key generation. The packet number is used by both sides of a QUIC connection as a per-packet nonce, which is necessary to authenticate and decrypt packets. In order to let the receiver decrypting packets also if they are received out of order, the packet number is not encrypted. The header of a QUIC packet is followed by one or more frames that could encapsulate data from different streams, implementing in this way QUIC stream multiplexing.

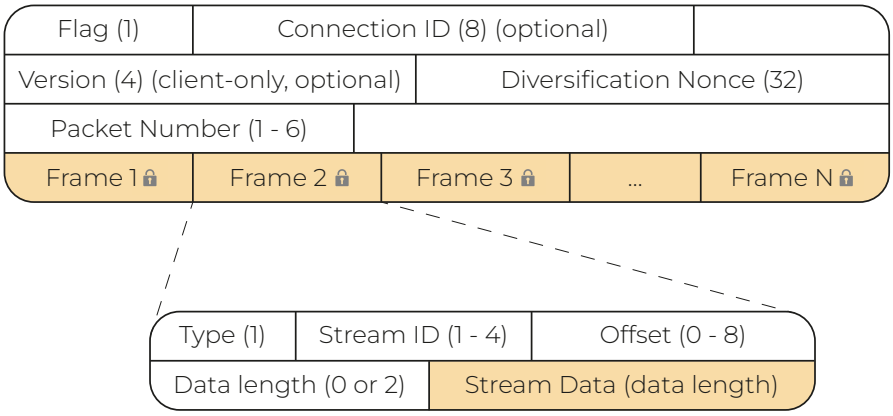


Figure 2.6. Google QUIC Packet

2.4.1 QUIC Crypto

The QUIC Crypto protocol [6] is the part of QUIC that provides transport security to a connection. Given its nature, QUIC allows to perform a combined cryptographic and transport handshake in order to set up a secure transport connection. When a connection is successfully created the client keeps information about the server in a cache. This information can be exploited later to resume the session without having to perform the whole handshake (with no round trips). Using a resumed session the client can send data immediately after the handshake packet without waiting for a reply from the server.

In TLS, the server picks connection parameters for each connection based on the client's advertised support for them, while QUIC Crypto protocol states that server preferences are completely static and are contained inside a *server config* with Diffie-Hellman public values. This *server config* is characterized by an expiry date and is signed by the server with its private key. Since this element is static, there's no need that the server executes a timing operation every time a new connection is established. On the contrary, till the same *server config* is not expired, the same signature is sufficient for many connections.

Client and server in Google QUIC agree on the keys to be used to protect the exchanged data using Diffie-Hellman. The *server config* contain the Diffie-Hellman value of the server, while the client provides its part in the first handshake message. Because the *server config* must be kept for some time in order to allow 0-RTT handshakes, this puts an upper bound on the forward security of the connection. As long as the server contribute to Diffie-Hellman key exchange remains the same, the encrypted data exchange within connection established with the same *server config* could be decrypted if they leak.

In addition the necessary key material is generated by an HMAC-based key derivation function (HKDF) with hash function SHA-256, that uses the approved two-step key derivation procedure specified in NIST SP 800-56C.

The design of QUIC Crypto protocol involves also the solution for the IP address spoofing problem. The IP address spoofing problem is addressed by issuing the client a *source address token*. It's an authenticated-encryption block that contains, at least, the client's IP address and a timestamp by the server. The server will only send a *source address token* to a client for a given IP address and when it receives it back by the client, it considers it as a proof of ownership of that IP address.

If a client remains on the same IP address, it can use a source address token and include it inside future requests in order to demonstrate ownership of their source IP address and avoid the necessary additional round trip to obtain the fresh token. If the client has changed its IP address or the client doesn't have a token, then the server may reject the connection and return a fresh token to the client.

In TLS, the reply protection is provided inserting in the key derivation process also two nonces, one generated by the client (client random) and one generated by the server (server random), so that the freshness of each party is ensured. Providing replay protection without input from the server is fundamentally very expensive and requires consistent state at the server. Consequently Google QUIC doesn't provide replay protection for the data sent by the client before the first response by the server. It's up the application to ensure that any such information is safe if replayed by an attacker.

2.4.2 Handshake

In QUIC the handshake time is reduced with respect to TLS 1.2 and TLS 1.3, indeed it takes only 1-RTT for first time connections, while for TLS 1.2 3-RTT are required (1 for TCP + 2 for TLS) and for TLS 1.3 2-RTT are needed (1 for TCP + 1 for TLS). Also for resumed sessions the handshake for QUIC takes a shorter time: 0-RTT for QUIC, while 2-RTT and 1-RTT for TLS 1.2 and TLS 1.3 respectively. This is because QUIC by design has TLS negotiation built into its protocol, whereas before starting a TLS handshake you need to set up a TCP channel separately.

The QUIC handshake starts with the client sending a client hello (*inchoate CHLO*) message to the server in order to receive back some information, as shown in Fig. 2.7. The server answers with a *REJ* message that contains:

- a *server config* that includes the server's long-term Diffie-Hellman public value;
- a certificate chain authenticating the server;
- a signature of the *server config* using the private key of the server;
- a source-address token: an authenticated-encryption block that contains the client's publicly visible IP address (as seen by the server) and a timestamp by the server.

With the source-address token the client can demonstrate the ownership of its IP address and can use it to resume the session during later handshake. The client must verify the *server config* checking the certificate chain and the signature. At this point the client can continue the handshake with *complete CHLO* message, that contains the client's ephemeral Diffie-Hellman public value.

After that the client can compute initial keys for the connection from the server's long-term Diffie-Hellman public value and its own ephemeral Diffie-Hellman private key and can send application data to the server protected with the initial keys to reach 0-RTT handshake without waiting for the server reply. If the handshake is successful, the server returns a server hello (*SHLO*) message. This message is encrypted using the initial keys and contains the server's ephemeral Diffie-Hellman public value. At this point both sides can calculate the final or forward-secure keys for the connection because each one peer has received the public value from the other one. Indeed with the *SHLO* message both sides switch sending packets encrypted with the forward-secure keys. QUIC's cryptography therefore provides two levels of secrecy: initial client data is encrypted using initial keys, while all the other data are encrypted using forward-secure keys. The forward-secure keys are better in terms of protection with respect to initial keys because they are ephemeral and so provide perfect forward secrecy. The client caches the server config and source-address token and when it connects again to the same server can use them to start the connection with a complete *CHLO*. If at least one of these two values is expired or the server has changed certificates the handshake will fail even if the client sends a complete *CHLO*. In this case, the server replies with a *REJ* message, just as if the server has received an inchoate *CHLO* and the handshake restarts from the beginning.

2.4.3 Multiplexing

TCP protocol requires that packets are delivered in order at the connection level and this can lead to a head-of-line (HoL) blocking problem. This is describing the situation in which there

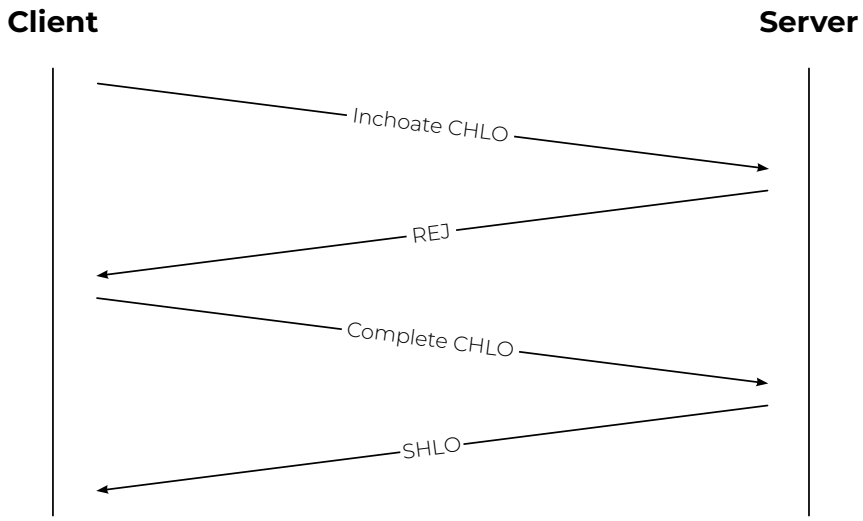


Figure 2.7. Google QUIC Handshake

are different parallel transfers over a single TCP connection and one packet is lost. Given the nature of TCP, the entire connection is stopped until the lost packet is re-transmitted. QUIC tries to overcome this problem by design, introducing the stream multiplexing in transport and allowing data to be delivered in order at the stream level. If a packet is lost, only the stream the packet belongs to is affected and not other streams. Streams are identified by stream IDs, which are statically allocated as odd IDs for client-initiated streams and even IDs for server-initiated streams to avoid collisions. A stream is created when one of the two peers sends the first bytes on an unused stream and is closed when someone sends the last frame of the stream, the one with the FIN bit set. If a stream is no more needed, it can be cancelled without having to tear down the entire QUIC connection. For implementing stream multiplexing, a single QUIC packet can carry stream frames from multiple streams and each frame incapsulates stream data.

2.4.4 Loss Recovery, FEC and Packet Pacing

Another QUIC's feature is the mechanism for loss recovery: this is based on the same mechanism used by TCP but QUIC simplifies it. Monotonically incresing packet numbers can help the loss recovery in QUIC because it's easier to distinguish an original packet from a retransmitted one. The packet number represents an explicit time-ordering, which enables simpler and more accurate loss detection than in TCP. Another important help in loss detection comes from the precise RTT estimation that is possible thanks to the acknowledgements, that encode the delay between the receipt of a packet and its acknowledgment being sent, and the packet numbers. QUIC supports also multiple ACK ranges, which helps in speeding up recovery and reducing spurious retransmissions and QUIC ACKs are irrevocable, reducing the memory pressure on the sender. QUIC supports pluggable congestion control and is designed to support different congestion control algorithms (the default is a modified version of TCP NewReno).

Another design feature of QUIC that involves packet losses is the Forward Error Correction (FEC). Since this module gives the possibility to recover in a fast way a lost packet, it's very useful to reduce the effect of the HoL blocking problem over a single QUIC stream. This mechanism is based on the concept of a FEC Group (Fg): it's composed by a series of packets and the FEC packet, that is the XORsum of the payloads of the other packets of the group. The size of the FEC group is the number of packets of the series plus one. The FEC packet gives the possibility to recover at most one packet loss within the FEC group. The FEC group size is a really important

parameter: a small FG implies high redundancy at cost of bandwidth, whereas a large FG implies low redundancy at low bandwidth cost.

With TCP data are sent as fast as possible, but if the speed is too high, the number of losses increases and the transmission can become very bursty because the congestion windows is repeatedly decreased and increased between losses. Packet Pacing is an approach used by QUIC to make the transmission less bursty by not sending at full rate. The performance of fast connections is reduced in this way, but this mechanism is really helpful in scenarios with low bandwidth.

2.5 IETF QUIC

Starting from 2015 a working group of IETF worked on the standardization of QUIC protocol, including several redesigns of features. The results of this work are contained in RFC 9000 [7] (2021). One of the most important purpose was to reconcile the cryptographic handshake with other standards. Google QUIC used its own handshake called QUIC-Crypto, that introduced the idea of a 0-RTT cryptographic handshake. TLS 1.3 was, in part, inspired by QUIC-Crypto, trying to integrate all the latency improvements that QUIC-Crypto provided. The working group finally decided that QUIC had to use TLS 1.3 for the handshake instead of the proprietary QUIC-Crypto, but still keeping QUIC's own record layer, meaning that the transport of the handshake messages would be managed by QUIC.

2.5.1 Stream

Streams are abstraction offered by QUIC to the application protocols in order to deliver information over the connection in the form of ordered sequences of bytes. QUIC provides the possibility to create bidirectional streams, which allow for data to be sent by both endpoints (in both directions), and unidirectional streams, which allow only the endpoint that initiated the stream to send data to its peer. The duration of a stream can coincide with the duration of the connection and in this case the stream is called long-lived. Streams can be created and cancelled by both the endpoints. Streams can operate concurrently and each stream can send an arbitrary amount of data, subject to flow control constraints and stream limits. There are no guarantees provided by QUIC about the ordering between bytes on different streams.

Each stream is characterized by a 62-bit integer, called stream ID, that identifies the stream within a connection, since it's unique for all the streams of the same connection. Consequently a stream ID cannot be reused by whatever endpoint if the connection is not changed. The two least significant bit of the stream ID are important to determine the characteristic of the related stream. If the identifier is 0-terminated it means that the stream has been initiated by the client, while if the least significant bit is equal to 1it indicates a server-initiated stream. if the second least significant bit of the stream ID is equal to 0 the stream is bidirectional, while unidirectional streams are characterized by the same bit set to 1.

An endpoint uses stream frames to encapsulate (in order) and send data by an application. In addition endpoints must be able to deliver stream data to an application as an ordered byte stream, and so a buffer for each endpoint in charge of keeping data received out of order, respecting always the flow control limit, is needed.

2.5.2 Connection

A connection between a client and a server is created through an handshake phase, during which the two peers establish the cryptographic parameters that will be used to protect the communication and negotiate the application protocol. An application protocol can use the connection during the handshake phase with some limitations. The time needed to complete an entire QUIC handshake is equal to 1-RTT, but this latency can be reduced at the expense of some security guarantees. Indeed a client can send 0-RTT application data, before having received server response but exposes itself to possible replay attacks. Also a server can also send application data

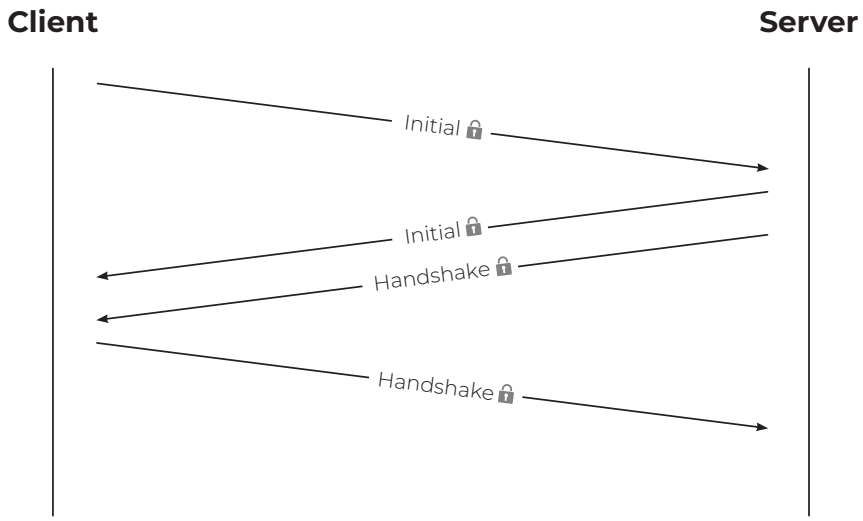


Figure 2.8. IETF QUIC Handshake

to a client before the handshake is completely finished, without waiting for the confirmation of the identity of the client contained in the final cryptographic handshake messages. QUIC connections are not strictly bound to a single network path, and connection migration is allowed thanks to the use of Connection ID. So it's possible to transfer a QUIC connection from a network path to a new one, both as a direct choice of an endpoint and when forced by a change in a middlebox.

Google QUIC design employed an 8-byte Connection ID chosen by the client to identify a connection. This design had two fundamental problems, however:

- A server had no control over the Connection ID and the server's infrastructure could therefore not include information in the ID that would be necessary for routing a connection's packets to the correct server;
- The Connection ID was not based on the IP address of the client and so it supports client migration to a new network attachment point (a client moving from a WiFi network to a cellular network, for instance). Doing so there was the possibility of leakage of private information, since a third-party observer could correlate a client's movements based on the otherwise unrelated networks where the same Connection ID showed up.

To solve this problems each connection is provided with a set of connection IDs, that identify the connection. Each endpoint selects the connection IDs for the other peer of the communication. In order to guarantee connection migration and to avoid problems for QUIC packets within a connection when changes in addressing at lower protocol layers (UDP, IP) happen, the IETF working group also built a mechanism for both endpoints to change their Connection IDs mid-connection. This allowed a migrating client to move across networks without breaking the connection, changing the Connection IDs and avoiding any privacy leakage. In addition, using this strategy, multiple connection IDs are used so that an observer cannot identify packets sent by an endpoint as belonging to the same connection without the cooperation from the endpoint itself. Connection IDs does not contain any information that can be used by an external observer to correlate them with other connection IDs for the same connection.

Packets with long headers include Source Connection ID and Destination Connection ID fields. These fields are used to set the connection IDs for new connections. Packets with short headers only include the Destination Connection ID and omit the explicit length. The length of the Destination Connection ID field is expected to be known to endpoints.

2.5.3 Handshake

QUIC relies on a combined cryptographic and transport handshake to minimize connection establishment latency. QUIC uses the CRYPTO frame to transmit the TLS 1.3 cryptographic handshake messages. The feature of QUIC protocol is that the handshake is encrypted as much as possible since packet protection is adopted for handshake messages and in addition ordering and reliable delivery is granted for handshake data. The cryptographic handshake must provide the following properties:

- Mandatory server authentication and optional client authentication, that lead to authenticated key exchange;
- Each connection produces distinct and unrelated keys, and 0-RTT and 1-RTT packets can be protected with the same keying material;
- Authenticated exchange of client and server transport parameters and confidentiality for server transport parameters;
- Authenticated negotiation of an application protocol.

In order to execute the handshake QUIC must be able to transport TLS handshake messages and it uses two functions to interface with TLS: the first function gives QUIC the possibility to request handshake messages to TLS, while the second one is used by QUIC handshake messages to TLS. Before the handshake TLS receives by QUIC the transport parameters to carry. In order to send the first packet and then start the handshake, QUIC client must request to TLS the first bytes to send. After the reception, the server has to provide these messages to TLS in order to start the handshake process on his side.

During the handshake four different types of CRYPTO frame are exchanged between client and server: Initial, 0-RTT, Handshake, and 1-RTT packets. Each crypto frame is characterized by a distinct encryption level, that determines also the keys that are used to protect the data. QUIC packets are created putting inside CRYPTO frames, that are in turn filled with the unprotected content of TLS handshake records. Instead of using TLS record protection, QUIC packet protection is preferred to secure data exchanged between client and server. Each endpoint has a current sending encryption level and a current receiving encryption level at any time during the handshake. When an endpoint receives a QUIC packet containing a CRYPTO frame from the network, it has to process that packet. If the packet uses the current TLS receiving encryption level, all the data received is added to the current input flow into the proper location. Subsequently the sequence of data will be sent to the TLS stack in the proper order. A protocol violation may happen if the data received belongs and extends the flow of an old encryption level. If the packet is characterized by a new encryption level, it is kept by the endpoint and it will be provided to TLS when the correct encryption level will become the current one. Handshake messages that arrive out of order or don't belong to the current encryption level are kept by QUIC waiting to be processed, while handshake bytes that arrive in order are buffered directly by the TLS stack. From the moment in time in which the handshake is finished TLS has no more to send any kind of data unless an application or QUIC request it. For example it may happen that the server must provide updated session tickets to a client and in this situation the intervention of TLS is necessary.

Fig. 2.8 shows one possible structure of IETF QUIC handshake (the used implementation of the protocol and the order in which packets arrive may change the structure of the process). The client starts the handshake with a *Initial* message that contains the TLS *ClientHello* message. The server answer in its turn with an *Initial* packet, containing the *ServerHello*, and an *Handshake* packet containing the remaining packet sent by the server to the client during a TLS 1.3 handshake. At this point the client can complete the handshake with its *Handshake* packet. From this moment client and server can exchange data that will be protected with the cryptographic material established by the two peers during the handshake.

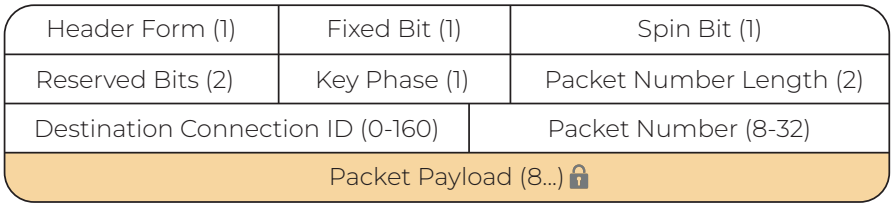


Figure 2.9. IETF QUIC Long Header packet

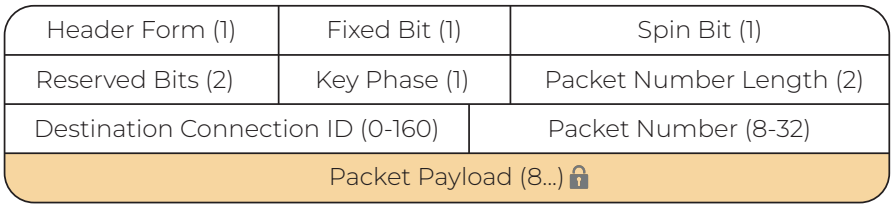


Figure 2.10. IETF QUIC Short Header packet

2.5.4 Session Resumption and 0-RTT Data

As well as TLS 1.3, also QUIC protocol includes the possibility to resume a previous session, carrying *NewSessionTicket* messages in CRYPTO frames after the handshake is finished. In order to implement session resumption, the endpoints must remember some information about the previous session. To this purpose the client can store the necessary information along with the session ticket, while the server can use the session ticket to help carry state.

A privacy issue may arise with session resumption because the resumed connection can be linked with the original one. Client can choose to disable session resumption to avoid this problem. To avoid connections correlation by other entities, session tickets should not be reused.

Session resumption in QUIC introduces the ability to send application data before the end of the handshake, since cryptographic parameters have been negotiated during previous connection. In order to send 0-RTT data, the client must provide the server with a valid session tickets, so that the server is can recover important and necessary information about the TLS state, QUIC transport parameters and the chosen application protocol.

2.5.5 Packet Format

Another novelty introduced by IETF involves the QUIC header, that was encrypted in a big part in the Google version of the protocol, except for some bits, including the packet number and key phase bits. The packet number in Google QUIC was used for both reliability and as a nonce (non-repeating value) for encrypting the packet. The IETF working group wanted to provide encryption also for the packet number, but this solution required another nonce. In order to avoid increasing the header overhead of the packet inserting the necessary additional nonce, they decide to use as random value the encrypted payload of the packet. The strategy to encrypt the header of the a QUIC packet became a two-step encryption process: first the packet is encrypted using the packet number as a nonce, then the packet number is encrypted using some of the encrypted packet as a nonce (and a different key).

Moreover IETF working group decided to define two types of header, long and short. The packets used during the handshake needed the long header, because they have to carry several

bits of information, while for packets sent after the connection establishment the short header, small as four bytes, is enough, since only some data in the header are necessary. The long header, shown in Fig. 2.9, is composed by the following fields: *Header Form*, a bit set to 1 for long headers, *Fixed Bit*, a bit set to 0 for Negotiation packet, 1 otherwise, *Long Packet Type*, a group of two bits containing a packet type, *Type-Specific Bits*, whose semantics is defined by the packet type, *Version*, 32-bit field containing the QUIC version, *Destination Connection ID Length*, a field of 8 bit containing the length in bytes (smaller than 20 bytes) of the *Destination Connection ID* field that follows it, *Source Connection ID Length*, a field of 8 bit containing the length in bytes (smaller than 20 bytes) of the *Source Connection ID* field that follows it and *Type-Specific Payload*, that is the payload of the packet.

A 1-RTT packet uses a short packet header and is depicted in Fig. 2.10. It contains the following fields: *Header Form*, a bit set to 0 for the short header, *Fixed Bit* set to 1, *Spin Bit*, representing the latency spin bit, *Reserved Bits*, a group of two bits that must be 0 (otherwise a protocol violation happened) and is protected using header protection, *Key Phase*, bit included in the header protection that allows a recipient of a packet to identify the keys used to protect the packet, *Packet Number Length*, two bits protected within the header containing the length of the Packet Number field, *Destination Connection ID*, *Packet Number*, that is protected with header protection and occupies from 1 to 4 bytes and *Packet Payload*.

2.5.6 Packet Protection

After the TLS handshake is completed, QUIC protocol can derive the keys to use in the AEAD negotiated algorithm to protect the packets. This provides strong cryptographic protections to all the packets for confidentiality and integrity. The only exception is represented by *Initial* messages, for which QUIC uses AEAD_AES_128_GCM algorithm with keys derived from the Destination Connection ID field of the first *Initial* packet sent by the client. For these packets the confidentiality and integrity protection is not so strong as for all the others, since the key derivation process is simpler and less secure. As for TLS 1.3, the keys derived from the TLS handshake cryptographic negotiated material are computed from the combination of the TLS secrets with the HKDF function.

For what concerns QUIC packets, also part of the header is protected, such as th *Packet Number* field. In this case the keys used for protection are different form the keys used for packet protection. The packet header protection regards the *Packet Number* field and four or five least significant bits of the first byte, respectively for long header packets and short header ones. These bits include the reserved bits and the *Packet Number Length* field for both types of header and the *Key Phase* bit for short header packets.

Chapter 3

Related Works

In the last years several studies to measure and compare different protocols have been made. All this works differ from each other for the context in which the protocols are measured, the architecture and tools used to execute the experiments, the metrics selected for the measurements and for the results obtained. Given the importance and the spread of the three protocols involved in the thesis (TLS, DTLS and QUIC), they can be studied and compared in different scenarios and using several possible techniques. For example it's possible to make a comparison of them in an IoT scenario, in which devices with a reduced amount of resources in terms of energy, memory e computational capacity tries to establish a connection to a server. On the other hand also a desktop scenario is suitable for a comparison of the protocols, especially QUIC and TLS, that can be used through a browser, such as Google Chrome that provides support for both of them, to connect to a webpage. All the works analysed for this thesis are summarized in Tab 3.1.

HTTP over UDP: an Experimental Investigation of QUIC Since the birth of QUIC, a lot of works about the comparison with the TCP+TLS+HTTP stack have been made, especially for what concerns the performance of the two protocols. The first one of these studies was conducted by G. Carlucci et al. (2015) [8] and this work consists of some experiments on QUIC to compare its performance with SPDY and HTTP/1.1 (over TLS) in terms of Page Load Time, defined as the time taken by the browser to download and process all the objects associated with a web page. In this case the experiments are conducted in a desktop scenario, in which the client, implemented through the Chromium Browser M39, connects to the server machine, on which a server application for each protocol under test is installed, and loads the pages available using both protocols. There are three kind of different pages to download, distinguished by the size intended as the number of images that the page contains. The measurements are performed with Chromium Browser developer console, that is able to compute the time elapsed from the request of the page to its full download. The performance are evaluated also varying the network conditions, such as link capacity and induced random losses, and this is possible thanks to a proprietary software, named *NetShaper*. This is an important point of the research because the results obtained vary depending on the network parameters setting. For each configuration of the variable network conditions ten experiments are performed and the result page load time is the average of all of them. This study revealed that when no additional losses are added to the channel QUIC has a lower page load time with respect to HTTP/1.1 and it demonstrates better performance than SPDY in case of a lossy channel . The performance of QUIC are reduced due to its Forward Error Correction module.

How quick is QUIC? This results are confirmed by a work of 2016 in which P. Megyesi et al. [9] compared in their study the performance of QUIC, HTTPS and SPDY and found that there is no a protocol better that the others regardless of network conditions. Also in this case the measurements are executed in a desktop scenario in which the client is represented by the Google Chrome browser and to automate the process of measurements a tool named *Chrome HAR capturer* [10] is used. This tool gives the possibility during a connection performed through the browser to generate a HAR (HTTP ARchive) log file, that contains

all the information needed to compute the page load time (that is again the metric selected to compare the performance of the two protocols). Differently from the previous work, in this study on the server side different sized pages are installed on Google Sites, a website-building platform from Google. As for the other work the performance are measured in different network conditions that are obtained through a shaper server installed between the client and the Google Sites server, that manipulates bandwidth, packet loss and delay of the connection using the Netem function of the Traffic Control package of Linux. For each combination of network parameters and page size (48 in total) at least a hundred of measurements are performed. The results of this study claim that QUIC downloads as fast as HTTPS from websites with small and medium size pages in presence of good network conditions (high bandwidth, low RTT and low packet loss), while the only situation in which QUIC demonstrates a poor performance is in presence of very high speed links and large sized pages. The presence of additional loss in the network has a lower impact on QUIC with respect to the other protocols, thanks thank to the FEC module.

Does QUIC make the Web faster? In the following years also other researchers obtained results that are similar to the ones got in the previous works. For example P. Biswal and O. Gnawli (2016) [11] conducted several experiments to compare the performance of QUIC and HTTP/2 in terms of page load time. They set up two different testbeds, one for controlled and the other one for uncontrolled experiments, in which the network conditions can't be manipulated but were the one available in the campus. In both the environments the client is a machine equipped with Chromium Browser and an extension developed to measure the time taken to load the page (including only resource loading and discarding script execution time). For controlled experiments the server is a machine connected to the client through a router, while in uncontrolled environment the server is a machine publicly available and the client can connect to it through both a wired and a cellular network. As in the previous work, during controlled experiment Netem function is used to manipulate network conditions. When a connection to the server is performed different kinds of pages can be downloaded, depending on the number and the dimension of objects contained in the page. Also in this case the network conditions influence a lot the performance, indeed QUIC has good performance in situation with low bandwidth, high delay and lossy links. QUIC is also faster than HTTP/2 when the page to download contains few small-sized objects. The only case in which HTTP/2 is clearly better than QUIC is when the target is a page with many objects of small dimension.

Taking a Long Look at QUIC It's necessary to consider also the work by A. M. Kakhki et al. (2017) [12], in which the authors go into a deep analysis of QUIC, in particular evaluating the performance of different version of the protocol in different environments for the experiments, that include desktop and mobile scenarios with wired and wireless networks. The testbed consists of servers running on Amazon EC2 and supporting HTTP/2 over TCP (and TLS) and QUIC, and three different client devices: a desktop a two mobile phones. All these devices run two applications, Google Chrome and Youtube, to test protocol performance. For what concerns Chrome, they use the debugging tool of the browser to evaluate the page load time while the client tries to download a static HTML page that references some images. Regarding video streaming performance, they measure QoE metrics through the Youtube iFrame such as time to start, buffering time and number of rebuffers. For each protocol and for each configuration, identified as client, server and network conditions, they execute at least 10 measurements and then compute the average of all the values obtained. For what concerns desktop scenario the results confirm what has been revealed by previous research: QUIC outperforms TCP+HTTPS in almost every scenario, especially in poor network conditions, with the exception of the case of page to download with high number of small objects. QUIC has also worse performance in presence of packet reordering, because the protocol interprets this situation as if losses are happening and starts to send packets more slowly. The same happens in case of a mobile device as a client, indeed QUIC behaves better than TCP in most cases. A clear advantage in terms of performance of QUIC can be encountered also in a video streaming scenario, but only in presence of high-resolution video.

QUIC: Better For What And For Whom? The last study that took up the measurements methodology and obtained similar results to the previous works is that of S. Cook et al. (2017) [13], in which the main goal was to identify in which conditions and for which actors QUIC is of considerable importance, comparing its performance to HTTP/1.1 and HTTP/2 (over TCP+TLS). They set up two different testbeds for their tests: a local one, to execute experiments in a controlled environment, and a remote one, to make tests on the real Internet. For the first kind of experiments QUIC and HTTP web servers are deployed and installed on the same machine using Docker. The content of each web server is a replica of one of some public websites (such as Youtube and Orange). The client is a machine equipped with *Perfy*, a proprietary tool that gives the possibility to script and automate tests. Indeed *Perfy* is able to launch the browser (Google Chrome), accessing a requested page, and compute the performance metrics (Page Load Time), based on the Navigation Timing API, that is integrated in the browser and gives information about the process of page loading. Another feature of this tool is that during a connection to a website it captures the traffic, that can subsequently be analysed. For the remote testbed the client side is the same, while on the server side there are the public Youtube and Orange real websites. They have decided to test the protocols both on first-ever and repeated connections and for each protocol each test consists of 100 iterations to obtain significant results. As the vast majority of the other works on the comparison between QUIC and HTTP (over TCP+TLS), they found that loss and delay have a lower impact on QUIC with respect to HTTP/2, that is outperformed by the protocol of Google in presence of lossy links or in a unstable network, such as a wireless mobile network.

QUIC and TCP: A Performance Evaluation Even if it seems that QUIC is better than TCP+TLS in many scenarios and under a lot of different network conditions, there have been also some works that state the contrary. In particular K. Nepomuceno et al. (2018) [14] worked to measure the performance of QUIC and HTTP and noticed unexpectedly that the second protocol behaved better than the first. They choose 100 pages between the ones proposed by Alexa site ranking and, thanks to the use of the tool *Mahimahi* [15], they are able to record and replay the selected websites emulating also different network conditions. Then using the Chrome browser as a client they try to connect to these replicated webpages and to compute the page load time (PLT) as a metric to evaluate the performance. To do so they capture an HAR log file for each experiment, that involves a webpage and a combination of network parameters (RTT and packet loss ratio) manipulated through *Mahimahi*. Each combination is tested 30 times and then the average PLT was computed. They find that TCP performs better than QUIC on at least 60% of the pages, considering all the possible network conditions. Also in this case the configuration of network parameters influences a lot the collected results, as well as the choice to enable caching in the browser, in fact setting properly the environment (made of network conditions and cache) the number of pages where QUIC outperforms TCP can go from 5.37% up to 40.62%.

A Performance Perspective on Web Optimized Protocol Stacks: TCP+TLS+HTTP/2 vs. QUIC Another important study to cite is the one of K. Wolsing et al. (2019) [16], that decided to use a completely different approach to analyse QUIC protocol and the stack composed by TCP, TLS and HTTP/2. First of all they tune TCP to match as much as possible QUIC's default, for example increasing the initial congestion window and enabling pacing, in order to provide a fairer comparison between the two protocols with respect to the ones proposed by the previous approaches. Secondly they decide to evaluate and measure visual web performance metrics instead of simply computing page load time, because they are more related with human perception. Indeed they choose metrics derived from video recording such as First Visual Change (FVC), Last Visual Change (LVC), and time the website reaches visual completeness of a desired threshold in percent (85% is the value chosen in this work). They create a measurement environment in which at the server side there are replicated websites with *Mahimahi* tool [15] and the QUIC (version 43) and HTTPS servers were generated respectively with Chromium sources and NGINX. On the other hand the client is a machine equipped with the *Browsertime* tool [17], that is used for each run of the experiments to open a fresh Chromium browser and record video while the loading process is evolving. All the videos are then used to evaluate all the visual metrics. During

their experiments QUIC performance results to be better with respect to TCP on various scenarios, but TCP parameter tuning gives the possibility to reduce the gap between the two protocols. QUIC works well especially in lossy network and it's due to design features, such as fast connection establishment and solution of the head-of-line blocking problem.

Evaluating QUIC Performance over Web, Cloud Storage and Video Workloads The most recent work that evaluated QUIC performance, comparing it with TCP+TLS, was proposed by T. Shreedhar et al. in 2022 [18] and introduced an approach to measure protocol performance different from the past. They conducted experiments from 2018 to 2021 using several tests written in C language for active measurements. They decide to test the two protocols in four different fields: web workloads, cloud storage workloads, video download and video streaming. To this purpose they develop three groups of tests written in C: *tls_perf* and *quic_perf* to measure respectively TLS (both 1.2 and 1.3 version) and QUIC (both in Google and IETF version) both in web and cloud storage workloads, and *video* tests to measure performance of the protocols in video download and video streaming. For what concerns QUIC management they use *lsquic* [19], an open source QUIC library developed by LiteSpeed that provides support for QUIC, while to set up connection with TLS they use *libcurl* library. Regarding web workloads they measure connection times, TTFB (Time To First Byte) and download times while the client (a VM or a Raspberry Pi) tries to connect to Alexa Top-1M websites (they discovered that only 5.7K websites supported QUIC). During this tests they can notice that QUIC in lossy networks has a clearly lower mean connection time than TLS 1.2/1.3 over TCP. In addition IETF QUIC has a lower handshake time also when it is compared to several version of Google QUIC. To evaluate the performance of the protocols in cloud storage workloads, they use them to download different sized files (from 1KB to 2GB) from Google Drive and measured throughput and CPU utilization. The comparison in this case is only between QUIC and TLS 1.2 because Google Drive doesn't support TLS 1.3. For what concerns cloud storage workloads, when considering files of large size (¿20 MB up to 2 GB) TLS 1.2 over TCP results in higher throughput with respect to QUIC, while it performs better when considering smaller file sizes (¡20 MB). The explanation can be found in the fact that the total download time is dominated by the handshake time and the TTFB. In terms of resource utilization, QUIC has much higher CPU utilization than TLS 1.2, especially when considering large files, due to in-kernel optimizations. Regarding video workloads, TLS 1.2 results in an higher overall download rate than QUIC and Google QUIC version 35 is the worst in terms of download rate. For video streaming performance, TLS/TCP presents a larger startup delay with respect to QUIC, and the gap in the performance increases in a lossy network. Despite lower overall download rate, QUIC has a better video content delivery with reduced stall events and lower stall durations due to its reduced latency overheads and better loss recovery mechanism.

TLS and Energy Consumption On a Mobile Device: A Measurement Study Most of the works previously reviewed, about the comparison between QUIC and other protocol stacks, showed an implementation in a desktop scenario, in which a client machine tries to connect through some tools and application to a server. There have been also in the past years, studies that tried to measure performance of a protocol in a different situation, such as IoT and mobile phone fields. It's the case of the work set up by P. Miranda et al. in 2011 [20], in which they made experiments to measure the energy consumption of TLS on a mobile device. They point out that the energy consumed during an entire TLS transaction it's not due only to the cryptographic component of the protocol, but has also a non-cryptographic part. All the asymmetric, symmetric and hashing operations concur to compose the cryptographic component of the energy consumption, while all the energy consumed for data transfer represents the non-cryptographic part. TLS contributes to increase also the non-crypto component with respect to the TCP only case, because of its messages that include additional items, such as certificates and message digests. The testbed is composed by a Nokia N95 phone, that acts as a client, thanks to a program developed with OpenSSL library [27], and connects to a public server through a dedicated WLAN access point or 3G access network. The energy is measured with Nokia Energy Profiler application, that is triggered directly by the client application when the TLS transaction starts, in order

Title	Year	TLS	DTLS	QUIC	Perf.	Sec.	Ref.
HTTP over UDP: an Experimental Investigation of QUIC	2015	x		x	x		[8]
How quick is QUIC?	2016	x		x	x		[9]
Does QUIC make the Web faster?	2016	x		x	x		[11]
Taking a Long Look at QUIC	2017	x		x	x		[12]
QUIC: Better For What And For Whom?	2017	x		x	x		[13]
QUIC and TCP: A Performance Evaluation	2018	x		x	x		[14]
A Performance Perspective on Web Optimized Protocol Stacks: TCP+TLS+HTTP/2 vs. QUIC	2019	x		x	x		[16]
Evaluating QUIC Performance over Web, Cloud Storage and Video Workloads	2022	x		x	x		[18]
TLS and Energy Consumption On a Mobile Device: A Measurement Study	2011	x			x		[20]
On TLS 1.3 Early Performance Analysis in the IoT Field	2016	x			x		[21]
IoTLS: Understanding TLS Usage in Consumer IoT Devices	2021	x			x		[22]
DTLS Performance in Duty-Cycled Networks	2015		x		x		[23]
DTLS Performance - How Expensive is Security?	2019		x		x		[24]
Low-Power IoT Communication Security: On the Performance of DTLS and TLS 1.3	2020	x	x		x		[25]
Which Secure Transport Protocol for a Reliable HTTP/2-based Web Service : TLS or QUIC?	2017	x		x		x	[26]

Table 3.1. Studied research papers.

to obtain accurate results. For the same reason, each experiments is repeated many times. They observe that the use of a ciphersuite that includes Diffie-Hellman increases significantly energy consumption. The main reason is that a transaction with DH is computationally much more intensive and lasts longer with respect to transactions with other ciphersuites, and so the network interface consume much more energy. If the client is connected to the server through the 3G network, access consumes from two up to four times the energy consumed over WLAN. The energy consumption can be reduced thanks to the use of session resumption. In addition to the energy measurements, they install a server locally and use the client to upload files of different size on it using both TLS and TCP and then compare the obtained results, in order to measure the communication data overhead of TLS. The results obtained show that the bigger is the amount of data to transfer, the more the energy overhead decreases, because the handshake energy is amortized by the energy used by the

record protocol to transfer data. The use of WLAN instead of 3G has a good impact in this evaluation, leading to a more important decrease of energy overhead. Moreover they claim that the choice of symmetric crypto and hash algorithms it's not important in order to reduce the energy consumption, because the overhead related to encryption and integrity protection is quite low. The only situation that differs from that is when the transmission rate is very high and a bottleneck due to a bad choice in terms of algorithms must be avoided.

On TLS 1.3 Early Performance Analysis in the IoT Field There are two other papers that reviewed works in which the performance of TLS protocol are tested in the IoT field. The first study to cite is the one of 2016 proposed by S. Bossi et al. [21], in which they made several tests on TLS 1.3 comparing the use of all the different ciphersuites and the one-way and mutual authentication in terms of performance. They set up a tesbed composed by a STM32F217IG1 device connected to the server machine through an Ethernet cable. The client application that run on the IoT device is realized with the wolfSSL library [28] version 3.4.6, while for the application of the server the OpenSSL library 3 [27] version 1.0.2a is used. They measure on the client the number if clock cycles needed to complete an handshake and the memory usage as metrics for the performance evaluation. The results reported in the paper show that in case of one-way authentication elliptic curve algorithms are faster that Diffie-Hellman ones and if an ephemeral version of these algorithms is use the key exchange process costs more clock cycles and reaches a level comparable to RSA private key operations. Diffie-Hellman solutions demonstrate also a greater memory usage with respect to elliptic curve solutions. In case of mutual authentication the benefit of an elliptic curve solution is even clearer. Elliptic curve solutions result in less memory requirements and higher security thanks to their perfect forward secrecy property with respect to RSA solutions. They also observe that the complexity and the burden for a IoT device to support a TLS 1.3 handshake that is characterized by the encryption of two third of the messages are balanced by the preference for AEAD ciphers and the introduction of new signature algorithms, such as Curve25519 and Ed25519.

IoTLS: Understanding TLS Usage in Consumer IoT Devices Another study regarding TLS and the IoT field is proposed by M. T. Paracha et al in 2021 [22]. In this work they wanted to evaluate the effective use by consumer IoT devices of the security protocol to protect the communication, the presence of correct certificate validation and the changes of the observed behaviour over time (indeed they collected data from their IoT devices for more that two years). They execute passive and active experiments inside a testbed composed of 40 TLS-supporting IoT devices, they can interact with through the companion apps. In order to capture the traffic generated by the devices, especially regarding TLS, they use the gateway installed in the testbed to provide network access to all the devices. During passive experiments they record the network traffic automatically generated by the device, that happens in particular when a device is not in use or when there is an interaction with one of the students enrolled in the laboratory to conduct the experiments. On the other hand in active experiments they intercept the traffic from devices while impersonating to be the server in a TLS connections (through mitmproxy). To this purpose, they use smart plugs to turn devices off and on repeatedly on command, based on the fact that a lot of the traffic is generated by a device when it's powered on. The obtained results reveal that 28 of the 40 devices use TLS 1.2 exclusively, while for the others they find in some cases the use of deprecated TLS versions over time. Regarding certificate validation they identify 7 devices that do not perform any certificate validation and other 4 devices that do not check for correct owner of the certificate indicated in the Common Name field. For these reason all these devices are vulnerable to different kind of attacks, that exploit these vulnerabilities.

DTLS Performance in Duty-Cycled Networks In addition to QUIC and TLS, also DTLS has been studied in terms of performance during the last years, especially in an IoT scenario, which is the field of greatest usage of this protocol. M. Vucinic et al. in 2015 [23] worked to a study on the performance of DTLS protocol in duty-cycled networks. They measure DTLS handshake time and energy consumption as metrics for performance evaluation on three different duty cycling protocols: preamble-sampling, the IEEE 802.15.4 beacon-enabled mode

and the IEEE 802.15.4e Time Slotted Channel Hopping mode. In their testbed they emulate WiSMote, a popular IoT platform, running a client application written using *tinyDTLS* library [29] for which the support is provided also by the Contiki operating system. To estimate the energy consumption they used the *Energest* module of Contiki OS. The DTLS handshake between client and server is repeated 1000 time and then they compute the average of all the values to obtain the final results. During their tests DTLS demonstrates poor performance when employed in radio duty-cycled networks, with an handshake process that can take till several tens of seconds and increases the duration with the number of hops in the network. In addition the reduced amount of resources available to the IoT client forces the node to maintain a maximum of 5 simultaneous DTLS sessions. For all these reasons they claim that DTLS can be used in applications in which the handshake is performed only a limited number of times.

DTLS Performance - How Expensive is Security? S. Gallenmüller et al. (2019) [24] constructed a model to evaluate the performance of applications that are based on DTLS. This model is composed by four main building blocks that are related to the most important costs in a DTLS application: packet IO, cryptographic hashing, state handling, and cryptographic processing. To this purpose, they create a DTLS VPN architecture, called *MoonSec* [30], to validate the model using a different library for each one of the building blocks: *DPDK* is dedicated to packet IO, *DenseMap* is used for connection state tracking, *OpenSSL* [27] handles encryption and decryption and *SipHash-2-4* manages 5-tuple hashing packets. *MoonSec* is a modular and flexible architecture, that provides has main features RFC-compliance and high performance. In their testbed they install two machines, one running *MoonSec* and the other one simply acting as a load generator. They execute black box measurements, in which they considered the machine running MoonSec as a black box and measure only the throughput, and white box measurements, evaluating the cost of packet reception and packet transfer, hashing, state table handling and packet processing. All the measurements are done computing the clock cycles necessary to execute the operation. Regarding the black box tests, they measure that the throughput grows linearly with the traffic till 950 Mbit/s and after that some packets start to be dropped. During white box measurements they introduce as input parameters three values: number of connections, number of bytes transmitted and number of packets transmitted. According to their results, the packet reception results in an average cost of 77 CPU cycle, while the packet transfer needs 66 CPU cycles. The tests on hashing operation instead produce an average cost in terms of CPU cycles equal to 62 for each packet, after an initial phase of higher values. Since the state table handling is divided into two main operations, insertion and lookup, they generate an equation to compute the clock cycles needed by each one of the two portions of this metric. The insertion results in a constant cost for each connection, equal to 400 CPU cycles, and a dynamic part, while the lookup is calculated as a static cost for each packet equal to 118. For what concerns packet processing measurements, they evaluate the costs in CPU cycles for performing and entire handshake, comparing the value obtained using different ciphersuites. They find that there is a significant gap between the cost the most expensive and the least expensive encryption algorithms they have tested, in particular DHE ciphers have a costs that are four times higher than elliptic curve based DHE ciphers.

Low-Power IoT Communication Security: On the Performance of DTLS and TLS 1.3 In 2020 G. Restuccia et al. [25] published the results of a work in which they evaluated the performance of TLS 1.3 and DTLS 1.3 in a IoT environment. To this purpose they provide the first implementation of DTLS 1.3 for systems with a constrained amount of resources. They execute tests with TLS and DTLS protocols on microcontrollers, measuring as performance metrics bytes over-the-air value, memory requirements and energy consumption. They set up a testbed with three IoT devices that run different implementation of TLS and DTLS protocol stack: they use the embedded solutions provided by WolfSSL [28] and MbedTLS [31], in addition to their own implementation. The obtained results reveal that the memory requirements remain roughly constant for both protocols between the latest version (version 1.3) and the previous one (version 1.2). DTLS 1.3 demonstrates a lower bytes over-the-air overhead with respect to DTLS 1.2 thanks to its record layer optimizations, while for TLS 1.3 the tests report a little increase in this metric compared to the previous

version of the same protocol. They notice an important difference in energy consumption between symmetric and asymmetric cryptography and TLS 1.3 results in a reduction of this value when PSK handshake is employed if compared to all the other possible handshakes of both protocols.

Which Secure Transport Protocol for a Reliable HTTP/2-based Web Service : TLS or QUIC? The protocol object of the thesis can be compared from the security point of view as well as from the performance one. The paper written by A. Saverimoutou et al. (2017) [26] contains a security analysis of TLS and QUIC protocols. They focus their attention on identifying the vulnerabilities of the two protocols and possible attacks against them, considering the possible impact web services based on HTTP/2. Each identified attack is related with the corresponding vulnerability and its feasibility and complexity are described as well as the drawbacks regarding the end-user's QoE (Quality of Experience) and QoS (Quality of Service) provided by the service. The TLS vulnerabilities considered are slow read attacks, Logjam, Lucky Thirteen attack, Broken RC4 as primary ciphersuite, while regarding QUIC they evaluate Replay attacks, Packet Manipulation attacks, Crypto Stream Offset. The results of the analysis show that an attack against a QUIC server mainly impacts end-users, but cannot degrade to much the service offered or make the server crash, and so QUIC is a more reliable solution to create a server than TLS. On the other hand the attacks against TLS typically have the purpose of making a DDOS and so have a huge impact on the server itself.

Chapter 4

Performance Evaluation

Since there are a lot of possible solutions to protect a communication channel between two peers it's important to evaluate strengths and weaknesses of each one of the protocols that offer this kind of service. Three of the most used solutions nowadays are the protocols analysed in this thesis: TLS and DTLS are specifically designed to ensure authentication, confidentiality and integrity to a communication, while QUIC is designed as a transport protocol that integrates also security features to protect the channel and the data exchanged between the two endpoints. These protocols will be measured and analysed from the performance point of view in order to determine which is the best one in different situations. The purpose is to evaluate which one has the best performance and if it happens at the expense of some security features. There are several possible ways to measure and evaluate a protocol and in this chapter some of these strategies (introduced also by other researchers) will be tested.

4.1 QUIC and TLS tests with the browser

As can be noticed analysing the works presented in the previous chapter, there are different ways to compare protocols, in particular the choice of the metrics that are used to evaluate the performance is a fundamental part when test have to be executed. Most of the works regarding the comparison between TCP+TLS+HTTP stack and QUIC used an approach in which the client, that is represented by the browser, tries to download a page from the server and the page load time (PLT) is the metric measured in each experiment. PLT is intended as the time taken by the browser to download and process all the objects that compose a web page. Consequently the first tests performed in this thesis are focused on measuring with the browser the time needed to download the page of three of the most accessed websites worldwide using both QUIC and TCP+TLS protocol.

4.1.1 Google Chrome

Google Chrome[1] is a web browser developed by Google based on the Chromium browser. The browser has grown rapidly over the years to become one of the most used worldwide. It supports both TLS (up to version 1.3) and QUIC protocols. Chrome added support for QUIC v1 (RFC 9000) in Chrome 90 and default-enabled it for all users in Chrome 93. For what concerns the QUIC protocol, Google Chrome uses a library named *quiche*[2] in order to implement it. It's a library created by Google that contains implementations of QUIC, HTTP/2 and HTTP/3 protocols. On the other hand, to manage TLS connections Chrome uses *BoringSSL* library, that is a fork of OpenSSL designed to meet Google's needs.

[1] https://www.google.com/intl/it_it/chrome/

[2] https://github.com/google/quiche

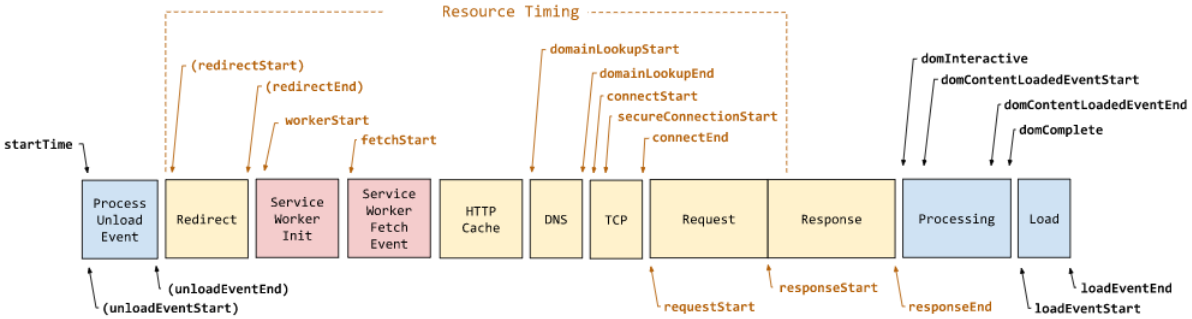


Figure 4.1. Timeline of the events recorded by the Navigation Timing API (source: W3C)

It's very easy to install Google Chrome on Linux. You just have to go to the Chrome website and download the DEB file. Then, double-clicking on the file just downloaded and then pressing on the "Install" button, it's possible to install easily Google Chrome browser with Software Install.

4.1.2 Navigation Timing API

The Navigation Timing API[3] is an interface for web applications to access the complete timing information for navigation of a document and can be used to measure performance of a web application. It should be preferred to JavaScript-based mechanisms because in many cases they are unable to provide a complete or detailed end-to-end latency picture. Instead the Navigation Timing API provides high resolution performance metric data related to the navigation of a document. All time values are measured with respect to the time origin, that is represented by the moment in time when the navigation started. The following list contains some of the values that can be measured through this API.

navigationStart This attribute must return the time immediately after the user agent finishes prompting to unload the previous document. If there is no previous document, this attribute must return the time the current document is created.

domainLookupStart This attribute must return the time immediately before the user agent starts the domain name lookup for the current document.

domainLookupEnd This attribute must return the time immediately after the user agent finishes the domain name lookup for the current document.

connectStart This attribute must return the time immediately before the user agent start establishing the connection to the server to retrieve the document.

connectEnd This attribute must return the time immediately after the user agent finishes establishing the connection to the server to retrieve the current document.

secureConnectionStart This attribute is optional. When this attribute is available, if https has been used, this attribute must return the time immediately before the user agent starts the handshake process to secure the current connection. If this attribute is available but HTTPS is not used, this attribute must return zero.

responseStart This attribute must return the time immediately after the user agent receives the first byte of the response from the server, or from the HTTP cache or from local resources.

[3] https://www.w3.org/TR/navigation-timing-2/

responseEnd This attribute must return the time immediately after the user agent receives the last byte of the current document or immediately before the transport connection is closed, whichever comes first. The document here can be received either from the server, the HTTP cache or from local resources.

loadEventStart This attribute must return the time immediately before the load event of the current document is fired. It must return zero when the load event is not fired yet.

loadEventEnd This attribute must return the time when the load event of the current document is completed. It must return zero when the load event is not fired or is not completed.

Among all the values that can be measured through the Navigation Timing API those just listed are the most relevant for measuring the performance of the protocol used to download the page. Fig. 4.3 represents a timeline containing all the parameters that can be measured using the API from the start of the navigation till the moment when all the elements of a web page are completely loaded in the browser window.

4.1.3 Page Download Time and Time To First Byte measurements

All the experiments are executed on a virtual machine realized with VirtualBox, running Ubuntu 20.04 as operating system, connected to a home router via Wi-Fi that provides Internet connectivity. The client is the Chrome browser, while the server in each test is the one of the real websites. The websites chosen to conduct the experiments are *www.google.com*, *www.youtube.com* and *www.facebook.com*, since they are among the most accessed worlwide every day. As can be noticed from Table 4.1, these three pages are characterized by different number of objects and different size.

Website	*Page Size*
www.google.com	45,2 KB
www.youtube.com	66,7 KB
www.facebook.com	28,2 KB

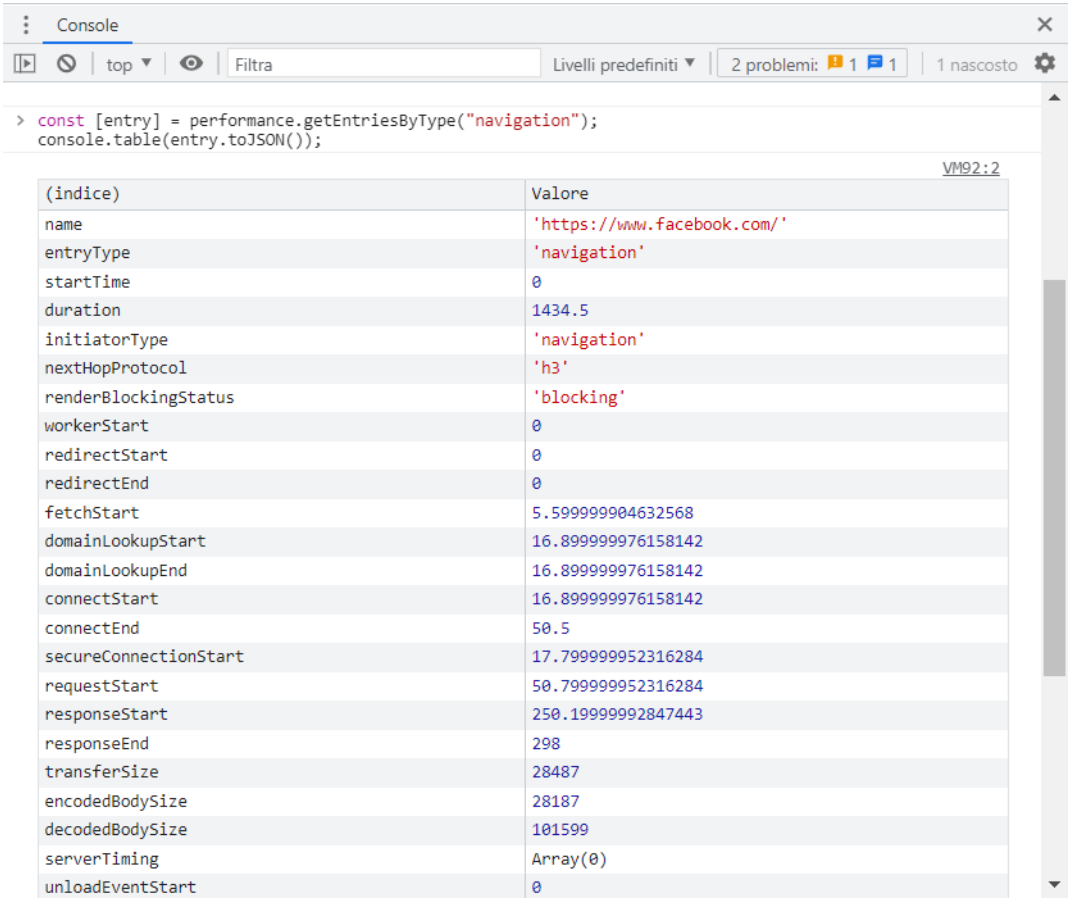
Table 4.1. Size of tested web pages

The servers that host all these pages support the creation of connections with both QUIC and TLS. All the timing information related to a connection to a webpage through the browser can be retrieved through the Navigation Timing API, that provides useful information to measure the performance of a connection to a website, using the Browser console. Once a web page is loaded in the browser, it's easy to get access to the console of the browser: one possibility is right-clicking on the screen and then pressing the "Inspect" button. Then in order to obtain all the values measured through the Navigation Timing API the following commands must be executed in the console:

```
$ const [entry] = performance.getEntriesByType("navigation");
$ console.table(entry.toJSON());
```

Fig. 4.2 shows an example of output obtained using the commands explained above, after a connection to Facebook webpage was performed.

The first metric measured in these experiments to compare the performance of QUIC and TLS protocols is the time needed to download the page. Even if a lot of authors decided to measure the Page Load Time (represented by the *loadEventEnd* value in the Navigation Timing API) in their work, this value includes also the time needed by the browser to process all the elements of the page and then load them. To avoid evaluating the performance of the browser and concentrate only on the protocol used to download the page the *Response End* value computed by the Navigation Timing API is a better choice, since it is defined as the time immediately after the browser receives the last byte of the current document or immediately before the transport connection is closed, whichever comes first. In addition, in order to exclude the time needed



(indice)	Valore
name	'https://www.facebook.com/'
entryType	'navigation'
startTime	0
duration	1434.5
initiatorType	'navigation'
nextHopProtocol	'h3'
renderBlockingStatus	'blocking'
workerStart	0
redirectStart	0
redirectEnd	0
fetchStart	5.599999904632568
domainLookupStart	16.899999976158142
domainLookupEnd	16.899999976158142
connectStart	16.899999976158142
connectEnd	50.5
secureConnectionStart	17.799999952316284
requestStart	50.799999952316284
responseStart	250.19999992847443
responseEnd	298
transferSize	28487
encodedBodySize	28187
decodedBodySize	101599
serverTiming	Array(0)
unloadEventStart	0

Figure 4.2. Screenshot of the output obtained in the browser console when Navigation Timing API parameters are retrieved

for the DNS translation, the page download time (PDT) evaluated during these experiment is defined as *responseEnd - domainLookupEnd*. The second value measured is the time to first byte (TTFB), defined as the time between the browser requesting a page and when it receives the first byte of information from the server and is computed as the difference of other two events measured thanks to the Navigation API, *responseStart - domainLookupEnd* (also in this case the time for the DNS translation is excluded from the computation).

During the experiments for each one of the three websites 20 tests have been executed, the first time using QUIC protocol and the second time using TCP+TLS. The switch between the two protocols has been made thanks to the Chrome flags, indeed *Experimental QUIC protocol* flag gives the possibility to enable (or disable) the use of QUIC by the browser when performing a connection to a website. In order to change the value of this flag in Google Chrome, you must enter *chrome://flags/* into the address bar and then select or search for the flag you want to enable. For each flag you can choose between three possible values: 'Enabled', 'Disabled' and 'Default'. Once you have selected the desired value, you must press on 'Relaunch' button, in order to restart Chrome and have the change effectively active. In addition, for all the experiments the cache of the browser has been disabled, so it was not possible to have resumed session but each time the client had to establish a new connection with the server. In order to disable the cache you have to right-click on the browser, select 'Inspect', move to the 'Network' tab and the check the 'Disable cache' box.

The results of the performance metrics (PDT and TTFB) related to each one of the 20 attempted connections can be retrieved are reported in two charts. As can be seen from Fig. 4.3, that contains the chart of the page download time metric evaluated during the experiments, QUIC protocol demonstrated a better performance in terms of time to download the page for Google and Facebook websites and the mean values obtained are respectively 169,5 ms and 191,8 ms.

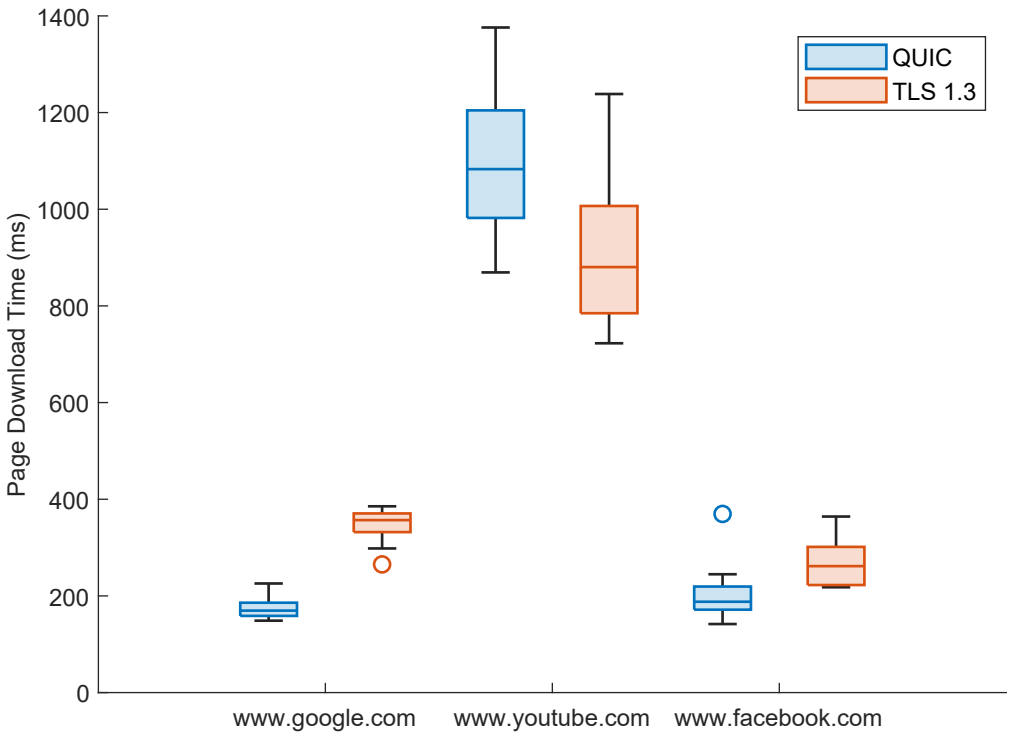


Figure 4.3. Boxplot of Page Download Time of three webpages downloaded with QUIC and TLS

Instead TLS was slower in downloading the two web pages, obtaining higher mean values for the page download time parameter: 340,9 ms and 259,2 ms respectively. On the contrary for Youtube website TLS provided a smaller mean download time equal to 905,6 ms with respect to QUIC that performed worse, with a mean value equal to 1084,9 ms. This results are confirmed by what can be read from the papers analysed in the previous chapter: QUIC is fast with small and medium size pages (as Google and Facebook), while performs poorly with large amount of data to download (indeed Youtube website is the biggest page considered, both in terms of size and number of elements).

The results of the evaluation of the TTFB are shown in Fig. 4.4. Also in this case QUIC performed better with respect to TLS when downloading Google and Facebook websites: QUIC showed mean values for the time to first byte respectively equal to 128,4 ms and 159 ms, while with TLS 241,9 ms and 215 ms were obtained on average. For what concerns Youtube website, the performance of the two protocols are similar and comparable. Indeed the mean values for the time to first byte obtained for QUIC and TLS are respectively 84,7 ms and 85,2 ms.

4.2 QUIC and TLS tests with lsquic and libcurl

TLS and QUIC protocols can be evaluated also using libraries for C language as well as with the browser, following what has been done in the paper of 2022 by T. Shreedhar et al. [18]. They have written two groups of tests, named *tls_perf* and *quic_perf*, in order to measure the performance of the two protocols in terms of connection time and time to first byte (TTFB) while connecting to websites belonging to Alexa Top-1M ranking. The connection time, defined as the time to set up a connection, for QUIC protocol corresponds to the handshake time, while for TLS includes TCP and TLS handshakes. Time to first byte (TTFB) is defined as the time taken from the client start

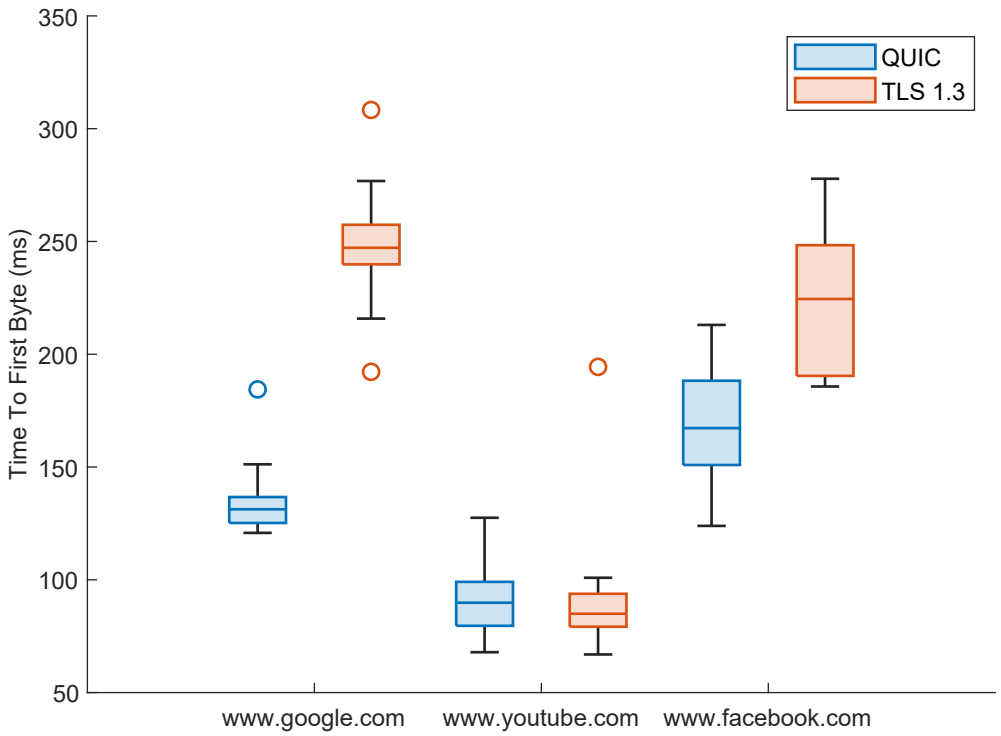


Figure 4.4. Boxplot of Time To First Byte of three webpages downloaded with QUIC and TLS

to the reception of the first byte of the server response, so it includes the set up of the connection (QUIC handshake or TCP+TLS handshake), sending of the HTTP request, server processing and reception of the HTTP response.

The tests for TLS protocol, written in C language, use libcurl[4] library which is necessary to configure and execute connections with different parameters such as URL, port and timeout as well as to compute and display the metrics requested for the performance evaluation. The libcurl library includes a function, *curl_easy_getinfo()* that can be called in order to retrieve some timing information related to a connection. In *tls_perf* the connection time, or handshake time, is measured by calling that method with *CURLINFO_APPCONNECT_TIME* option: this call gives as output the time elapsed from the client start until the TLS handshake with the server is completed. This time also contains the DNS lookup time that can be retrieved and then subtracted using the option *CURLINFO_NAMELOOKUP_TIME*, that measures the time from the client start until name resolution is completed. On the other hand the option *CURLINFO_STARTTRANSFER_TIME* provided by libcurl can be used to measure time to first byte for TCP/TLS. The time retrieved in this way includes the connection negotiation, the sending of the request and the time for the server to compute the result. As for the connection time, also in this case the DNS lookup time must be removed. An example of a C program that contains the use of all the options reported above with the function *curl_easy_getinfo()* of libcurl library is shown in Fig. 4.5

The second group of tests, *quic_perf*, written in C, includes the lsquic[5] library, an open-source QUIC client library for integration purposes that provides support for both Google QUIC versions (in particular currently number 43, 46 and 50 are supported) and IETF QUIC versions (currently

[4]https://curl.se/libcurl/

[5]https://github.com/litespeedtech/lsquic

```
#include <stdio.h>
#include <stdlib.h>
#include <curl/curl.h>

int main (int argc, char** argv)
{
    curl_global_init(CURL_GLOBAL_DEFAULT);
    CURL *curl;
    CURLcode res;
    double connect_dns, connect_tls, ttfb;

    curl = curl_easy_init();

    //initialization and setting of curl context

    res = curl_easy_perform(curl);
    if(res == CURLE_OK) {
        res = curl_easy_getinfo(curl, CURLINFO_NAMELOOKUP_TIME, &connect_dns);
        if(res == CURLE_OK) {
            res = curl_easy_getinfo(curl, CURLINFO_APPCONNECT_TIME,
                &connect_tls);
            if(res == CURLE_OK) {
                res = curl_easy_getinfo(curl, CURLINFO_STARTTRANSFER_TIME,
                    &ttfb);
            }
        }
    }

    return 0;
}
```

Figure 4.5. Example of use of CURLINFO_APPCONNECT_TIME and CURLINFO_STARTTRANSFER_TIME libcurl options to retrieve TLS handshake time and time to first byte

Draft-27, Draft-29 and Version 1). For QUIC the connection time is measured thanks to the use of the callback *on_hsk_done()* provided by lsquic library. This function is called every time a QUIC handshake is completed and can be configured to measure the time form the start to the end of the handshake and also to capture whether or not the connection attempt is successful. The DNS lookup time is not included in the measurement and so this solution is compatible with the definition provided by the authors of the paper. In order to measure the time to first byte for QUIC it's necessary to use a callback, named *on_read()*, that is used to process bytes received by the client after the handshake and is invoked for the first time when the first byte of the response from the server are received by the client. Also in this case the measurement includes the handshake time and the sending of the request as well as the server processing time and the reception of the response. In addition the DNS lookup time is again excluded from the computation of the result.

4.2.1 libcurl

Libcurl is a client-side library that support a lot of different protocols, among which HTTP and HTTPS, that can be used to evaluate TCP+TLS performance. This library is free and fast and can build and work on numerous different platforms. In addition it's thread-safe and compatible with IPv6.

If you want to use libcurl functions in your C program you will need to install curl on your system. To do so it's enough to download the latest curl package or one of the specific libcurl packages listed from the curl download page and follow the instructions below. You have first to unzip the file just downloaded with the command:

```
$ tar xvf curl-[version].tar.gz
```

Subsequently you have to enter the folder of the library and execute this sequence of commands:

```
$ ./configure --with-openssl [--with-gnutls --with-wolfssl]
$ make
$ make test (optional)
$ sudo make install
```

You need to adjust the configure line in order to select the correct TLS library that you want to use. The following command can be used to list all the possible options for the configuration:

```
$ ./configure --help
```

The library is provided with a C API that gives the possibility to use libcurl inside a C program.

4.2.2 lsquic

LiteSpeed QUIC (lsquic) Library is an open-source implementation of QUIC and HTTP/3 for servers and clients. This library currently supports different IETF QUIC versions (v1, Internet-Draft versions 29 and 27) and also some Google QUIC versions, such as version 43, 46 and 50. CMake, zlib, BoringSSL and go are prerequisites to build lsquic library.

If you want to build and install lsquic you have to start cloning BoringSSL with the following command:

```
$ git clone https://boringssl.googlesource.com/boringssl
$ cd boringssl
```

If you want to use a specific BoringSSL version use the following command:

```
$ git checkout [version]
```

Compile the library with optimizations:

```
$ cmake -DCMAKE_BUILD_TYPE=Release . && make
```

You need to remember where BoringSSL sources are for the following steps:

```
$ BORINGSSL=$PWD
```

After having built BoringSSL you can build and install LSQUIC following the instructions below. First you have to get the source code:

```
$ git clone https://github.com/litespeedtech/lsquic.git
$ cd lsquic
$ git submodule init
$ git submodule update
```

Then you can compile the library statically:

```
$ # $BORINGSSL is the top-level BoringSSL directory from the previous step
$ cmake -DBORINGSSL_DIR=$BORINGSSL .
$ make
```

```
lsquic_conn_t * lsquic_engine_connect(lsquic_engine_t *engine, enum
    lsquic_version version, const struct sockaddr *local_sa, const struct
    sockaddr *peer_sa, void *peer_ctx, lsquic_conn_ctx_t *conn_ctx, const
    char *sni, unsigned short base_plpmtu, const unsigned char *sess_resume,
    size_t sess_resume_len, const unsigned char *token, size_t token_sz)
```

Figure 4.6. Prototype of lsquic_engine_connect() function

4.2.3 Connection Time and Time To First Byte measurements

In order to reproduce the tests for the TLS protocol the code made available by the authors of the paper in a Github repository[6] is used. With respect to the code in the repository, two changes have been applied: the computation of the time to first byte and the total time to download the page must be supplied with the subtraction of the DNS lookup time, according to the definition given by the authors of the paper. On the other hand, for the measurements of QUIC protocol the code provided by the authors is not used and another solution is adopted: on the Github repository of lsquic library different programs for testing are available and between them there is an *http_client* that can be used to obtain the same result as *quic_perf* tests, indeed in addition to the creation and management of the connection, the program is responsible of printing statistics about the connection if requested, such as TTFB and handshake time. Also in this case two changes to the code must be applied in order to be compliant with the definitions and the experiments made by the authors of the paper. The way the time to first byte is computed must be modified, since in the *http_client* it is intended as the time from the handshake completion to the reception of the first byte of the response. To do so the starting point of the evaluation of the TTFB is changed with the moment in time in which the client is started (as it is for the handshake time). The second change that must be made concerns the insertion of the possibility to select the version of QUIC protocol that will be used by the client to attempt the connection with the server. In this way it will be possible to compare the behaviour of different versions of QUIC as done by the authors of the paper. This change has been implemented inserting an option in the *http_client* program that let the user to specify the version of the protocol, that will be then used as parameter in the function of the library named *lsquic_engine_connect()* that is called to start the handshake (with the version of QUIC specified as parameter) and establish the connection. As is shown in Fig. 4.6, this function receives as second parameter a variable of type *lsquic_version* that is an enum which can assume 8 different values that correspond to the QUIC versions supported by the library: LSQVER_043, LSQVER_046, LSQVER_050, LSQVER_ID27, LSQVER_ID29, LSQVER_I001, LSQVER_VERNEG and N_LSQVER. After these changes the *http_client* program can be used to reproduce *quic_perf* since it implements the same logic and gives as output the same results as the tests created by the authors of the paper. All the code developed and used to conduct these experiments, for both TLS and QUIC, can be compiled and executed following the guide in Appendix A, in the corresponding section of the user's manual.

These two groups of test are executed in the same environment used for the experiments reported in the previous section: a virtual machine running Ubuntu 20.04 connected to the Internet through a home router via Wi-Fi. Also the target websites chosen from the Alexa ranking that are used for the measurements are the same as before: www.google.com, www.youtube.com and www.facebook.com. Both the groups of test are executed with every version of the protocol available for each websites 50 times. With *tls_perf* it's possible to choose between TLS 1.2 and TLS 1.3, while QUIC tests give the possibility to switch between versions 43, 46, 50 (Google QUIC), Draft-27, Draft-29 and Version 1 (IETF QUIC) of the protocol. The output of both programs is recorded and the values obtained for the connection time and TTFB are used to generate charts. Fig. 4.7 shows the CDF of connection times for all the different versions of QUIC and TLS for each website. Table 4.2 shows the correspondence between the names in the legend

[6] https://github.com/RohitPanda/tls_perf

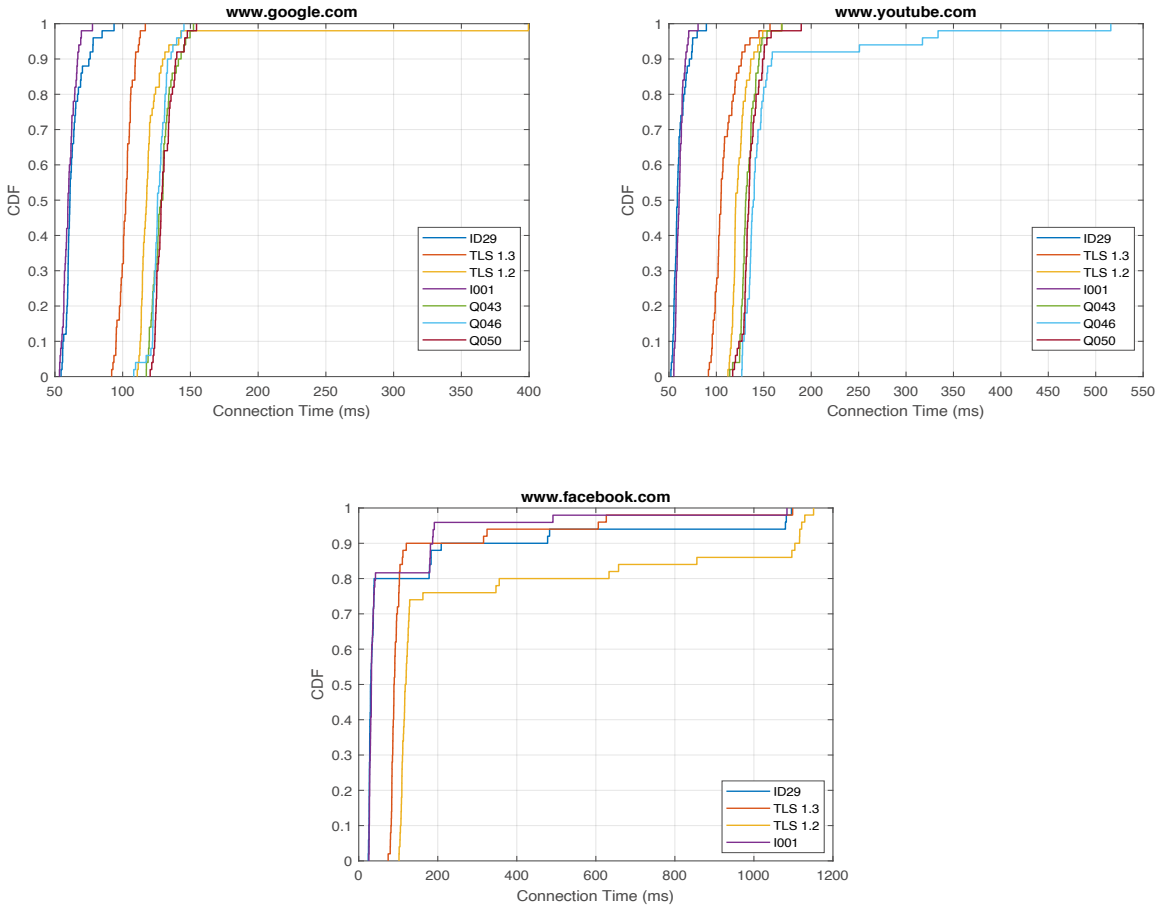


Figure 4.7. Cumulative Distribution Functions of Connection Time values obtained in experiments with three different sites

of the chart and the real QUIC version. It can be noticed that Facebook doesn't support the Google versions of QUIC protocol but only IETF ones (ID-29 and Version 1). However for all the protocols involved the behaviours for each websites are really similar. Indeed the latest versions of IETF QUIC (Draft-29 and Version 1) outperform all the Google versions of the same protocol and also TLS 1.2 and TLS 1.3 for each website as can be expected. The average value for the fastest versions of QUIC is around 60 ms. Surprisingly TLS 1.3 demonstrates better performance with respect to Google versions of QUIC (43, 46, 50), that in their turn have a comparable behaviour with respect to TLS 1.2. For TLS 1.3 the average value of the connection time is around 100 ms for all websites visited during the experiments, while for Google QUIC versions and TLS 1.2 the mean value grows till around 120-130 ms. Different enhancements have been provided to QUIC protocol in the transition between Google version and IETF version and this is the reason why a better performance of QUIC Version 1 with respect to versions 43, 46 and 50 was expected. Nevertheless QUIC was expected to be better in all its versions than TLS protocol but the test results state otherwise. A similar behaviour for what concerns time to first byte measurements can be noticed looking at Fig. 4.8 that shows the CDF of time to first byte values obtained executing the experiments on the three websites. Also in this case the latest versions of QUIC protocol (Draft-29 and Version 1) provide the best performance with an average value for TTFB of around 120-125 ms for Youtube website and around 150 ms for Google website. TLS 1.3 follows the fastest protocols with a mean TTFB almost 50 ms higher with respect to IETF QUIC for each website and outperforms TLS 1.2 and again surprisingly Google QUIC. These last two protocols increase their average TTFB with respect to TLS 1.3 of almost 15-20 ms. Since the TTFB as it was defined includes the handshake time, it's easy to understand that both the metrics evaluated

Legend name	*QUIC version*
Q043	Google QUIC 43
Q046	Google QUIC 46
Q050	Google QUIC 50
ID29	QUIC Internet Draft 29
I001	IETF QUIC v1

Table 4.2. Table containing the translation between the names in the legend of the charts and the QUIC protocol versions

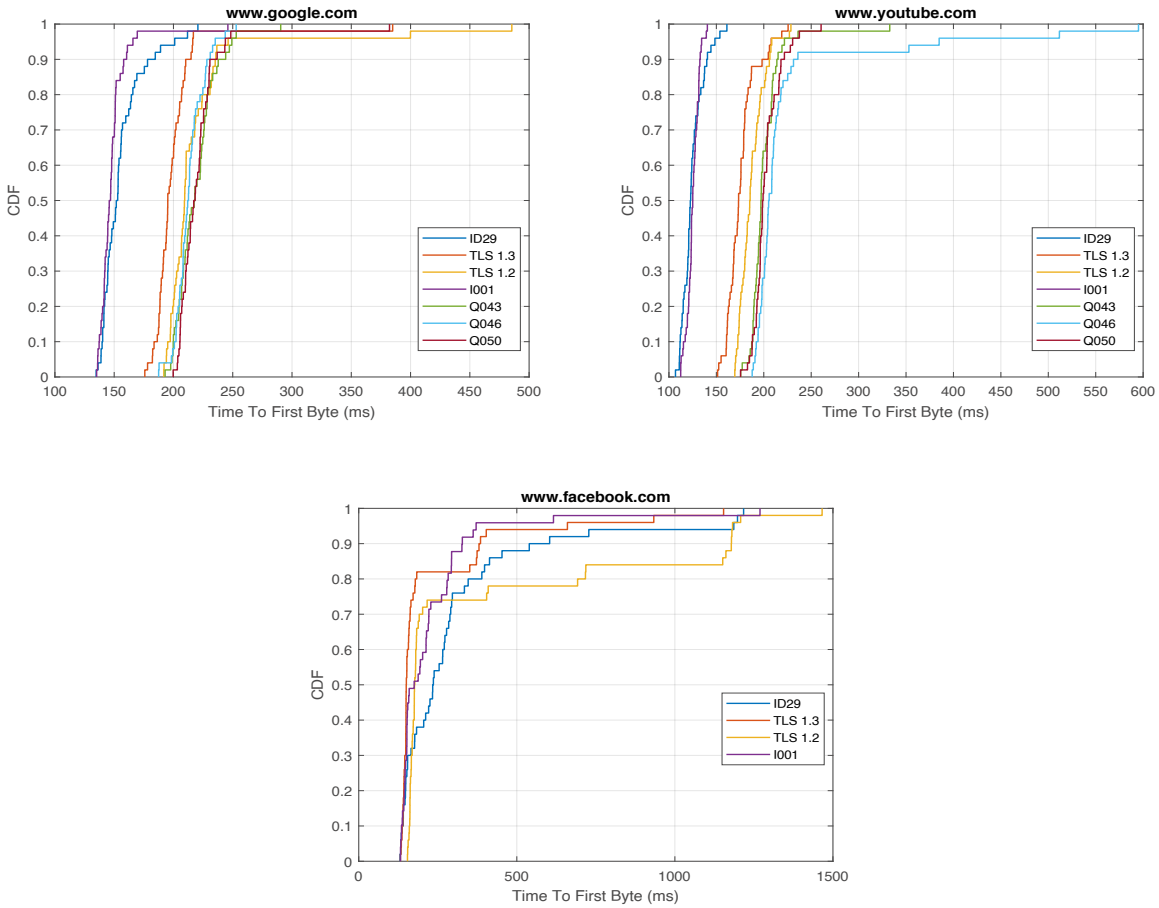


Figure 4.8. Cumulative Distribution Functions of Time To First Byte values obtained in experiments with three different sites

with the experiments maintain the same behaviour. Looking at the chart for Facebook website something unexpected can be noticed: all the protocols behave in a similar way and the advantage of using QUIC with respect to TLS is no more noticeable.

4.3 QUIC, TLS and DTLS tests with lsquic and OpenSSL

A feature that characterizes a protocol and differentiates it from the others is the handshake time, intended as the time needed to establish a connection between a client and a server. Each protocol studied in this thesis includes an handshake phase in which the client and the server exchange cryptographic material and agree on which algorithms and keys to use in order to protect the

channel between them. TLS is characterized by an handshake lasting 2-RTT for version 1.2 of the protocol and 1-RTT for version 1.3, but since this security protocol is used to protect a TCP channel, the time needed to establish a secure communication channel between a client and a server from scratch must be increased of 1-RTT, that is the time needed for the 3-way handshake of the TCP protocol. So in total 3-RTT are needed for having a TLS 1.2 channel, while 2-RTT for a TLS 1.3 one. QUIC instead is characterized by a 1-RTT handshake and no additional time is needed to create a secure channel since this protocol runs on top of UDP. The same consideration can be made for DTLS (version 1.2), but in this case the handshake process is similar in the number of messages and their content to TLS 1.2 handshake. The difference in the handshake with respect to TLS 1.2 is represented by the cookie exchange process, immediately after the *ClientHello* message, to prevent DOS attacks.

4.3.1 Mininet

Mininet[7] is a tool used to create a realistic virtual network composed of virtual hosts, switches and links interconnecting them on a single host. In addition Mininet uses NetEm, an enhancement of the Linux traffic control facilities, and for this reason it's possible to customize the network modifying various parameters of the links, such as delay, bandwidth and loss. Delay is emulated by queueing and holding back all outgoing packets for a fixed amount of time, while loss is emulated by randomly dropping packets with a certain probability. If you want to install Mininet, you just have to execute the following command:

```
$ sudo apt install mininet
```

If Mininet complains that Open vSwitch isn't working, make sure it is installed and running:

```
$ sudo apt install openvswitch-switch
$ sudo service openvswitch-switch start
```

4.3.2 OpenSSL

OpenSSL[8] is an open source toolkit for the Transport Layer Security (TLS) protocol. The protocol is implemented through a general purpose cryptographic library, which can also be used stand-alone. The OpenSSL toolkit includes:

- libssl, that is an implementation of all TLS protocol versions up to TLSv1.3;
- libcrypto, the general purpose cryptographic library;
- openssl the OpenSSL command line tool, that can be used for creation of key parameters, creation of X.509 certificates, CSRs and CRLs, calculation of message digests, encryption and decryption, SSL/TLS client and server tests and handling of S/MIME signed or encrypted mail.

In order to install OpenSSL you need first to get the source code. You can do that cloning the Github repository of the library with the following command:

```
$ git clone https://github.com/openssl/openssl.git
$ cd openssl
```

Then you can build and install the library executing the following commands:

```
$ ./Configure
$ make
$ sudo make install
```

[7] http://mininet.org/

[8] https://www.openssl.org/

The OpenSSL version that was built and used for all the experiments is 1.1.1f. If you execute the command indicated above, the library will be compiled automatically with the following command through the compiler:

```
$ gcc -fPIC -pthread -m64 -Wa,--noexecstack -Wall -Wa,--noexecstack -g -O2
    -fdebug-prefix-map=/build/openssl-51ig8V/openssl-1.1.1f=.
    -fstack-protector-strong -Wformat -Werror=format-security
    -DOPENSSL_TLS_SECURITY_LEVEL=2 -DOPENSSL_USE_NODELETE -DL_ENDIAN
    -DOPENSSL_PIC -DOPENSSL_CPUID_OBJ -DOPENSSL_IA32_SSE2
    -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_MONT5 -DOPENSSL_BN_ASM_GF2m
    -DSHA1_ASM -DSHA256_ASM -DSHA512_ASM -DKECCAK1600_ASM -DRC4_ASM
    -DMD5_ASM -DAESNI_ASM -DVPAES_ASM -DGHASH_ASM -DECP_NISTZ256_ASM
    -DX25519_ASM -DPOLY1305_ASM -DNDEBUG -Wdate-time -D_FORTIFY_SOURCE=2
```

4.3.3 Wireshark

Wireshark[9] is the most widely-used tool for network protocol analysis. This tool let the user inspect deeply hundred of protocols with live capture and offline analysis. Wireshark is a multi-platform tool and it's supported by a lot of different operating systems. With this tool you can record all the packets that go through a specific interface of your system. Wireshark is able to read and write many different capture file formats, among which one of the most used is tcpdump (libpcap). Executing the following two commands on Ubuntu it's very easy to install the latest version of Wireshark;

```
$ sudo apt update
$ sudo apt install wireshark
```

Once you have opened the tool, you can start capturing packets with Wireshark by simply double-clicking on the interface in the welcome screen that you want to monitor. Otherwise if you already know the name of the capture interface you can start Wireshark from the command line:

```
$ wireshark -i [interface_name] -k
```

4.3.4 Handshake measurements

The experiments have been conducted in an environment created with Mininet on a virtual machine running Ubuntu 20.04. The network topology used for the experiments is shown in Fig. 4.10 and is generated with Mininet through the following command:

```
$ sudo mn --link tc,bw=10,delay=10ms
```

It's a simple network in which a client is connected to the server through a switch. The links between the switch and the two hosts are both characterized by a bandwidth of 10 Mbps and a delay of 10 ms. Consequently the RTT between the client and the server is equal to 40 ms, since each packet traverses two links (one to the switch, one to the destination) and the response traverses two links coming back.

For each one of the protocols under test a client application and a server one have been created (written in C). All the code developed and used to conduct these experiments can be compiled and executed following the guide in Appendix A, in the section about the handshake measurements of the user's manual. For what concerns TLS and DTLS protocols, the OpenSSL library has been used to develop client and server applications. The OpenSSL library has been compiled with the default configuration that includes the use of AES-NI hardware accelerator to optimize AES operations, if available. The machine used for the tests is equipped with an Intel(R) Core(TM)

[9] https://www.wireshark.org/

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>
#include <openssl/err.h>

int main (int argc, char** argv)
{
    SSL_CTX *ctx;
    int server;
    SSL *ssl;
    struct hostent *host;
    struct sockaddr_in addr;

    SSL_library_init();
    OpenSSL_add_all_algorithms(); /* Load cryptos, et.al. */
    SSL_load_error_strings(); /* Bring in and register error messages */
    ctx = SSL_CTX_new(TLS_client_method()); /* Create new context */

    SSL_CTX_set_min_proto_version(ctx, TLS1_2_VERSION);
    SSL_CTX_set_max_proto_version(ctx, TLS1_2_VERSION);

    ssl = SSL_new(ctx); /* create new SSL connection state */

    SSL_set_cipher_list(ssl, ''ECDHE-ECDSA-AES256-GCM-SHA384'');

    //conclusion of the initialization and start of the client

    return 0;
}
```

Figure 4.9. OpenSSL example for choosing TLS version and setting the ciphersuite to use for the handshake

i7-1065GT CPU that supports AES-NI instruction set, so the OpenSSL library may be up to 10 time faster in performing AES computations.

Regarding TLS protocol, in client and server is possible to select the version of the protocol to use thanks to two different methods provided by the library, so that both version 1.2 and 1.3 are tested: *SSL_CTX_set_min_proto_version()* and *SSL_CTX_set_max_proto_version()*. Fig. 4.9 shows an example of the use of these functions in a C program.

During the evaluation, the TLS handshake has been tested with all available ciphersuites that OpenSSL supports for each version of the protocol. For what concerns TLS 1.2 a ciphersuite can be selected in the client program thanks to the method *SSL_CTX_set_cipher_list()*. An example of the selection of a ciphersuite in a C client is represented in Fig. 4.9. In this case all the available ciphersuites are divided into 3 groups:

- Ciphersuites that use RSA as signature algorithm and DHE for key exchange;
- Ciphersuites that use RSA as a signature algorithm and ECDHE for key exchange;
- Ciphersuites that use ECDSA as a signature algorithm and ECDHE for key exchange.

Regarding TLS 1.3 there are 3 ciphersuites that can be used with the OpenSSL library to attempt the handshake, that are TLS_AES_128_GCM_SHA256, TLS_AES_256_GCM_SHA384, TLS_CHACHA20_POLY1305_SHA256. For this version of the protocol the library function to

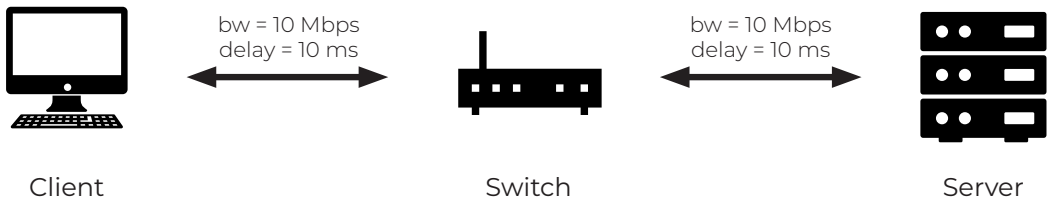


Figure 4.10. Topology created with Mininet to execute the experiments

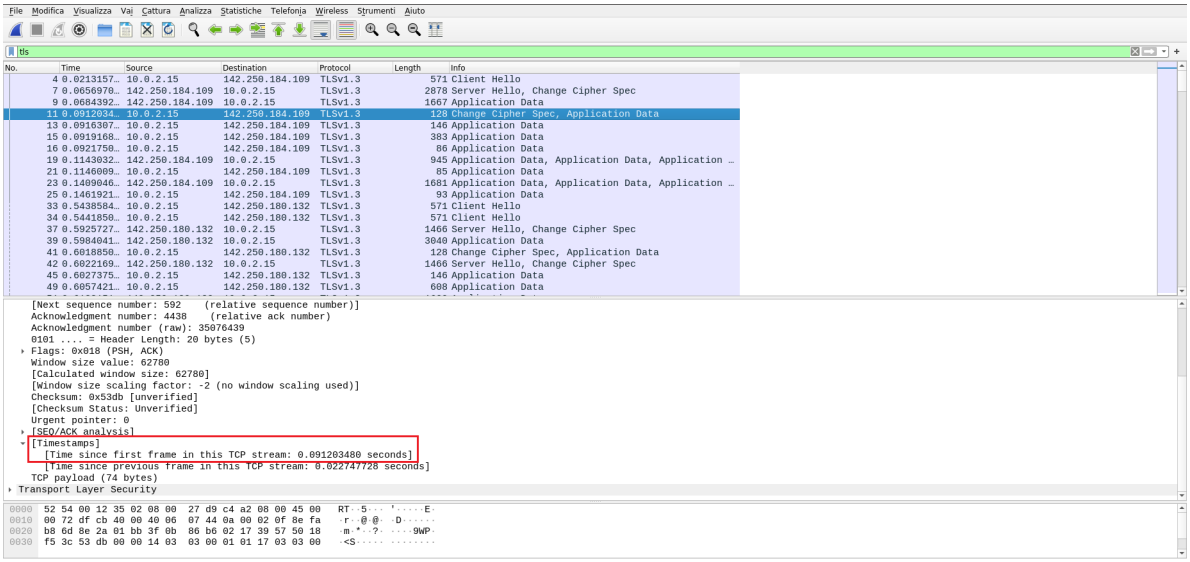


Figure 4.11. Screenshot of Wireshark showing the parameter (timestamp) used for the measurements

select the correct ciphersuite in the client is *SSL_CTX_set_ciphersuites()* and it can be used in the same way with respect to the aforementioned function for TLS 1.2.

DTLS 1.2 client and server are implemented with other two C programs including the OpenSSL library. The most important different between DTLS 1.2 and TLS 1.2 applications is the implementation of the cookie exchange process. Indeed during the DTLS 1.2 handshake, when the server receives the first *ClientHello* message from the client, it answers with a cookie. This cookie must be sent again by the client to the server in order to proceed with the handshake. This exchange is introduced in the DTLS handshake in order to avoid DoS attacks and IP spoofing. In order to implement the cookie exchange with OpenSSL, it's necessary to use *SSL_CTX_set_cookie_generate_cb()* and *SSL_CTX_set_cookie_verify_cb()* functions in the server program, as depicted in Fig. 4.12. The first one receives as parameter an SSL_CTX pointer and a pointer to a function that can be used to generate a cookie, while the second one receives the pointer to the same SSL_CTX and a pointer to a function useful for the verification of the correctness of the cookie sent back by the client. The handshake of this protocol is tested with the same ciphersuites that are used for TLS 1.2.

In order to setup the experiments with QUIC protocol the lsquic library is used. In particular, two programs named *echo_client* and *echo_server* that are available on the repository of the library are used for the tests. These two programs were meant for sending a message with the client and receiving back the same message by the server. In order to test the handshake of QUIC protocol the client program has been modified in order to create the connection with the server and then terminate, after having released all the memory resources previously acquired. This is obtained inserting the proper instructions for the cleaning of the program at the end of

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <openssl/bio.h>
#include <openssl/rand.h>

int generate_cookie(SSL *ssl, unsigned char *cookie, unsigned int
    *cookie_len) {

    //implementation of the logic to generate the cookie

}

int verify_cookie(SSL *ssl, const unsigned char *cookie, unsigned int
    cookie_len){

    //implementation of the logic to verify the correctness of the cookie

}

int main (int argc, char** argv)
{
    SSL_CTX *ctx;
    SSL *ssl;

    SSL_library_init();
    OpenSSL_add_all_algorithms(); /* Load cryptos, et.al. */
    SSL_load_error_strings(); /* Bring in and register error messages */
    ctx = SSL_CTX_new(DTLS_server_method()); /* Create new context */

    SSL_CTX_set_max_proto_version(ctx, DTLS1_2_VERSION);
    SSL_CTX_set_min_proto_version(ctx, DTLS1_2_VERSION);

    SSL_CTX_set_cookie_generate_cb(ctx, generate_cookie);
    SSL_CTX_set_cookie_verify_cb(ctx, verify_cookie);

    //conclusion of the initialization and start of the client

    return 0;
}
```

Figure 4.12. OpenSSL example for implementing DTLS 1.2 cookie exchange during the handshake

the *on_hsk_done()* callback, a function that is called every time that the handshake is completed. The version of QUIC selected for the measurements is version 1, that is the most recent of the protocol, standardized by IETF with RFC 9000 [32].

Another important point to discuss, regarding the handshake, is about the authentication process. The handshakes measured during these experiments implement server authentication. To this purpose, it was necessary to create a trusted certificate for a CA (both an RSA and a DSA one), that issued a certificate for the server. Thanks to OpenSSL toolkit it was possible to accomplish this task. Indeed it offers the possibility to crate and manipulate keys and certificates. The first step to execute is about generating a key and certificate for the CA. The following commands can be used to generate an RSA 2048 bit key and the corresponding trusted certificate:

```
$ openssl req -new -newkey rsa:2048 -nodes -out ca.csr -keyout ca.key
$ openssl x509 -trustout -signkey ca.key -days 365 -req -in ca.csr -out
    ca.pem
```

At this point the file named "ca.key" contains the key of the CA, while the file "ca.pem" contains the RSA certificate. The following steps consists in generating the server RSA key, creating the Certificate Signing Request (CSR) and making the CA issue a certificate for the server. These procedures can be executed through the following commands:

```
$ openssl genrsa -out server.key 2048
$ openssl req -new -key server.key -out server.csr
$ openssl x509 -req -days 365 -in server.csr -CA ca.pem -CAkey ca.key
    -set_serial 01 -out server.crt
```

At the end of this process the file named "server.key" contains the server RSA key (2048 bits), while the certificate can be found inside the "server.crt" file. The same procedure can be executed to obtain a CA with a DSA certificate that issues a new DSA certificate for the server. The first two commands to execute to generate the key and the certificate for the CA are the followings:

```
$ openssl ecparam -genkey -name prime256v1 -out ca.key
$ openssl req -x509 -new -SHA384 -nodes -key ca.key -days 3650 -out ca.crt
```

Now "ca.key" and "ca.pem" contain respectively the key and the certificate of the CA. Subsequently the Certification Authority issues a DSA certificate for a server that owns a DSA key and requests it. The following commands can be used to implement these last steps:

```
$ openssl ecparam -genkey -name prime256v1 -out server.key
$ openssl req -new -SHA384 -key server.key -nodes -out server.csr
$ openssl x509 -req -SHA384 -days 365 -in server.csr -CA ca.crt -CAkey
    ca.key -CAcreateserial -out server.crt
```

Also in this case, the output is composed by two files, "server.key" and "server.crt", inside which there are the key and the certificate of the server.

During the experiments the handshake time has been measured with Wireshark. With this tool is possible to capture all the packets exchanged between client and server during the handshake and then, since each packet is characterized by a timestamp, the time needed to setup the connection can be computed through the difference between the timestamp of the last packet of the handshake and the timestamp of the first one. For example, as can be seen from Fig. 4.11, for each TLS (and TCP) packet Wireshark gives the possibility to read in the *Timestamps* field the time elapsed since the first frame in the TCP stream. In this way it's very easy to compute the duration of the handshake (TCP+TLS), indeed the timestamps of the last packet of the TLS handshake represents the time from the client TCP SYN (first packet in TCP handshake) to the client *ChangeCipherSpec* message (last packet in TLS 1.3 handshake).

For each protocol and each ciphersuite 10 measurements have been executed, with the exception of QUIC that has been tested 30 times, since lsquic library doesn't enable the choice of the ciphersuite used for the handshake. The results of the experiments are summarized in the CDF chart in Fig. 4.13. As can be noticed, the fastest protocol to complete the handshake is QUIC, that results in an average handshake time of around 34,5 ms. As expected the result for QUIC protocol is very close to 1-RTT (40 ms). TLS 1.3 demonstrates an handshake time higher than QUIC with a mean value of 87,2 ms. For this protocol the values obtained with different ciphersuites have been grouped together in the chart since there were no considerable differences between all the measurements. Consequently, since 10 measurements have been executed for each ciphersuite, the line of TLS 1.3 in the chart represents 30 values in total (the same as QUIC protocol). Also in this case the results follow the expectations, indeed TLS 1.3 showed its 2-RTT handshake (1-RTT for TCP channel setup and 1-RTT for TLS handshake properly). For what concerns TLS 1.2 handshake 3 groups of ciphersuites have been analyzed. The ECDHE algorithm for key exchange can be used both with RSA and ECDSA signature algorithm, providing a mean handshake time of around 116,7 ms and 116,2 ms respectively. If a ciphersuite that employs

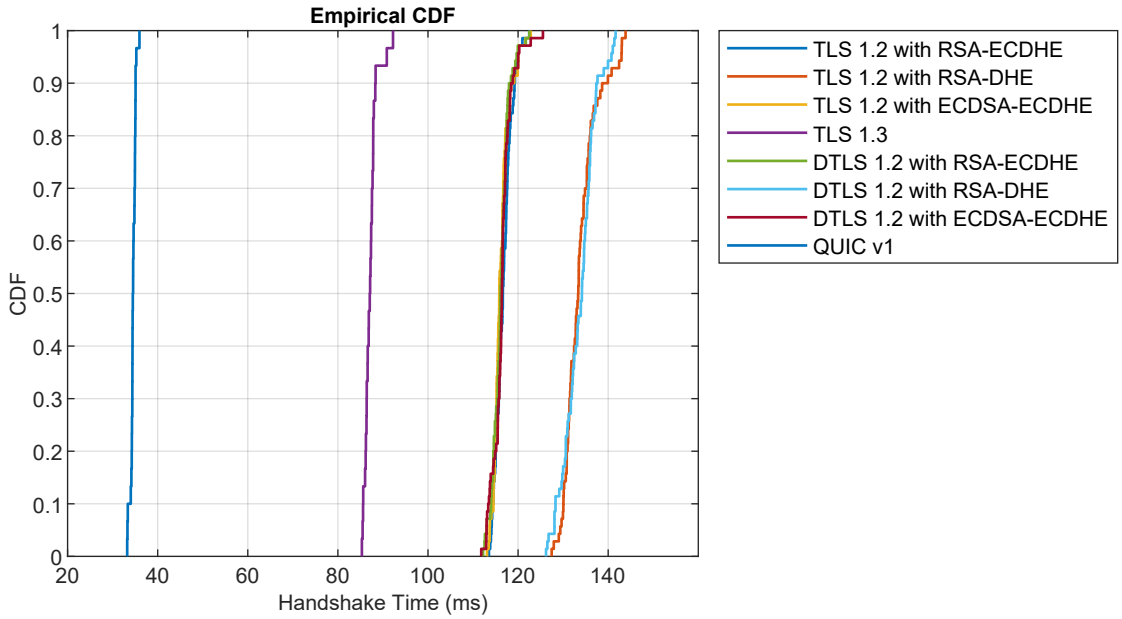


Figure 4.13. Cumulative Distribution Function of handshake time for TLS, DTLS and QUIC using different ciphersuites

RSA as a signature algorithm and DHE as a key exchange algorithm is chosen to establish the connection between client and server, the average handshake time for TLS 1.2 increases till a value of 133,8 ms. This difference is due to the fact that ECC is stronger than RSA and needs keys of lower size, since ECC key size of 256 bits is equivalent to a 3072-bit RSA key and 10000 times stronger than a 2048-bit RSA key. Consequently smaller keys means less data that must be transmitted from the server to the client during an SSL handshake. In addition, ECC requires less processing power (CPU) and memory, resulting in significantly faster response times and throughput. As the chart shows, DTLS 1.2 provided similar performance with respect to TLS 1.2 on average with all the ciphersuites that have been used for the tests. Indeed the mean handshake time of DTLS, 1.2 when a ciphersuite that is composed by RSA and DHE is used, is equal to 133,7 ms. Also in this case the use of ECC resulted in an advantage in terms of time spent for the handshake. Indeed according to the results, DTLS 1.2 demonstrates an handshake time around 116,2 ms when ECDHE algorithms are used for key exchange and RSA for signature, while the solution that involves ECDSA and ECDHE revealed an average handshake time of 116,4 ms.

4.3.5 File Transfer Time measurements

Another application in which the protocols studied in this thesis are employed, especially TLS and QUIC, is the file download from a server. For this reason the last metric evaluated during the performance evaluation phase is the file transfer time. To this purpose a client application and a server one have been written for all the three protocols under tests. In addition the performance of the three security protocols is compared also with a solution in which the file is transferred with the HTTP protocol without any security feature implemented. All the applications are written in C language: for TLS and DTLS protocols the OpenSSL library has been used, while lsquic library has been chosen to implement QUIC client and server.

The infrastructure used to conduct the experiments is again created with the help of Mininet. The base network topology used is the same as the one for the experiments on the handshake: two hosts (a client and a server) connected through a switch. But this time the file transfer performance of the protocols are tested in different network conditions, varying through Mininet the parameters of the link connecting the hosts and the switch, such as bandwidth and delay.

For both the parameters an high and a low value are chosen. For the bandwidth the two values selected are 5Mbps and 100Mbps, while the delay can be equal to 5ms or 50ms (consequently the RTT between client and server will be respectively equal to 20ms or 200ms).

Another factor introduced in each test is represented by the file size, since the performance of a protocol could be affected by the amount of bytes that it has to transfer. Three files with different size have been created to execute the experiments, respectively of 500kB, 5MB and 50MB. The file creation was addressed with the *rand* command of the OpenSSL toolkit. This command is used to generate a requested number of random bytes using a cryptographically secure pseudo random number generator. For example the following command can be used to generate 500K random bytes and write them to a file instead of the standard output.

```
$ openssl rand -out [output_file] 512000
```

During the tests all the three files are downloaded 10 times with all the protocols, each time in a different combination of network parameters. The measurements are executed in each one of the four combinations that can be deduced from the Table 4.3 reported below.

Parameter	*Values*
Bandwidth	5Mbps, 100Mbps
Delay	5ms, 50ms
File Size	500kB, 5MB, 50MB

Table 4.3. Different environment parameters for file transfer experiments

The starting point to implement and execute these tests is the set of applications developed to measure the handshake, described in the previous section. All these applications are extended, both client and server side, in order to add the possibility for the client to request a file and for the server to identify the requested file and send it to the client as a response. All the code developed and used to conduct these experiments for all the protocols under test, can be compiled and executed following the guide in Appendix A, in the section of the user's manual regarding file transfer time measurements.

For what concerns the OpenSSL library there are two functions that can be used to send and receive bytes protected through a security protocol (TLS or DTLS): *SSL_read()* and *SSL_write()*. Fig. 4.14 and Fig. 4.15 show an example in which these two functions are used both client and server side. As can be noticed, the client, after having initialized all the necessary contexts of the library, first sends the name of the requested file to the server using *SSL_write()* function. Then it starts a while cycle in order to read in a buffer all the bytes that will be sent by the server, till the end of the file. All these bytes are then written on a special file, named /dev/null, that is used in order to discard immediately the input provided to the *fwrite()* function and avoid wasting CPU cycles in output operations. In this way only the performance of the protocol will be measured, without taking into account not relevant CPU operations. On server side instead, the application, once the variable and contexts have been initialized and the connections has been established with a client, waits for the name of the file to send and reads it using the *SSL_read()* function. Once this is done, the server program will read the file in chunks inside a buffer and will send it to the client through the *SSL_write()* function. When the file has been completely send the server closes the connection and waits for a possible new one.

There is an important difference to underline between the implementations of DTLS and TLS protocol with the OpenSSL library, in particular regarding the server side. This difference is due to the fact that DTLS, differently from TLS, is based on UDP and so each endpoint that sends some packets doesn't need to wait for an ACK to proceed sending other data on the channel. Consequently the problem arises when the throughput of the sender becomes bigger than the bandwidth of the channel: this situation will lead to packet loss and will make the receiver impossible to read all the data the sender has sent. For what concerns the experiments of this thesis, when the DTLS server is requested to send a big file (5MB or 50MB) and no countermeasures are applied, especially if the network is characterized by a low bandwidth (5Mbps), the network becomes soon congested and the client can't receive all the bytes that compose the file,

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>

int main (int argc, char** argv)
{
    //variable and context initialization
    //client start and handshake

    SSL_write(ssl, req, strlen(req));
    fp = fopen("/dev/null","wb");

    while ((n = SSL_read(ssl, buf, BUF_SIZE)) != 0) {
        fwrite(buf, 1, n, fp);
        memset(buf, 0, BUF_SIZE);
    }

    //resource deallocation and clean up

    return 0;
}
```

Figure 4.14. OpenSSL example for implementing file exchange client side

but only a portion of that. The solution applied to the DTLS implementation to address this problem is the insertion of an instruction, named usleep(), between sending one packet and the next one, in order to slow down the server. The value passed as a parameter to the function is the number of microseconds for which the execution of the calling thread is at least suspended when the usleep() is called. Obviously this value changes accordingly to the bandwidth: when the network is characterized by a bandwidth equal to 5Mbps, it's necessary to pass 1400 as parameter to the usleep(), in order to let also to the biggest file to be completely exchanged between client and server, while when the bandwidth is equal to 100Mbps it's enough that the server waits 15 microseconds between one packets sent and the next one.

Regarding QUIC protocol instead, the functions that are used to send and receive bytes are *lsquic_stream_read()* and *lsquic_stream_write()*. Both these functions receive the same parameters: the identifier of a stream, a buffer (that is used as a destination for reading or as a source for writing) and the size of the buffer. They are used inside two different callbacks *on_read()* and *on_write()*, that are called each time the application (client or server) express the intention to receive or send some data. Every time an application using lsquic library wants to write to a stream must call the function *lsquic_stream_wantwrite()*, that receives two parameters: the first one is the identifier of the stream to write, while the second is a boolean value indicating whether the caller wants to write to stream or not. The same thing happens in case of reading willingness with the function *lsquic_stream_wantread()*, whose prototype shows the same input parameters as the corresponding function for writing.

The starting point for creating the client and server application to implement file transfer with QUIC protocol is again composed by the *echo_client* and *echo_server* applications available in the bin folder of the Github repository of lsquic library. If for the previous experiments the only change applied is the interruption of the computation once the handshake has been completed, in this situation the two programs must be further extended. On client side, the possibility to specify as a command line argument (with -f option) the name of the requested file is added. After the connection with the server is established and a new stream is created, the client informs the server about the name of the file he is willing to download writing on the stream just created. Subsequently the client starts the reading phase, waiting for the server response and it will write

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>

int main (int argc, char** argv)
{
    //variable and context initialization

    while(1) {

        //server start and connection accept

        SSL_read(ssl, filename, N);

        FILE *file = fopen(filename, "r");
        while((n = fread(buf, 1, BUF_SIZE, file))>0) {
            SSL_write(ssl, buf, n);
            memset(buf, 0, BUF_SIZE);
        }
        fclose(file);
    }

    //resource deallocation and clean up

    return 0;
}
```

Figure 4.15. OpenSSL example for implementing file exchange server side

all the bytes received on /dev/null in order to avoid affecting the performance of the protocol, wasting CPU cycles for output operations. On the other hand, when the server has accepted the connection from the client and a new stream has been created, it immediately reads the name of the file the client wants to download from the stream. After that the writing phase starts: the server opens the requested file, reads it in chunks and sends them to the client, writing the buffer containing each time a new chunk on the stream.

The performance of TLS, DTLS and QUIC security protocols are compared during these experiments also with the HTTP protocol. This is done to understand what is the price to pay in terms of performance to obtain a secure communication between client and server. Also for the HTTP protocol a simple client application and a server one have been written in C language. These programs exploit socket programming in order to make the client and the server communicate using the *send()* and *recv()* functions. The HTTP server has to create a TCP socket and bind it to the server address. Then using the *listen()* function the server socket is put in a passive mode, where it waits for the client to contact the server to make a connection. When the execution reaches the call to *accept()* function, the connection is established between client and server and they are ready to transfer data. On the client side the C program has to create a socket and connect it to the server. At this point the client can send an HTTP request to the server containing the name of the file it wants to download and the server, after having opened and read the correct file in chunks, has to send it back to the client as a response.

All the results obtained during measurements are summarized in different boxplots. Each chart is representative for a combination of network conditions and a file size. For what concerns TLS 1.2 and DTLS 1.2 protocols in each chart has been reported also the ciphersuite negotiated during the handshake. Both protocol has been tested with two different ciphersuites, one

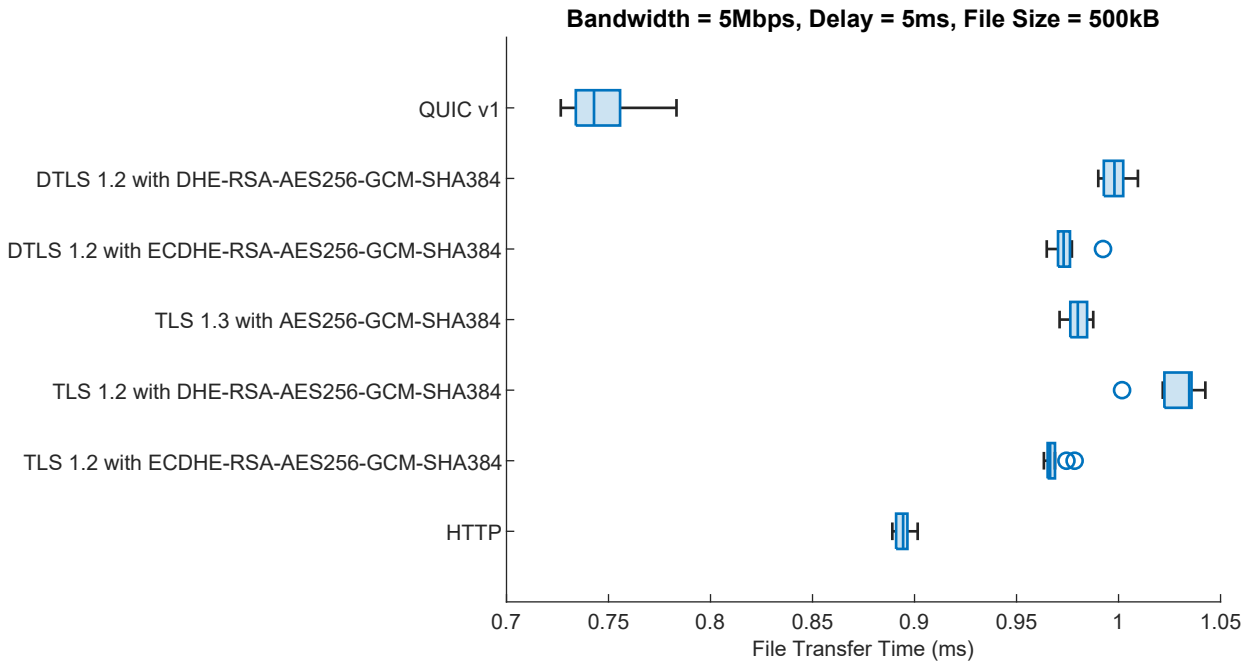


Figure 4.16. Boxplot of time needed to download a 500kB file with HTTP, TLS, DTLS and QUIC in a network with 5Mbps bandwidth and 5ms delay

with ECDHE algorithm and the other one characterized by DHE, since the choice of the ciphersuite demonstrated to affect the performance of the protocol during the experiments about the handshake. For TLS 1.3 the ciphersuite AES256-GCM-SHA384 has been chosen, since the three available ciphersuites for the protocol implemented with OpenSSL resulted in very similar with respect to each other. Among all the available versions of QUIC, the most recent one (Version 1) has been chosen for the tests.

The first chart reported in Fig. 4.16 is a boxplot about the time needed by the tested protocols to download a 500kB file in a network characterized by 5Mbps of bandwidth and 5ms of delay in each link. As can be clearly seen from the chart, QUIC is the fastest protocol in downloading the file, resulting in a mean file download time equal to 0,75 s, followed by HTTP that demonstrated an average file download time of around 0,89 s. All the remaining protocols had very similar results with respect to each other. Indeed TLS 1.3, TLS 1.2 with ECDHE and DTLS 1.2 with ECDHE resulted in a mean value for the time needed to download the 500kB file respectively equal to 0,98 s, 0,97 s and 0,97 s. As can be expected, the slowest protocols are TLS 1.2 with DHE and DTLS 1.2 with DHE with an average file download time equal to 1,03 s and 1,00 s. This is due to the choice of the ciphersuite, but also to the duration of the handshake (about 3-RTT for both protocols), that in case of a file of such a small size represents an important part in the entire time measurement.

Fig. 4.17 shows the results of the experiments in the same network conditions but with a file of 5MB. As can be noticed from the chart, the results are very close to the one reported above for the 500kB file. QUIC is again the fastest protocol with an average file download time equal to 6,46 s, while TLS 1.2 and DTLS 1.2 with DHE are the slowest (9,31 s and 9,22 s respectively). In the middle all the other protocol: HTTP with 8,88 s of mean time to download the file leading the group and TLS 1.3, TLS 1.2 with ECDHE and DTLS 1.2 with ECDHE that resulted respectively in an average time equal to 9,19 s, 9,16 s and 9,20 s. Since the size of the file transferred is clearly increased with respect to the previous experiments (about 10 times larger), the differences in the results due to the duration of the handshake are less marked, as its importance compared to the entire time required to download the file (in terms of time) is considerably reduced.

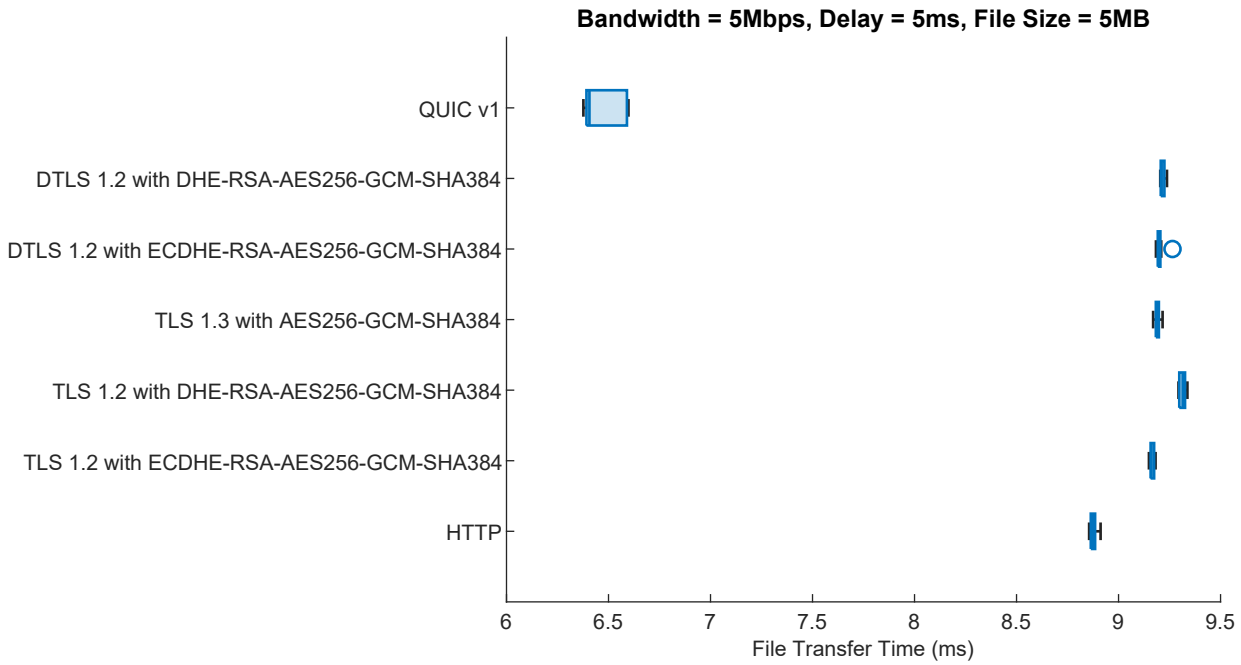


Figure 4.17. Boxplot of time needed to download a 5MB file with HTTP, TLS, DTLS and QUIC in a network with 5Mbps bandwidth and 5ms delay

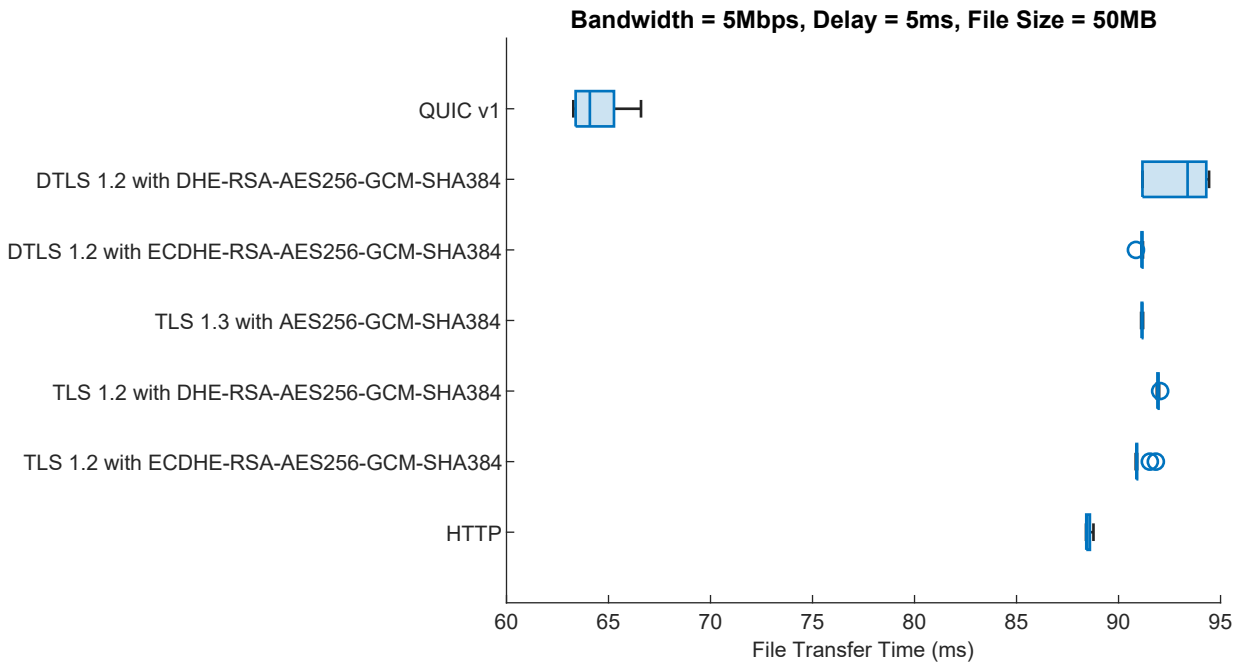


Figure 4.18. Boxplot of time needed to download a 50MB file with HTTP, TLS, DTLS and QUIC in a network with 5Mbps bandwidth and 5ms delay

The results for the transfer time values measured considering a big file of 50MB are reported in Fig. 4.18. In this case the behaviour mirrors that obtained with the two files of smaller size.

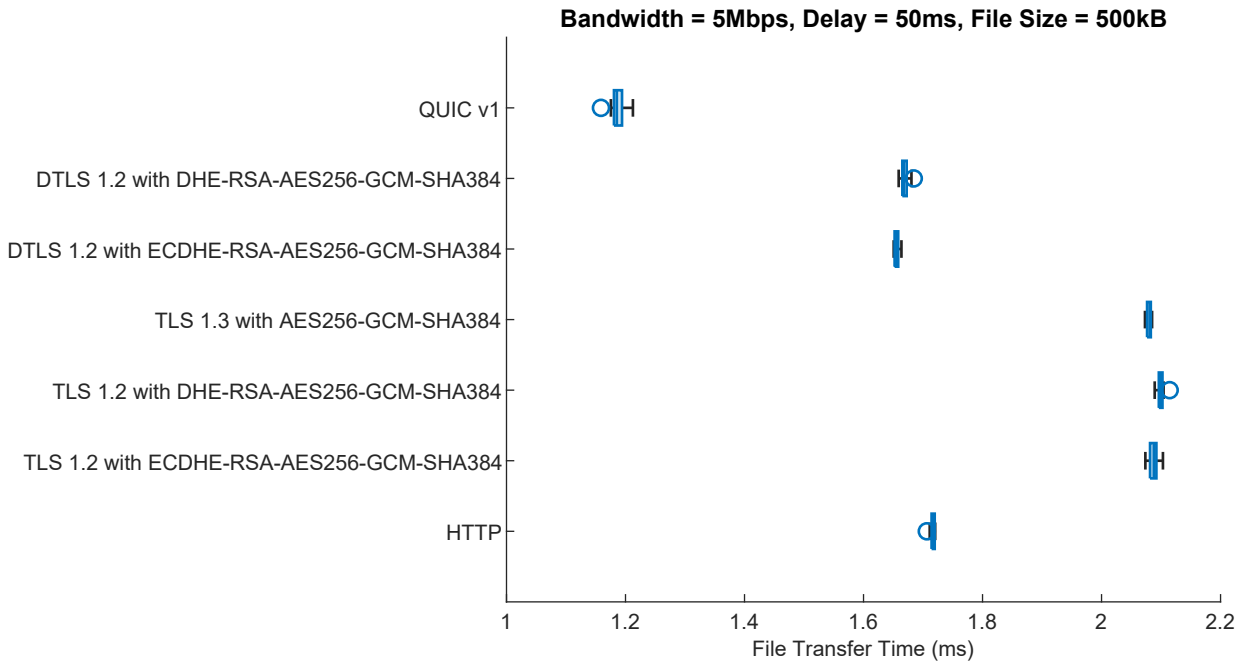


Figure 4.19. Boxplot of time needed to download a 500kB file with HTTP, TLS, DTLS and QUIC in a network with 5Mbps bandwidth and 50ms delay

QUIC resulted to be again the best solution (64,52 s of average file download time) and it clearly outperforms all the other protocol, especially the secure ones. HTTP follows QUIC v1 with a mean time measured equal to 88,52 s. All the other solutions represented by TLS 1.2, TLS 1.3 and DTLS 1.2 resulted in an average file transfer time between 91 and 93 s. The differences between these last presented solutions are really not noticeable and are reduced due to the big file size with respect to the experiments previously presented with smaller files in the same network environment.

The second situation that is presented in the charts is the worst one in terms of network conditions: it is indeed characterized by the smallest value for the bandwidth (5Mbps) and the greatest value for the delay (50ms). On average QUIC resulted to be the best choice to download the file independently of the size also in this scenario. As has been reported in some of the work analysed in the Related Works chapter, QUIC usually outperforms the other protocols especially in poor networks conditions. So in this case the results obtained during the tests confirmed what can be expected.

The boxplot Fig. 4.19 shows the results of the measurements of the file transfer time considering a network with 5Mbps and 5ms of delay in each link and a file of 500kB. The chart can be divided in three zones: the first one composed only by QUIC protocol, that is the fastest in downloading the file, the second one that involves DTLS 1.2 and HTTP and the third zone in which there are TLS 1.2 and TLS 1.3. The time needed on average by QUIC to download a 500kB file in poor network conditions is equal to 1,19 s. In the second zone DTLS 1.2 resulted to be a little bit faster with respect to HTTP with an average file transfer time equal to 1,66 s (independently of the ciphersuite selected), against 1,72 s that is the value obtained with HTTP. The performance of DTLS are better that HTTP and considerably better than TLS in this case because the presence of the sleep() instruction inside the code to slow down the server is mitigated by the high vale of the delay. In this case it's easier to exploit the potential of DTLS protocol in terms of speed in transferring files. In the last zone a similar behaviour for what concerns TLS 1.3 and TLS 1.2 can be noticed. Indeed the average values obtained for the file transfer time are respectively 2,08 s, 2,09 s (TLS 1.2 with ECDHE) and 2,10 s (TLS 1.2 with DHE).

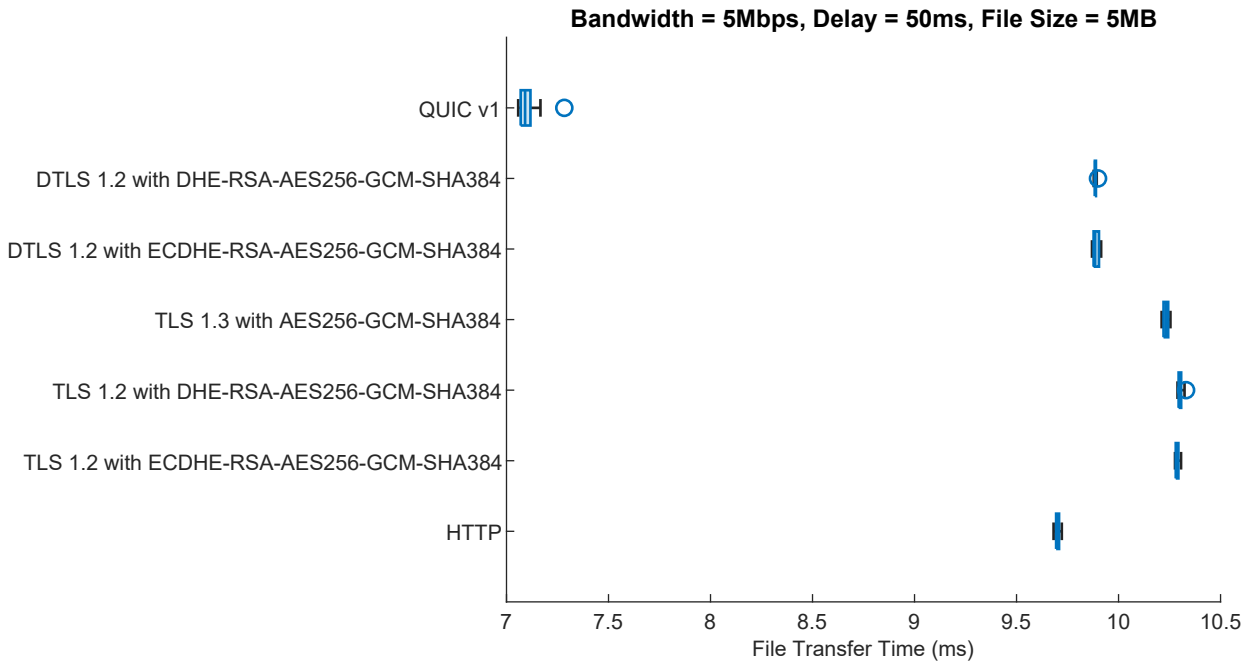


Figure 4.20. Boxplot of time needed to download a 5MB file with HTTP, TLS, DTLS and QUIC in a network with 5Mbps bandwidth and 50ms delay

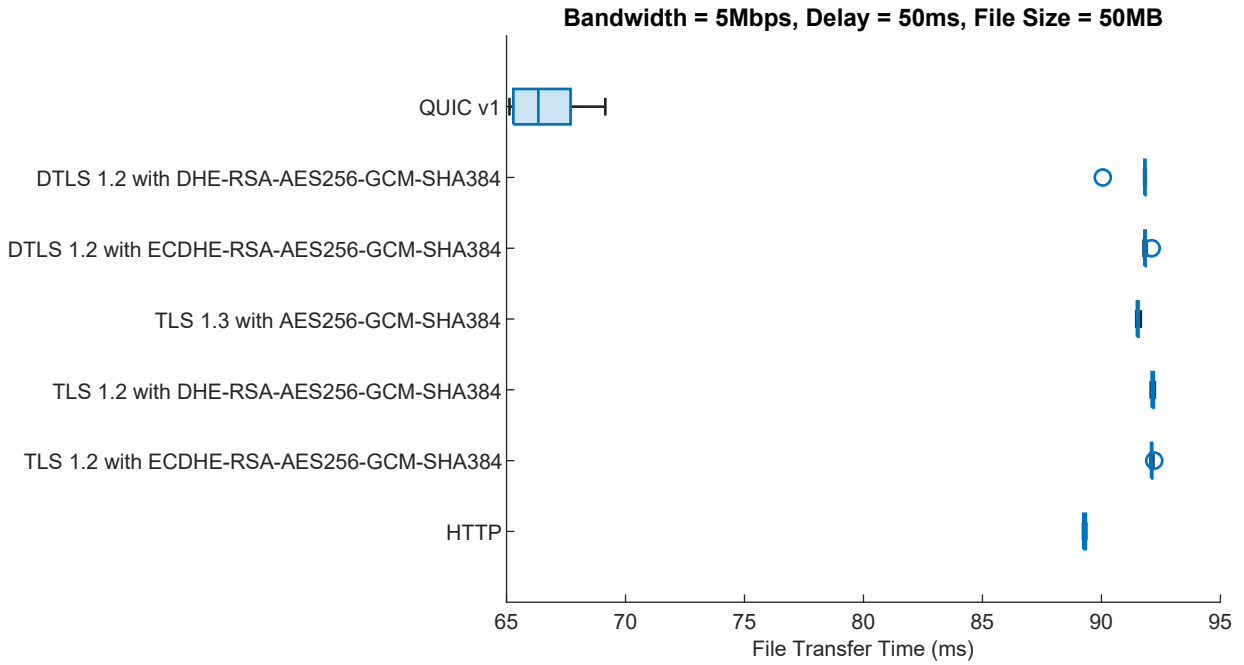


Figure 4.21. Boxplot of time needed to download a 5MB file with HTTP, TLS, DTLS and QUIC in a network with 5Mbps bandwidth and 50ms delay

When the values measured during the download of a file of 5MB are analysed a similar situation is highlighted (Fig. 4.20). QUIC v1 takes on average 7,11 s to download the file and it's again the

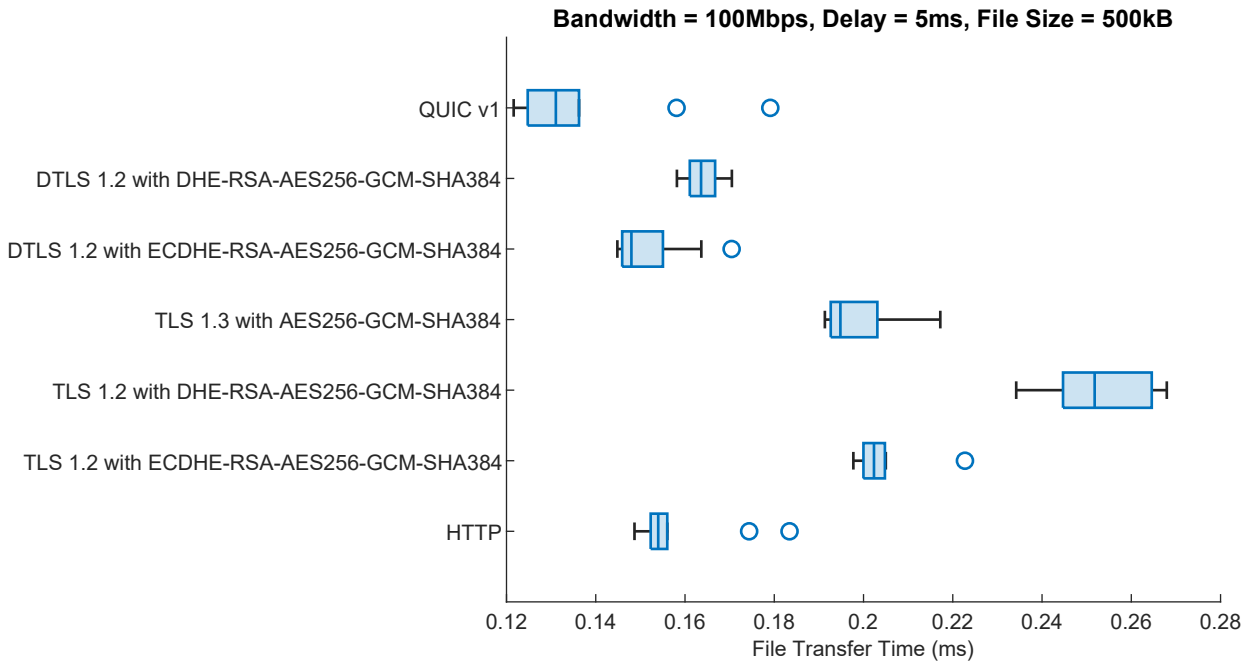


Figure 4.22. Boxplot of time needed to download a 500kB file with HTTP, TLS, DTLS and QUIC in a network with 100Mbps bandwidth and 5ms delay

fastest protocol in this set of measurements. In this case, the differences between the second and the third zone are less remarkable. Indeed HTTP provided values for the file transfer time that range between 9,68 s and 9,72 s, with a mean value equal 9,70, while DTLS average file download time is only 0,19 s higher than that (this is valid for both evaluated ciphersuites). In the last zone TLS 1.3 resulted a bit faster that TLS 1.2, with mean file transfer values respectively equal to 10,23 s and 10,30 s.

If we consider the results obtained transferring a big file (50MB) between client and server, in the same networks conditions, there are no more three different zones in which the behaviours of the different protocols can be divided, but QUIC and HTTP resulted faster that all the others, as can be noticed looking at Fig. 4.21. When QUIC v1 is used as the protocol for file transfer, the maximum value obtained for the download time is equal to 69,15 s, while the minimum is 65,12. Even if a considerable variability characterizes these values, QUIC v1 is once again the fastest protocol by far in transferring a big file. It's again followed by HTTP that provided a mean file transfer time of around 89,30 s, and so more than 20 s greater than QUIC. All the others protocol (TLS and DTLS in all their versions) resulted in a similar behaviour, with a mean value that ranges between 91,54 and 92,16 seconds. It's important to notice that, keeping the same network conditions and increasing the file size, the performance of DTLS protocol worsen. This is probably due to the presence of an instruction (usleep(1400)) in the server code, used to slow down the server process and to make the client possible to read all the packets. The problem is that the usleep() instruction gives only guarantees on the minimum number of microseconds that the process execution will be suspended. Since with the increase of the file size, also the number of sent packets becomes considerably greater and consequently also the number of calls to the usleep() instruction, probably the time the server has to wait in order to avoid make the network congested represents an important portion of the time needed in total by the client to download the file.

The next scenario considered is the best one in terms of network conditions: it's indeed characterized by the links with high bandwidth (100Mbps) and low delay (5ms). Also in this case all three different files have been downloaded 10 times with all the protocols. The chart in Fig. 4.22 is a boxplot generated with the results of the file transfer time considering the small file (500kB).

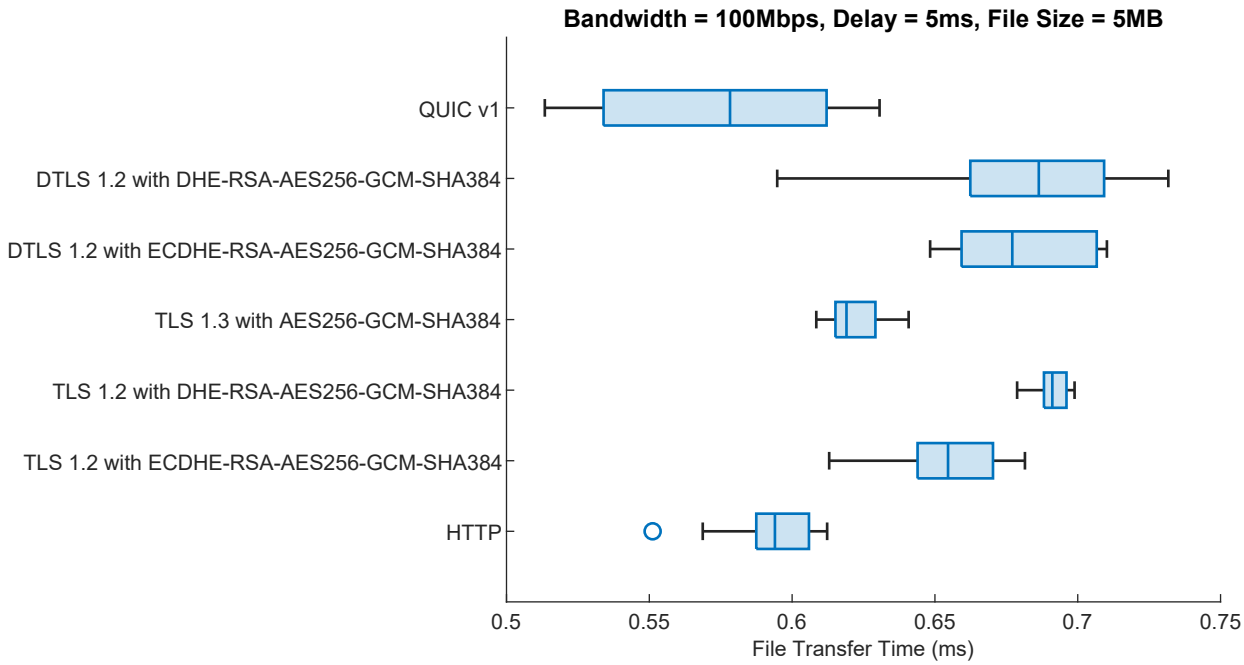


Figure 4.23. Boxplot of time needed to download a 5MB file with HTTP, TLS, DTLS and QUIC in a network with 100Mbps bandwidth and 5ms delay

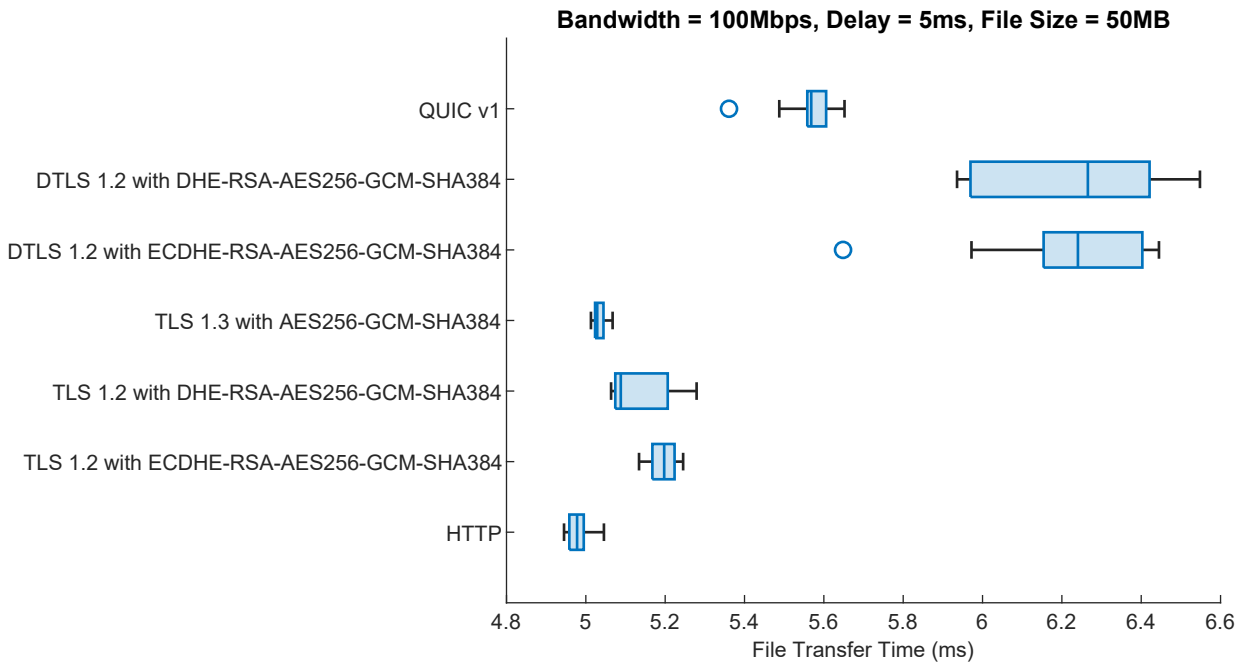


Figure 4.24. Boxplot of time needed to download a 50MB file with HTTP, TLS, DTLS and QUIC in a network with 100Mbps bandwidth and 5ms delay

As can be easily noticed, there is no a protocol that outperforms the others in downloading this file: QUIC v1, HTTP, DTLS 1.2 with ECDHE and DTLS 1.2 with DHE provided similar

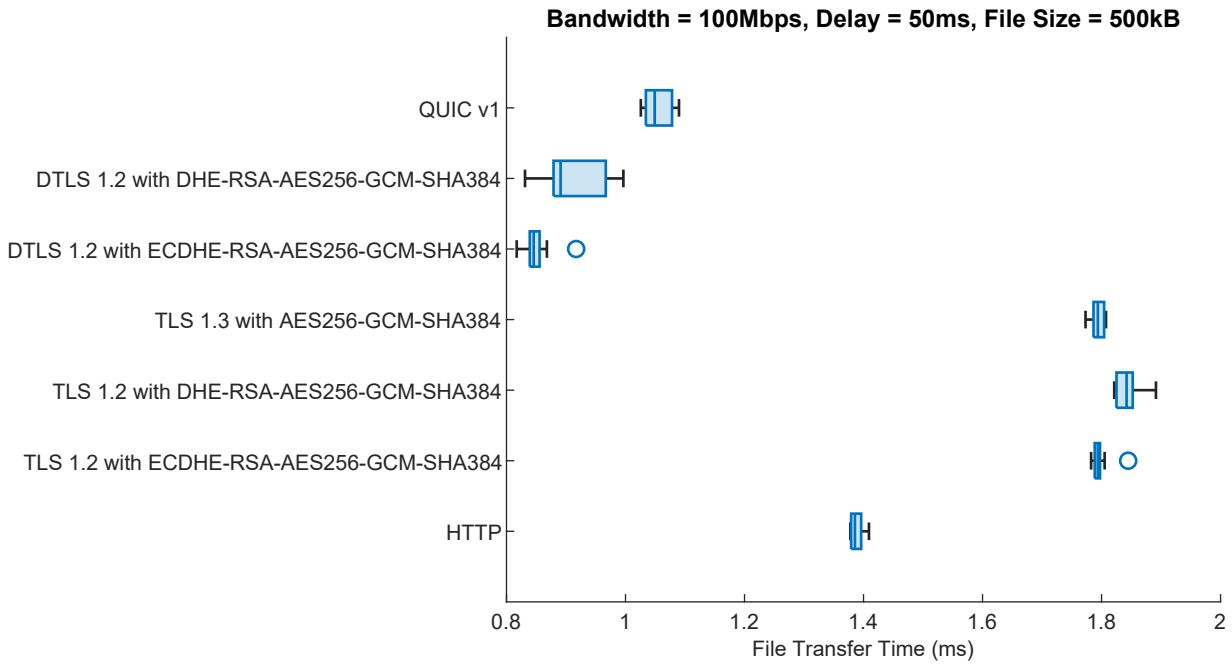


Figure 4.25. Boxplot of time needed to download a 500kB file with HTTP, TLS, DTLS and QUIC in a network with 100Mbps bandwidth and 50ms delay

performance with an average file transfer time respectively equal to 0,14, 0,16, 0,15, 0,16 seconds. TLS (1.2 and 1.3) resulted in a worse performance with mean values that range between 0,20 s and 0,25 s.

If the size of the file is increased till 5MB, the behaviour changes a bit. Fig. 4.23 shows the best performance are obtained when QUIC v1 and HTTP are employed, with 0,58 s as the mean value for file transfer time for QUIC and 0,59 s for HTTP. For what concerns the remaining protocols, TLS 1.3 is the fastest on average (0,62 s), followed by all the others that provided a mean file transfer time that goes from 0,65 (for TLS 1.2 with ECDHE) to 0,69 s (for TLS 1.2 with DHE). The worsening in the performance of DTLS 1.2, when the file size increases, can be again noticed.

When the size of the transferred file reaches 50MB the results are qualitatively and quantitatively different with respect to the ones obtained with smaller sized files. Indeed in this situation HTTP is the best choice to transfer the 50MB file, resulting in an average file download time equal to 4,98 s. This protocol is immediately followed by TLS, with version 1.3 faster on average than version 1.2 of at least 0,1 seconds. QUIC, with its 5,56 seconds average file transfer time, is positioned in the middle of the chart in Fig. 4.24. This is not an unexpected results, since some of the works analysed in this thesis, have already highlighted in the past the fact that QUIC is able to outperforms the other protocols especially in poor networks conditions, but its performance worsen when it has to deal with big files and high performance networks.

The last combination of network parameters that remains to be analyzed, is the one characterized by high bandwidth (100Mbps) and high delay (50ms). The results about the file transfer time considering all the files tested (500kB, 5MB and 50MB) in this environment are summarized in charts, one for each file size, represented in Fig. 4.25, Fig. 4.26 and Fig. 4.27. For what concerns the small file (500KB), DTLS performance resulted better with respect to QUIC v1. In particular DTLS with ECDHE provided the shortest file transfer time on average, equal to 0,85 s, followed by DTLS 1.2 with DHE with a mean value of 0,91 s and QUIC v1 with 1,05 s. The slowest protocol in this case resulted to be TLS, with comparable performance for versione 1.3 and 1.2 with ECDHE (1,80 and 1,79 mean values) and a slightly grater average file transfer time for TLS 1.2 with DHE equal to 1,84 s. HTTP demonstrated to be a good solution, in the middle between

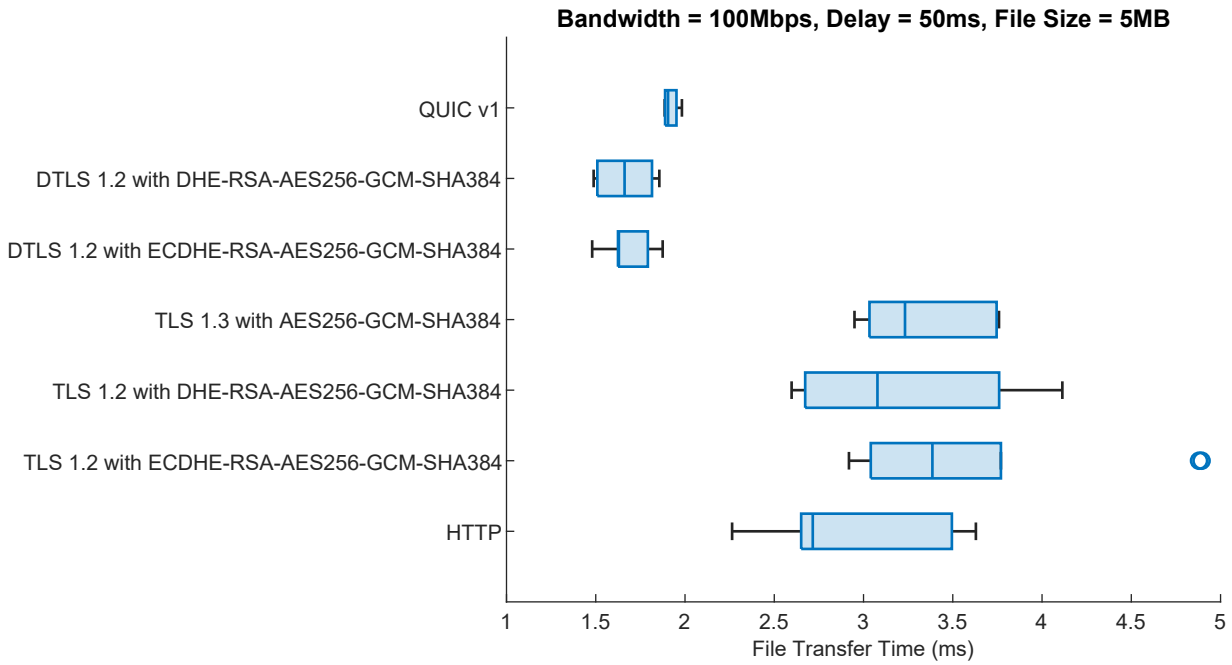


Figure 4.26. Boxplot of time needed to download a 5MB file with HTTP, TLS, DTLS and QUIC in a network with 100Mbps bandwidth and 50ms delay

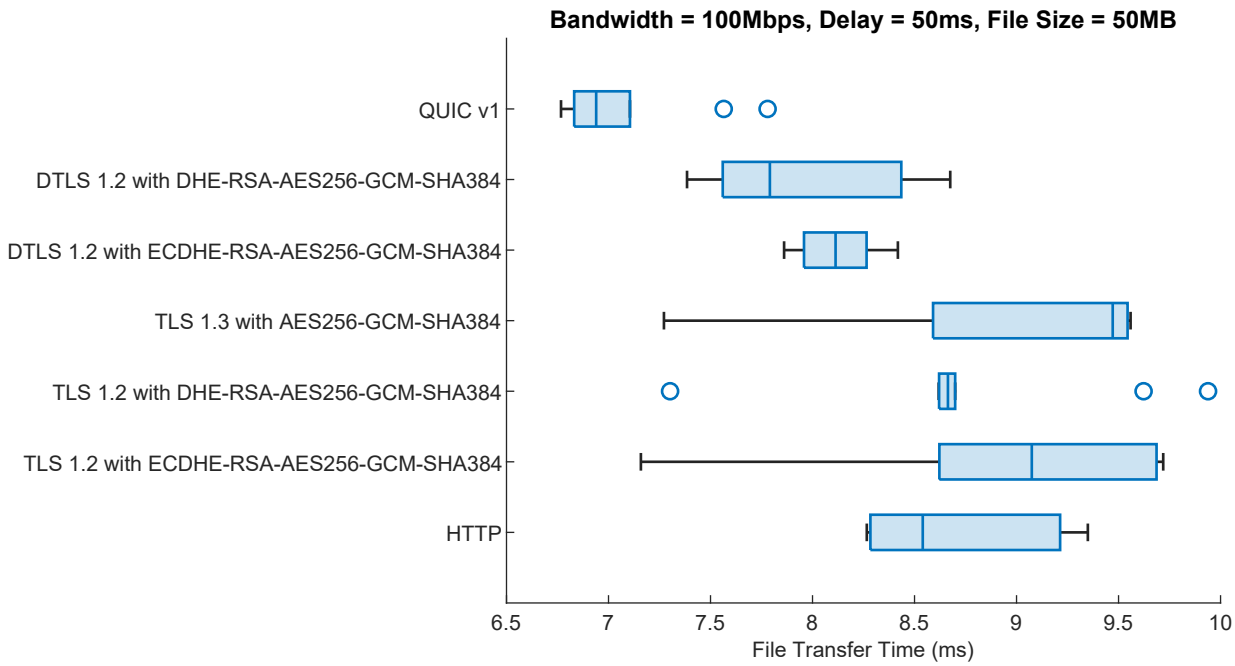


Figure 4.27. Boxplot of time needed to download a 50MB file with HTTP, TLS, DTLS and QUIC in a network with 100Mbps bandwidth and 50ms delay

the fastest and the slowest.

If the 5MB file is considered, the behaviour encountered is still the same, with the only

exception that the difference between HTTP and the slowest protocols is reduced. Indeed we have that, without any kind of security, the file takes on average 2,96 seconds to be downloaded, while if TLS is chosen to protect the file transfer, the mean value obtained ranges from 3,19 s and 3,61 s, depending on the version adopted. In this situation QUIC v1 and DTLS 1.2 are still the best choice and the fastest protocols for the file download. In particular DTLS provided a mean file transfer time equal to 1,67 s (independently of the ciphersuite used), while QUIC is a bit slower, resulting in a mean value of around 1,92 s.

The presence of the usleep(15) instruction in the code to slow down the DTLS server affects the performance of the protocol also in this environment with high bandwidth and high delay. Indeed, even if the execution of the server process is suspended every time for a reduced amount of microseconds, when the size of the file reaches a big value, such as 50 MB, the number of waits after sending each packet grows a lot. For this reason QUIC v1 protocol provided a better performance than DTLS 1.2 while downloading a 50MB file, providing a mean file transfer time equal to 7,06 s. All the remaining protocols (HTTP, TLS 1.2 and TLS 1.3) resulted in similar values for what concerns the mean file download time, between 8,68 s and 9,01 s. Surprisingly TLS 1.3 is the worst protocol in terms of performance during these measurements.

As a result, QUIC v1 provided good performance almost in all scenarios, especially in the environment with poor network conditions. The only case in which QUIC is not the best choice to download a file is when the network offers very high performance (high bandwidth and low delay) and the file to be downloaded is big sized. For what concerns TLS protocol, it resulted to be not the best solution for file download compared to the other protocol and in many situations this protocol provided the slowest performance. TLS 1.3 performed on average a little bit better than TLS 1.2 (probably it's due to the reduced handshake latency). DTLS protocol instead could represent a good solution to transfer a file, especially in a network with high bandwidth. Unfortunately its performance are limited by the fact that the server needs to be slowed down to prevent its throughput overcoming the network bandwidth and avoid the consequent packet loss. The problem of DTLS is that if packet loss occurs, the lack of an ACK mechanism (since DTLS runs on top of UDP) leads to only partial reception of the file, since packet lost will not be sent again. In most cases HTTP is a very good choice to transfer a file, but the drawback in this case is the lack of security and the fact that the file is sent and received completely in plaintext.

Chapter 5

Security Evaluation

5.1 TLS History

The Secure Sockets Layer (SSL) protocol was designed by the Netscape Corporation in order to provide a solution for a secure communication between client and server applications. SSL 2.0 was released in 1995 and represented the first version of the protocol. It was found affected by well-know security issues, which were addressed by the 1996 release of SSL 3.0 [33]. Nevertheless the problems weren't solved, indeed during the following years different vulnerabilities that affected this version of the protocol were detected and SSL 3.0 was phased out.

The Internet Engineering Task Force (IETF), a technical working group responsible for developing Internet standards to ensure communications compatibility across different implementations standardized Transport Layer Security protocol Version 1.0 (TLS 1.0) with RFC 2246 [34] in 1999. Even if TLS 1.0 is based on SSL 3.0. there are significant differences between the two protocols that make interoperation impossible. TLS 1.1, specified in RFC 4346 [2] (2006), was developed to address weaknesses discovered in TLS 1.0, primarily in the areas of initialization vector selection and padding error processing. Initialization vectors were made explicit to prevent a certain class of attacks on the Cipher Block Chaining (CBC) mode of operation used by TLS. The handling of padding errors was altered to treat a padding error as a bad message authentication code rather than a decryption failure. In addition, the TLS 1.1 RFC acknowledges attacks on CBC mode that rely on the time to compute the message authentication code (MAC). The TLS 1.1 specification states that to defend against such attacks, an implementation must process records in the same manner regardless of whether padding errors exist.

TLS 1.2, specified in RFC 5246 [1] in 2008, made several cryptographic enhancements with respect to previous versions, particularly in the area of hash functions, with the ability to use or specify the SHA-2 family of algorithms for hash, MAC, and Pseudorandom Function (PRF) computations. TLS 1.2 also adds authenticated encryption with associated data (AEAD) cipher suites. TLS 1.3, specified in RFC 8446 [3] (2018), represents a significant change to TLS that aims to address threats that have arisen over the years. Among the changes are a new handshake protocol, a new key derivation process that uses the HMAC-based Extract-and-Expand Key Derivation Function (HKDF), and the removal of cipher suites that use RSA key transport or static DiffieHellman (DH) key exchanges, the CBC mode of operation, or SHA-1. Many extensions defined for use with TLS 1.2 and previous versions cannot be used with TLS 1.3. In addition, this most recent version of the TLS protocol tries to include solutions against attacks that are possible against version 1.2. Indeed TLS 1.2 is vulnerable to man-in-the-middle and downgrade attacks. For example, POODLE is a MITM attack that exploits SSL-3 fallback to decrypt data. The variant in December 2014 exploits CBC padding vulnerabilities in TLS (version 1.0 and 1.2). To this end, TLS 1.3 introduces a downgrade protection mechanism, that is the inclusion of two predefined values (DOWNGRD01 or DOWNGRD00) in server random when a client try to negotiate with a TLS 1.3 server an older TLS version (or SSL 3.0). All the events regarding SSL/TLS described above are summarized in the timeline in Fig. 5.1.

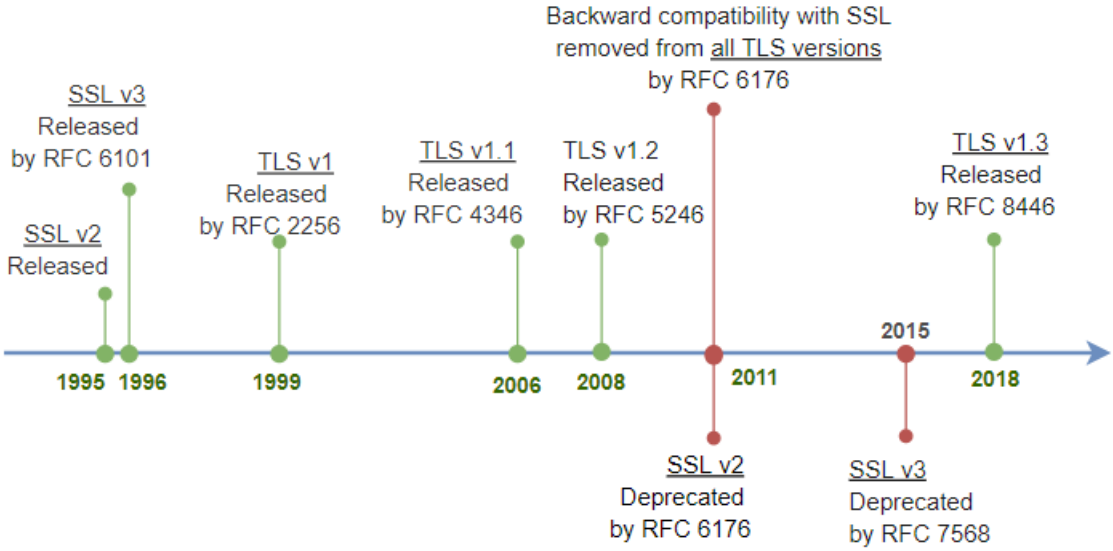


Figure 5.1. Timeline of SSL/TLS versions

Since TLS versions 1.0 and 1.1 were superseded by TLS 1.2 in 2008 and subsequently replaced by TLS 1.3 in 2018, recently IETF published RFC 8996 [35] (2021) in order to deprecate TLS 1.0 and TLS 1.1. There are also several technical reasons for deprecating these versions. They require the implementation of older cipher suites that are no longer desirable for cryptographic reasons. There is a lack of support for current recommended cipher suites, especially authenticated encryption with associated data (AEAD) ciphers, which were not supported prior to TLS 1.2. The integrity of both TLS 1.0 and TLS 1.1 depends on a running SHA-1 hash of the exchanged messages. This makes it possible to perform a downgrade attack on the handshake by an attacker able to perform $2\hat{7}7$ operations, well below the acceptable modern security margin. Similarly, the authentication of the handshake depends on signatures made using a SHA-1 hash or a concatenation of MD5 and SHA-1 hashes, allowing the attacker to impersonate a server when it is able to break the severely weakened SHA-1 hash. Neither TLS 1.0 nor TLS 1.1 allows the peers to select a stronger hash for signatures in the ServerKeyExchange or CertificateVerify messages, making the only upgrade path the use of a newer protocol version. In addition, the support for four TLS protocol versions increases the possibility of misconfiguration. As with any other protocol upgrade, TLS is seen as more secure than SSL 3.0 because of added measures to block exploitation and mitigate vulnerabilities on each version. TLS is widely used throughout the web today, and is the top choice for transaction security.

5.2 DTLS history

DTLS is a communication protocol designed to protect data of application protocols based on datagram transport. It prevents eavesdropping, tampering or message forgery of the communicating applications having an unreliable connection between them. The DTLS protocol provides similar communication security guarantees in an unreliable transport channel as that provided by TLS protocol to a reliable transport channel. DTLS evolution happened with different versions corresponding to the different versions of TLS evolution as a result of modifications over the previous versions. These improvements were regarding enhanced security or a patch for the reported vulnerability or support for different protocols. The DTLS versions released so far are: DTLS 1.0 based on TLS 1.1, specified in IETF RFC 4347 [36] (2006), DTLS 1.2 based on TLS 1.2, described in IETF RFC 6347 [4] (2012) and DTLS 1.3 based on TLS 1.3, specified in RFC 9147 [37] (2022). DTLS 1.1 is left to harmonize DTLS version numbers with TLS versions. The different versions gradually improved in ciphersuite support corresponding to the increasing number of their versions thereby making them stronger against the cryptographic brute force attacks. In particular

the standardization of version 1.2 of the protocol involved different modifications with respect to DTLS 1.0 in order to match the security features provided by TLS 1.2, for example the addition of AEAD ciphers. Since TLS 1.3 had introduced a large number of changes with respect to TLS 1.2, in 2022 DTLS protocol was further modified with the publication of the most recent version (1.3). First of all a new handshake pattern was introduced, which led to a shorter message exchange. The support for weak and old cryptographic ciphersuite was removed and only AEAD ciphers are supported with DTLS 1.3. Moreover a new session resumption mechanism was introduced and the key derivation mechanism was updated.

5.3 QUIC History

QUIC is a protocol introduced by Google in 2012 in order to reduce the latency of transport between two endpoints and provide security to the communication. Chrome was equipped with the support for QUIC in 2013. The initial design idea of QUIC came from SPDY, a protocol later standardized as HTTP/2, which has as main feature the possibility to create different streams within the same TCP connection (multiplexing). QUIC is able to solve the problem of HOL blocking that affects TCP, while maintaining at the same time the possibility of stream multiplexing. The idea of Google was to create a protocol that runs over UDP and provides high security as TLS with low latency connectivity. In addition Google QUIC must implement other features to make up for the lack of a reliable transport protocol in its stack: Among them there are packet pacing to reduce packet loss, packet error correction to reduce retransmission latency and a pluggable congestion control mechanism. Moreover the design of QUIC protocol by Google includes a connection identifier to reduce reconnections for mobile clients.

The QUIC crypto protocol is the part of QUIC that provides transport security to a connection. With QUIC crypto protocol, when the client has cached information about the server, it can establish an encrypted connection with no round trips. QUIC handshakes should be around 5 times more efficient than common TLS handshakes (2048-bit RSA) and at a greater security level.

QUIC has to deal with the problems of source address spoofing and replay protection. The IP address spoofing problem is handled by issuing the client, on demand, a source-address token. This is an opaque byte string from the client's point of view. From the server's point of view it's an authenticated-encryption block (e.g. AES-GCM) that contains, at least, the client's IP address and a timestamp by the server. The server will only send a source address token for a given IP to that IP. Receipt of the token by the client is taken as proof of ownership of the IP address in the same way that receipt of a TCP sequence number is. Clients can include the source address token in future requests in order to demonstrate ownership of their source IP address. If the client has moved IP addresses, the token is too old, or the client doesn't have a token, then the server may reject the connection and return a fresh token to the client. But if the client has remained on the same IP address then it can reuse a source-address token to avoid the round trip needed to obtain a fresh one.

QUIC doesn't provide replay protection for the client's data prior to the server's first reply, because providing replay protection without input from the server is fundamentally very expensive. It's up the application to ensure that any such information is safe if replayed by an attacker.

In 2015 Google brought QUIC to the IETF that then created a working group for the standardization of the UDP-based, stream-multiplexing, encrypted transport protocol based on previous Google's implementation and deployment experiences. QUIC addresses the HOL blocking problem with stream multiplexing in the transport layer, reduces the connection establishment latency combining the cryptographic handshake with the transport one, providing 1-RTT handshake for first-time connections as well as 0-RTT handshake for session resumption exploiting TLS 1.3.

5.4 TLS and DTLS attacks

Since its first version in 1999, TLS has obtained popularity and widespread usage and year after year TLS became the most adopted solution between all the security protocols. Given this important growth, during years attackers worked to find vulnerabilities, develop exploits and mount

attacks against TLS. As a result, TLS was often updated and had a significant evolution through different versions, from 1.0 to 1.3.

Over the last years, various types of attacks have been designed to take advantage of properties of TLS protocol. There are different targets for the attacker within the protocol design implementation: from weaknesses in the handshake protocol, passing through vulnerabilities of some block cipher mode of operations or some stream ciphers, till problems with compression algorithms and weak export ciphers. In the next sections a lot of known attacks against TLS protocol will be described, analysing the vulnerabilities exploited, the working principles, the feasibility and complexity, and also possible countermeasures. For each of the analysed attack, the versions of the protocol that are vulnerable will be indicated.

5.4.1 Attacks on Handshake Protocol

The Bleichenbacher attack [38] is applicable during a TLS handshake (up to version 1.2), when key-exchange take place using RSA algorithm and the padding used is PKCS#1 v1.5. Note that during TLS session setup with RSA key exchange, client chooses a random 48 bits number (2 bits of protocol version and 46 random bits), padded following the PKCS encoding scheme (PKCS padded data must always starts with 0x00 0x02 with public key operations). The result is then raised to public exponent e modulo n. After decryption, the receiver checks the correct alignment of data and verifies if the plaintext starts with 0x00 0x02, otherwise the packet is discarded. Then all the bytes are skipped until 0x00 is found.

Suppose an attacker, that doesn't know the message M but only the public key (e,n), gets the ciphertext C ($M^e \bmod n$). The attacker then multiplies this ciphertext with a chosen s, obtaining essentially the encryption of $M * s$ (for the homomorphic property of multiplication), and it will repeat this procedure, changing every time the value of s, until the result is accepted by server.

$$C' = C * s^e \bmod n$$

When server accept this C' it means C' after decryption starts with 0x00 0x02 and C' is a valid encryption for $M * s$ with PKCS padding. Consequently:

$$M = C' * s^{-}1 \bmod n$$

In order to avoid this kind of attack, the first solution is to avoid using RSA for key exchange, choosing DH or ECDH instead, that in addition provide also perfect forward secrecy. Give this consideration, TLS 1.3 could represent a good solution, since RSA cannot be used for key exchange in the latest version of the protocol. In addition the use of OAEP for padding instead of PKCS is a mitigation for this attack, as well as not informing the client about failure when pre master secret decryption failed.

All versions of TLS up to 1.2, and SSL v3 before it, support optional renegotiation. After the initial handshake is completed the client can request renegotiation by sending a new *ClientHello* message in the current record layer. Also the server can do that by sending a *HelloRequest* message in the record layer, which triggers the client to send a new *ClientHello* message. In November 2009, Ray and Dispensa described a man-in-the-middle attack [39] that exploits how certain applications based on TLS process data across renegotiations. The attacker delays client initial *ClientHello* towards the server and instead establishes its own TLS session with the server. Then it transmits a message m_0 over that record layer and passes client initial *ClientHello* to the server, that views this as a valid renegotiation and responds accordingly. The attacker relays the handshake messages between client and server, who eventually establish a new record layer, to which the attacker has no access. The client then transmits a message m_1 over the record layer just created with the server. Here comes the problem of some application (including HTTPS and SMTPS) of TLS-protected data processing. Indeed these applications concatenate m_0 and m_1 and treat them as coming from the same party in the same context. For example, if the request sent by the attacker ends with a custom "ignore" header prefix, it causes the first line of the request from the client to be discarded by the server. The other parts of the message sent by

the client, including any cookie or other authentication/authorization headers, are kept with the effect of authorizing the attacker's request.

Two countermeasures to this attack were standardized by the Internet Engineering Task Force (IETF) TLS working group with RFC 5746. These countermeasures, named Signalling Ciphersuite Value (SCSV) and Renegotiation Information Extension (RIE), are applicable to SSLv3.0 and TLS versions 1.0-1.2 and were adopted by major TLS implementation providers and web browsers and servers, including Apache, Apple, Google, Microsoft, Mozilla, and OpenSSL. In RIE, the parties include the *Finished* message from the previous handshake in a ClientHello/ServerHello extension, correlating in this way the two handshakes. SCSV is a slight modification that is more compatible with buggy implementations. With the standardization of TLS 1.3 the renegotiation was removed, consequently this attack cannot be executed against the latest version of the protocol.

During a 3SHAKE attack [40], a TLS client connects to a malicious server and presents a client credential. Then the server can impersonate the client at any other server that accepts the same credential. Concretely, the malicious server performs a man-in-the-middle attack on three successive handshakes between the honest client and server, and succeeds in impersonating the client on the third handshake. The attack proceeds in three steps, shown in Fig. 5.2. In the first phase, the target of the attacker A, positioned in between a client C and a server S, is to create two sessions that share the same key materials and session parameters (SID, master secret, client random and server random), but with different server certificates and *Finished* messages. To this purpose, A receives the request to establish a connection from C and at the same time it connects as a client to S using the same client random sent previously by C. Then it answers to C forwarding the server random and the SID received by S.

The second step of the attack exploits a vulnerability in the session resumption mechanism of TLS. Indeed session resumption on a new connection happens with an abbreviated handshake that only verifies that the client and server share the same master secret, ciphersuite, and SID, but it does not authenticate again the client and server identities. At this point, when C reconnects to A and asks to resume its previous session, A does the same with S. Since the two sessions share relevant parameters and keying material, A can simply forward the handshake messages unchanged between C and S. At the end of the abbreviated handshake, the two connections have now in common also the *Finished* message.

The two previous steps are used to prepare the real attack, that is implemented in the last phase. Indeed the fact that the two connections just created share the same *Finished* message, makes the attacker possible to exploit the renegotiation mechanism of TLS, that gives the possibility to switch an open connection from one mode to the other. This is typically used to change a connection only server authenticated into a mutual authenticated connection. In addition an extension was introduced, named Renegotiation Indication, to bind the renegotiation handshake to the previous handshake, by having the client present the old *Finished* message. During the third step, S asks for renegotiation with client authentication on its connection with A, in response to a restricted resource request by A. Then, A forwards the renegotiation request to C, that authenticates with its client certificate. A simply forwards all messages from C to S and back, consequently both the handshake completes successfully, since the expected Renegotiation Indication extension values on both connections are the same.

During the renegotiation handshake, C receives a certificate for S even though it was expecting to be connected to A. C silently allow the server certificate to change, because of a vulnerability in the renegotiation mechanism: during the process both the server and client certificates can change, but no definitive guidance is given to applications on how to deal with such changes. At the end of renegotiation, A cannot read or send messages on both the created connections, because it no longer knows the connection keys or master secret. However, its prior messages on both connections may well be prefixed to the messages sent after renegotiation.

There are other two potential cyberattack scenarios related to the TLS renegotiation process, especially if it's client initiated. A man-in-the-middle attack (injection vulnerability) that inserts malicious data into HTTPS sessions through an unauthenticated request. By doing this, the attacker may run commands with the credentials of an already authorized user and even gather other users' credentials. The other scenario regards a DoS attack, that starts hundreds of handshakes from clients for the same TCP connection.

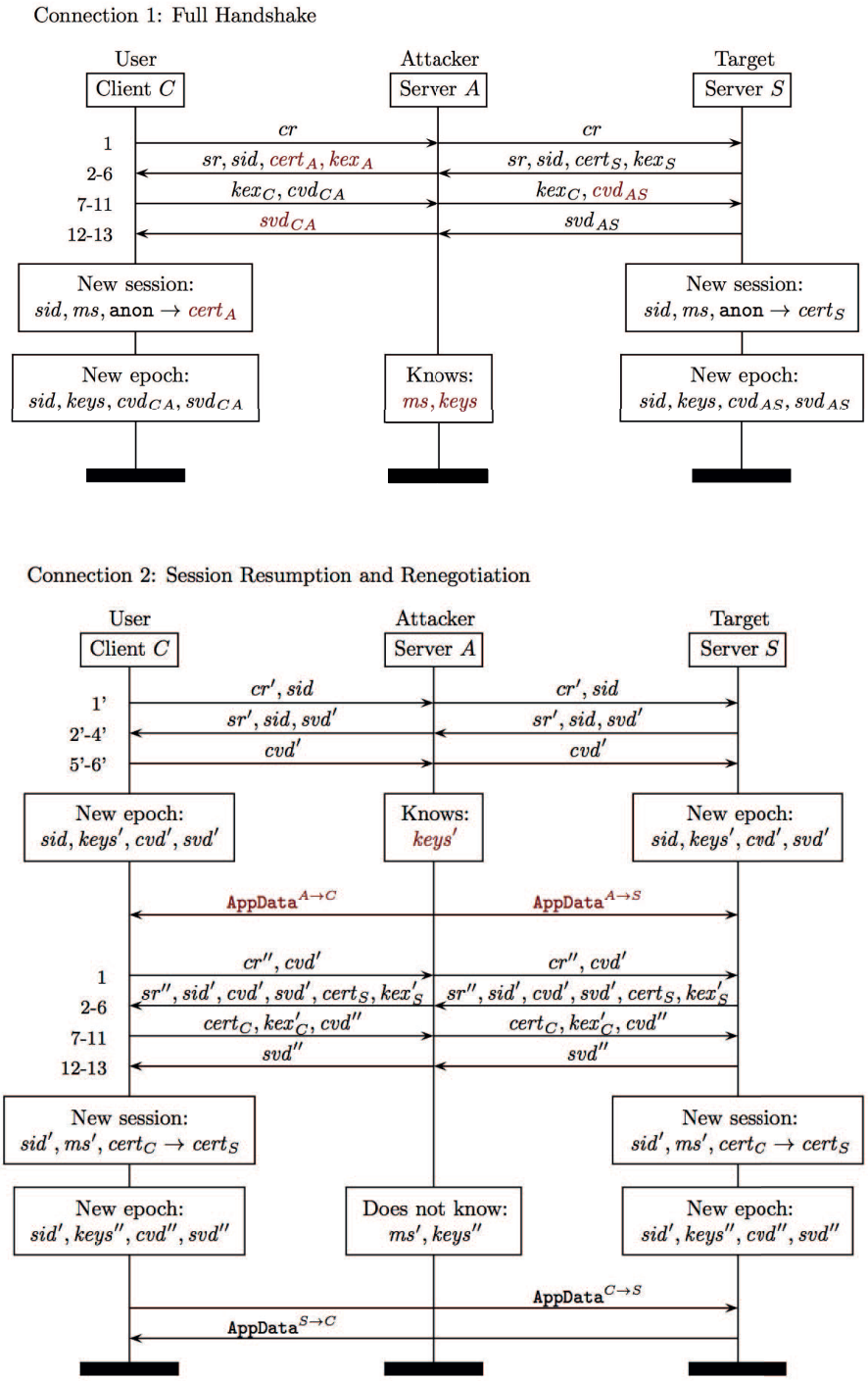


Figure 5.2. The 3SHAKE attack (source: [40])

This attack can be conducted against all the versions of TLS up to version 1.2. The reason TLS 1.3 is not vulnerable is that the renegotiation has been removed in its standardization. The countermeasures for this attack are the same cited above for the renegotiation attack: to avoid this attack is necessary to bind the master secret to the full handshake or to bind the session resumption handshake to the original full handshake.

5.4.2 Attacks on CBC Mode

BEAST [41] attack allows a man-in-the-middle to decrypt HTTPS-protected session cookies, simply by injecting Javascript into the user's web browser. The attack exploits a vulnerability present in all versions of SSL and TLS 1.0, regarding the way long messages are encrypted using block ciphers. TLS 1.0 uses a block cipher to encrypt data with Cipher Block Chaining mode of operation. CBC mode is essentially a means of encrypting long messages in short fixed-size blocks. When CBC mode is used, the message is first divided into even-sized blocks. In order to encrypt a block of plaintext, it is before XORed with the previous block of ciphertext. For the first block of plaintext a random Initialization Vector (IV) is used. TLS designers made an improvement for the CBC mode, to avoid having to generate a new IV for each message. Indeed the protocol uses the last block of the previous ciphertext message that was sent as an IV for the current message.

If we assume that an attacker A is sniffing the encrypted data sent by the browser, then the IV that will be used for the next message can be obtained. If the attacker can identify the block of ciphertext C (the encryption of M that maybe contains a cookie) that he wants to decrypt, he can easily obtain the block of ciphertext C' that immediately precedes C.

$$C = E_k(M \oplus C')$$

If the attacker is able to transmit any chosen block of data M*, it will produce a ciphertext C* that can be sniffed. The attacker can try to make some guesses about the value of the message M (that it wants to decrypt) and can generate M* as follows:

$$M^* = IV \oplus M \oplus C'$$

The TLS layer will encrypt it as follows:

$$C^* = E_k(IV \oplus M^*) =$$

$$E_k(IV \oplus IV \oplus M \oplus C') =$$

$$E_k(M \oplus C')$$

If the attacker guessed correctly, the ciphertext C* that comes out of the browser will be identical to the captured ciphertext C that it wants to decrypt. The problem with this attack is that it works only if the number of guesses for the value M is small. But there is an optimization for this attack which consists in a particular alignment of the TLS messages. For example, in a situation with block size equal to 16 bytes, it could be useful for the attacker that a 15 byte padding is inserted before the cookie when it is sent by the browser, so that the first byte of the cookie is situated in the last byte of the first block sent. Using this optimization the number of necessary guesses is reduced from 2^{128} to $16 * 2^8$, since the attacker is guessing the cookie one byte at time. In order to discover the entire cookie, this operation must be repeated one time for each byte, reducing each time the size of the known padding.

Initially, switching to the RC4 stream cipher was recommended to quickly prevent the vulnerabilities associated with CBC in TLS. But later, in 2013, many vulnerabilities were found also in RC4 stream cipher. Consequently the simplest and most efficient way of preventing a BEAST attack is to upgrade the version of TLS protocol to the most recent ones (1.2 and 1.3), disabling support for TLS 1.0 and 1.1, as well as SSL on the server.

LUCKY 13 [42] is an attack against TLS protocol that was found in 2013 by Nadhem AlFardan and Kenny Paterson of the Information Security Group at Royal Holloway of the University of London. It's a cryptographic timing attack that can be used against the implementations of the protocol that use Cipher Block Chaining (CBC) mode of operation. This attack affects SSL 3.0 and all the versions of TLS up to 1.2. The attack allows a man-in-the-middle to recover plaintext from a TLS connection when CBC mode encryption is used, exploiting a bug in the way that TLS data decryption works with CBC. LUCKY 13 uses the same attack mechanism as the padding oracle attack.

In TLS Message Authentication Code (MAC) is used to authenticate and to provide integrity of the message. When a message has to be sent, a MAC is applied to the plaintext, and then up to 255 bytes of padding are added to make the message size a multiple of the cipher block size (8 or 16 bytes). This message block is finally encrypted using the block cipher with CBC mode. The receiver takes the encrypted block, decrypts it and XORs it with the previous ciphertext block. After decryption, the validity of the padding is checked and then, in case of success, it is removed and integrity of data is checked against the calculated MAC.

LUCKY 13 attack relies on a difference in processing times between TLS messages. Indeed, because of the way that HMAC-SHA1 works, it takes four units of CPU time to checksum the first 55 bytes of data, plus an extra time unit for every additional 64 byte chunk. If two bytes are truncated from an encrypted packet, there is a chance of approximately one in 65000 that the server would end up wrongly thinking the message was padded, removing the padding and validating what was left. This would take 4/5 of the time of processing a packet that wasn't modified. This would happen because the tweaked packet had tricked the server into checksumming 55 bytes or less of data. By systematically trying all possible two-byte tweaks (2^16 if the block size is equal to 16 bytes, such as for AES CBC), and assuming perfect timing measurement, it could be possible for an attacker to recover two of the encrypted bytes.

A LUCKY 13 attack requires about 2^{23} TLS sessions to collect a whole block of TLS-encrypted plaintext in its most basic version. With some optimizations and in the best circumstances, the number of needed TLS sessions to recover one plaintext byte can be reduced to 2^{13}. In order to execute the attack in a precise and efficient manner, the attacker must be placed in the same network as the web server, in order to reduce any noise and perform the timing attack.

One possible way to avoid LUCKY 13 attack is to use a stream cipher instead of a block cipher, avoiding this way the need for plaintext padding. Another solution could be to implement a Encrypt-then-MAC mechanism instead of a MAC-then-Encrypt, so that the quantity of data to be checksummed does not depend on the plaintext. The best countermeasure to LUCKY 13 attack is to use authenticated encryption algorithm, like AES-GCM. To this purpose the best idea is to upgrade the version of TLS protocol to TLS 1.3, that always uses authenticated encryption.

The POODLE attack [43] (Padding Oracle on Downgraded Legacy Encryption) exploits a vulnerability in the SSL 3.0 protocol, that let an attacker eavesdrop on communication. To work with legacy servers, many TLS clients implement a downgrade mechanism, that consists in offering the highest supported protocol version during the first attempt of handshake and retrying in case of failure with earlier protocol versions. This downgrade can be triggered by network glitches or by an active attacker that controls the network between the client and the server and interferes with any attempted handshake.

Encryption in SSL 3.0 uses either the RC4 stream cipher, or a block cipher in CBC mode. RC4 is well known to have biases, meaning that if the same secret (such as a password or HTTP cookie) is sent over many connections and thus encrypted with many RC4 streams, more and more information about it will leak. POODLE is not exploiting RC4 vulnerabilities, but it is an attack against CBC encryption as used by SSL 3.0. The problem is that when a block cipher in CBC mode is used for encryption in SSL 3.0 the integrity of padding cannot be fully verified during decryption, because padding is not deterministic and not protected by MAC.

If L is the block size in bytes, the weakness is the easiest to exploit if there's an entire block of padding, which consists of L-1 arbitrary bytes followed by a single byte of value L-1. To process an incoming ciphertext C divided in n blocks ($C_1 ... C_n$), given an initialization vector C_0, the receiver first determines $P_1 ... P_n$ as:

$$P_i = D_K(C_i) \oplus C_{i-1}$$

Then checks and removes the padding and, after that, verifies the correctness of the MAC. If there's a full block of padding and an attacker replaces C_n with any earlier ciphertext block C_i from the same encrypted stream, the ciphertext will still be accepted if the final byte of $D_K(C_i) \oplus C_{n-1}$ results to contain the value L-1, otherwise i will be rejected. In this situation a padding oracle attack can be executed.

In the web setting, this SSL 3.0 weakness can be exploited by a man-in-the middle attacker to recover HTTP cookies. To prevent the POODLE attack, SSL 3.0 support must be disabled from

both servers and browsers, and only TLS 1.2 or higher version must be supported. In addition, TLS_FALLBACK_SCSV must be enabled. This protocol extension guarantees that during a negotiation, the protocol never falls back to earlier protocol versions that are below the highest SSL or TLS version supported by the server. Implementing TLS_FALLBACK_SCSV prevents a downgrade attack to force the use of the vulnerable SSL 3.0, that can be used only when an existing legacy system is involved.

5.4.3 Attacks on Compression algorithm

Compression Ratio Info-leak Made Easy (CRIME [44]) is an attack on TLS that was developed by researchers Juliano Rizzo and Thai Duong. CRIME is a side-channel attack that exploits a vulnerability in TLS compression applied to HTTPS requests. Using CRIME an attacker can recover secret information, such as session tokens. This attack can be executed on web sessions protected by TLS that adopts DEFLATE or gzip as compression method. The purpose of compression in a session is to reduce network congestion or the loading time of web-pages.

Compression is a mechanism to use a reduced number of bits to transmit or store a big amount of data. DEFLATE is one of the most used algorithms for TLS compression and consists of two sub algorithms: Lempel-Ziv coding or LZ77, and Huffman coding. LZ77 is used to eliminate the redundancy when repeating sequences occur, while Huffman coding is used to eliminate the redundancy if the same symbol is present several times. It works by simply replacing the repeated strings with the reference to the last occurrence of the same string as (distance, length). Distances are limited to 32K bytes, and sequence lengths are limited to 258 bytes.

When TLS compression is used, it is applied on all the transferred data, as a long stream. In particular, when used with HTTP, compression is applied on all the successive HTTP requests in the stream, including the header. CRIME is a brute-force attack that works by leveraging a property of compression functions, and noting how the length of the compressed data changes. To realize a CRIME attack, an attacker can abuse a weakness in the TLS protocol compression mechanism to decrypt the HTTPS cookies set by a website.

The size of the content of an HTTPS request is equal to the length of the encryption of the result of the compression of header and body of the request. The important thing to underline is that an attacker can know the compressed content length, even if the content is encrypted, and it's very useful to mount the attack Indeed the attacker, by means of JavaScript, can modify the request of the client inserting in the header a string that reproduces the website cookie, already present in the original request (e.g. "secretcoookie=..."). When the compressed content diminishes in size, it's likely that the injected content has matched some part of the secret content the attacker wants to discover. The idea is to change every time the input string in the fake request and observe the change in length to discover the session cookie.

If the attacker knows the location of the secret, an optimization of this attack can be executed. Since the maximum size of a TLS record is known and it's equal to 16kB, the attacker can try to pad in the proper way the request path, so that there is only one unknown byte to guess in the first record. The attacker must change the value of the unknown byte in the request until it's discovered. In order to recover the entire session cookie, this procedure must be repeated every time for each byte of the secret value to guess, reducing the padding of one byte every time, leaving only one byte to guess for each attempt.

This attack can be conducted against each version of TLS up to 1.2 that uses a compression method. The only useful countermeasure that can be introduced to avoid this kind of attack is to disable TLS compression. TLS 1.3 is not vulnerable to this attack because compression is no more allowed in this version of the protocol.

Timing Info-leak Made Easy (TIME [45]) is a chosen plaintext attack on HTTPS responses that was developed by Tal Be'ery and Amichai Shulman of Imperva. TIME starts from timing information differential analysis in order to discover the compressed payload size and then applies the same attack model of CRIME. In order to launch a TIME attack and break the TLS encryption, an attacker need to have the control only on the plaintext. There are two improvements introduced with TIME attack model with respect to CRIME model. The first difference is that

TIME doesn't require the presence of a man-in-the-middle to be executed, while the second is that TIME shifts the attack target from HTTP requests to HTTP responses. The problem is that TLS compression is used for HTTP requests but it has been disabled in most browsers and servers. On the other hand, for responses HTTP compression is used, as a best practice for speed and bandwidth gains, and it can be exploited to mount an attack.

The target of the attacker is to force the length of the compressed data to overflow into an additional TCP packet and then pad the compressed data to align to the amount of TCP packets allowed within the TCP window. In this way if an incorrect guess occur, despite the compression, another packet has to be sent and an additional full round trip time is necessary, since the ACK for the previous packet must be waited. This situation gives the possibility to measure and detect a significant delay.

The attacker can try to recover the user secret that has been sent to the server (e.g., "secret element = unknown data"), sending to the server multiple requests and guessing it byte by byte. The attacker injects malicious code capable of sending multiple requests with attacker controlled data to the target website. Since the user inputs get reflected within the server response, the attacker has only to measure the shortest response time for every character in the specific position of the payload which will happen only in the case of a correct guess. Indeed if payload length is exactly on the window boundary and the byte the attacker is trying to guess doesn't match the secret value, the attacker can measure a significant delay due to the additional RTT. When the attacker-controlled data matches the correct guess, the response time does not consist of additional RTT. Every time a byte of the secret is recovered, the padding is adjusted to maintain 1 extra byte more than the boundary. By repeating this procedure for all the bytes, the attacker can completely recover the secret with no eavesdropping. TIME attack can be used to find out all sensitive information that is reflected in the response, not only the session cookie.

This timing attack does not directly exploit any vulnerability in TLS, but the timing information leaks. Adding random timing delays to the decryption could be an idea for slowing down the attack, but it doesn't work in order to prevent it. Indeed an attacker who can gather enough samples can average many observations which make the randomness disappear. Moreover applications should have a strict restriction on the reflection of user input in the response.

Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext (BREACH [46]) is an attack demonstrated by Yoel Gluck, Neal Harris, and Angelo Prado at BlackHat USA 2013. As well as CRIME attack, BREACH exploited the compression and encryption combination used to interact with users and web-servers. As a difference with respect to CRIME, BREACH has as target HTTP compression, that is applied only to the body of responses, instead of TLS compression. The algorithm used for compression is DEFLATE, composed by LZ77 coding and Huffman coding, as discussed before. BREACH works by attacking the LZ77 compression while minimizing the effects of Huffman coding, reducing in this way the number of false positives.

To be vulnerable to this kind of attack, a web application must:

- Be served from a server that uses HTTP-level compression;
- Reflect user-input in HTTP response bodies;
- Reflect a secret (such as a CSRF token) in HTTP response bodies.

The goal of the attacker is to recover the value of the CSRF token and to this purpose a string containing a guess of it (e.g. "CSRFtoken=...") is inserted in the HTTP request. Then the attacker has only to measure the response size. Indeed, since the response will reflect the real CSRF token and the attacker input in the HTTP response body, if after compression the length will be smaller it means that the guess matches at least a part of the secret value. The idea is to change input and measure and compare lengths to guess the correct secret value, repetitively until the whole secret is recovered.

It is important to notice that BREACH does not require TLS-layer compression, but it focuses on HTTP compression. Consequently the TLS version used is not relevant and the attack works against any cipher suite. Moreover disabling TLS compression does not affect the possibility of

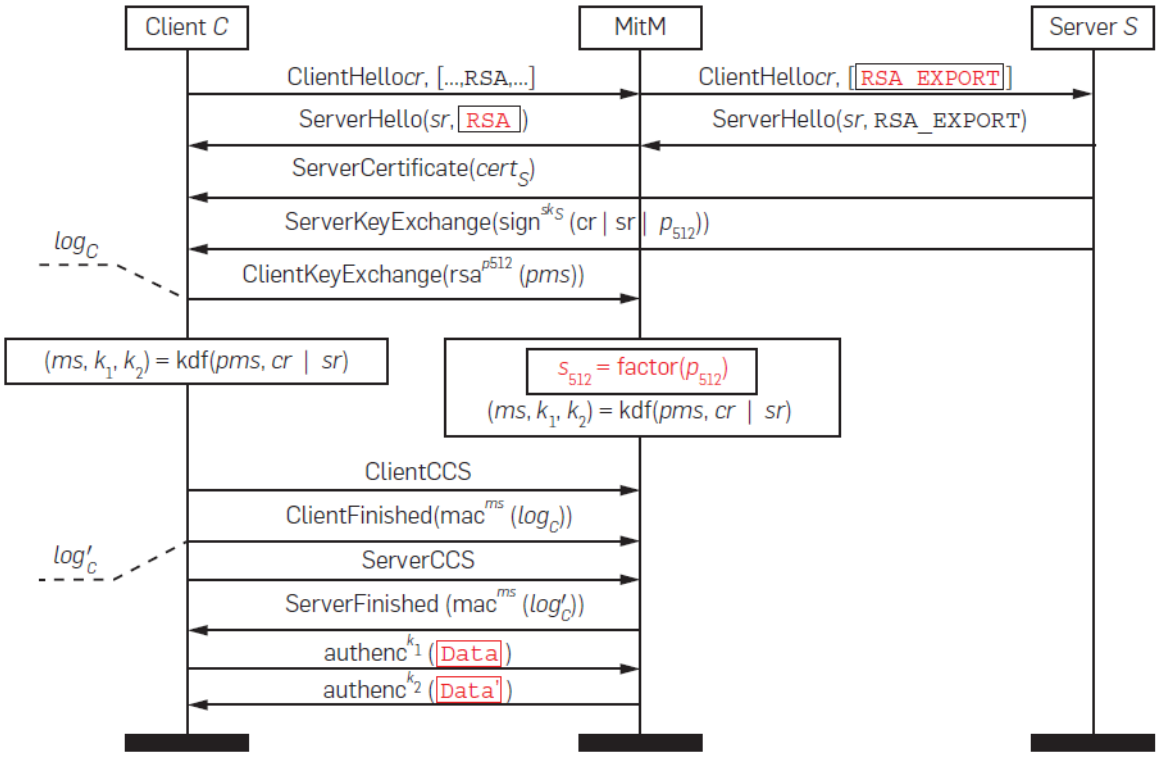


Figure 5.3. The FREAK attack (source: [47])

a BREACH attack. The attack is simple to execute if a stream cipher is used, while if a block cipher is chosen, additional work must be done to align the output to the cipher text blocks. There are different solutions that can be used to mitigate this attack: separating ant secret from user input, hiding traffic length by including random bytes to responses, randomizing token value in every response and limiting the rate of requests.

5.4.4 Attacks exploiting export ciphers

in 2015 a new TLS vulnerability, called FREAK attack [48, 47], was discovered. It allows an attacker to intercept HTTPS connections between vulnerable clients and servers and force them to use weakened encryption, which the attacker can break to steal or manipulate sensitive data.

The attack starts with a vulnerable client sending the client random, the list of supported ciphers and supported curves to the server within the *ClientHello* message, as depicted in Fig. 5.3. The attacker, that must be a MitM, substitutes the list of supported ciphers with a weak export cipher (40 bit key). The server receives the message modified by the atttacker and answers with its *ServerHello* message, containing server random and the selected cipher. Given the change applied by the attacker to the first client message, the server is forced to choose the weak cipher. If server is badly configured, it will accept to choose the export cipher and then it will send also the remaining messages to complete the handshake, such as *Certificate*, *ServerKeyExchange* and *CertificateVerify*. In this way client generates a 40-bit key that will become the shared key with the server. At this point the client answers the server with *ClientKeyExchange*, *ChangeCipherSpec* and *Finished* messages, protected with the weak key. The *Finished* message is computed over what the client sent, including the good list of ciphers. The attacker has to intercept the *Finished* message sent by the client, brute force the weak key, through 2^{40} attempts, recompute the message according to the attacker changes, and relay it to the server. In this way the server won't be able to detect the attack and in turn will generate the weak shared key. The attacker will be able to

sniff all the traffic and to create fake messages since weak keys are used also for authentication and integrity.

In order to prevent this kind of attack the only solution is to disable support for export versions of cipher suites and all other ciphers whose security is questionable.

DROWN [49] is a cross-protocol attack that exploits a vulnerability in SSL 2.0, an old protocol version that has several serious flaws. SSL 2.0 has been deprecated for more than 15 years and has been formally prohibited in 2011. The big problem arises when "export" ciphersuites, characterized by an encryption strength of about 40 bits, are supported. If the target of the attack is a server that uses SSL 3.0 or TLS protocol, the attack can anyway be executed if the support for SSL 2.0 is still provided and both protocols use the same public key. If a DROWN attack is successful, it may lead to the disclosure of important and sensitive data, that can be read and stolen by an attacker.

The attack proceeds following different steps. The attacker observes an encrypted TLS session that uses RSA key exchange in order to decrypt it. It's an important condition for the execution of this attack since it cannot be executed against a connection that uses DHE or ECDHE key exchange. There is a probability of about 1/1000 that the session observed is suitable for the attack, so the attacker will need to gather about a thousand encrypted sessions in order to succeed in decrypting one of them.

Another assumption that has to be made for the applicability of DROWN attack is that the TLS server uses the same RSA private key for a SSL 2.0 system. At the second stage of the attack, the attacker creates multiple connections to the other system using the cross-protocol vulnerability. The attacker begins a SSL 2.0 handshake with that system, asking for a 40-bit export ciphersuite and using as *ClientMasterKey* message a value derived from the one that the attacker wants to decrypt. The attacker, observing the server response, brute-forces the 40 bit value generated by the server when it received the value sent by the attacker. At the end, after about 40,000 such attempts, the attacker can recover the master secret and obtain the session key. At this point the attacker can decrypt sensitive data that were sent during previous recorded sessions. The cost of DROWN is about 2^{50} operations, since the attacker need to run a 40 bit brute force attack for each one of the 1000 SSL 2.0 connections established with the server.

The only countermeasure that can be adopted to avoid the DROWN attack is to remove the support of SSL 2.0 on the server. In addition the private ley of the server must not be used in any other system that supports SSL 2.0 connections, such as web servers or email servers.

Logjam [50] is an attack against TLS (up to version 1.1) protocol implementations that use Diffie-Hellman key exchange with 512 to 1024 bit keys. The first target of this attack is to downgrade a TLS connection from non-DHE_EXPORT to a DHE_EXPORT. At this point the attacker can exploit the weakness of 512-bit export-grade cryptography in order to maliciously get access to the connection and inject harmful data into it. In addition, if there were old sessions that used the same DH parameters, the attacker can decrypt them.

The security of the Diffie-Hellman key exchange is related to the complexity of the solution of the discrete logarithm problem. This problem is thought to be a mathematically intractable, at least when Diffie-Hellman is implemented in cryptographically strong groups (2048 bits for DHE parameters), while the situation is different for export variant of DHE that involves prime numbers of only 512 bits. Another important point makes easier to solve the discrete logarithm problem when DHE_EXPORT is used. Indeed with this algorithm, in most cases the same pre-generated prime numbers are used. For this reason the discrete logarithm problem is even easier to break. Solving this problem, a man-in-the-middle attacker recover all the DH parameters and take over client connections towards the server. Since the majority of servers use the same built-in Diffie-Hellman parameters, researchers have theorized that an attacker can use the complex and expensive precomputation for cracking one server on numerous servers. Fig. 5.4 shows how the attack works.

In order to avoid Logjam attack, it's necessary to use only strong cipher suites with 2048-bit or larger primes, preventing the use of parameters of smaller dimension. Clients have to be set to reject Diffie-Hellman primes of less than 1024-bit. The latest versions of TLS protocol (1.2 and 1.3) prevent the use of export ciphers and weak prime numbers, consequently are not vulnerable to this kind of attack.

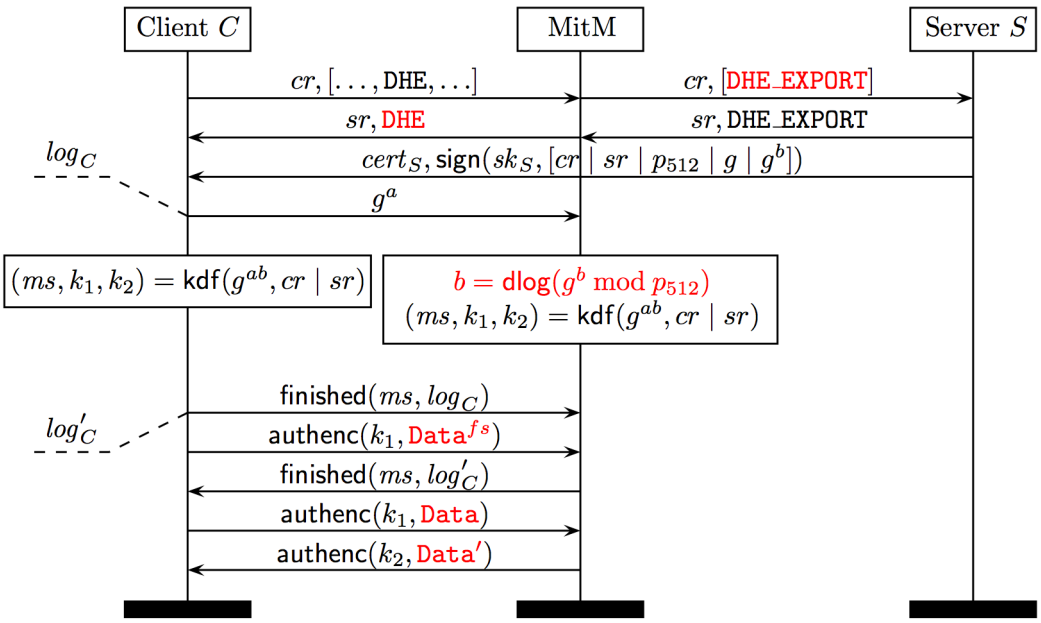


Figure 5.4. The Logjam attack (source: [50])

5.4.5 Attacks on RC4 algorithm

RC4 is a stream cipher developed by Ron Rivest in the late 1980s. The first phase of RC4 is the key scheduling algorithm (KSA), that takes an initial array and initializes it to values 0 to 255. For each index of the array a shuffling occurs that mixes in the key. The second phase of the stream cipher is the pseudo-random generation algorithm (PRGA), that receives as input the output of KSA and shuffles again the array. The ciphertext is obtained XORing the plaintext with the output of the PRGA that is the keystream.

Several weaknesses affecting RC4 stream cipher have been found over the years. A research by Bernstein, Paterson, Poettering and Schuldt [51] explained two possible ways to mount a plaintext recovery attack on RC4. The first one of these two attacks, a broadcast attack, recovers the plaintext when it is sent repeatedly in many different sessions, while the second can lead to plaintext recovery even if it is sent encrypted repeatedly in the same session.

The broadcast attack is based on strong biases in the first 257 bytes of encryption. If the same plaintext is encrypted using unique keys from 2^{28} to 2^{32} times, the attacker will be able to recover roughly the first 200 bytes of plaintext. For position of the plaintext to recover, the full set of observed ciphertexts is used. The attacker tries each one of the 256 possible values for each byte, XORing it with each ciphertext in order to obtain the PRGA as output. It then calculates which plaintext candidate resulted in PRGA outputs which most closely matches the known PRGA bias for that position.

The other attack exploits long-term biases in RC4 and makes possible to recover half of a 16-byte secret value, analysing $6 * 2^{30}$ encryptions of the same plaintext message with a single key. As a difference with respect to the broadcast attack, this attack can be executed to recover plaintext also from a single encryption stream in which the same plaintext is sent repeatedly. The vulnerability in this case includes transitional biases in the PRGA stream that recur at fixed positions in the stream. The attacker starts with a known plaintext byte that repeats at a fixed position and tries to find a candidate for the plaintext byte that repeats at the next position. The candidate that most closely matches the transition bias of the PRGA is selected. After that, repeating this process, the attacker can recover the next unknown plaintext byte.

Different countermeasures have been proposed, but they don't work to completely protect against this kind of attacks. Indeed, for example, discarding the initial keystream bytes of output

and randomizing HTTP requests can be bypassed by an attacker. The only recommended working solution is to avoid using RC4, preferring instead patched versions of CBC based algorithms or Authenticated Encryption based algorithms.

Bar Mitzvah [52] is an attack against TLS protocol that can be built exploiting the Invariance Weakness in the RC4 stream cipher. The Invariance Weakness is a vulnerability in RC4 stream cipher, that, if exists, leads to the preservation of the least significant bits of the state permutation intact throughout the initialization process of the keys. These bits are the processed by the PRGA algorithm and so determine the least significant bits of the pseudo-random output. These biased stream bytes are XORed with the plaintext bytes and will generate significant leakage of plaintext bytes from the ciphertext bytes. The big problem with these biases is that, even if they occur rarely, they are effective in 100 keystream bytes with very high probability.

Since the Invariance Weakness involves only the first 100 bytes of the keystream, only the first 100 bytes of traffic, respectively upstream and downstream can be targeted by an attack. In addition the first encrypted bytes that are sent on a TLS connection are represented by the *Finished* message (usually 36 bytes). Consequently the attack can only recover about 64 bytes of protected data. In order to mount the attack, an attacker first sniffs a large number of TLS connections encrypted with RC4, waiting the use of a weak key. At this point the attacker can recover the LSBs of the keystream bytes and use them to extract the LSBs of the plaintext bytes from the ciphertext. The first application of this attack is recovering a part of a session cookie, using a large number of its encryptions. This attack is expected to have a hit with a weak key every 2^{24} connections. Since for each hit the probability of the Invariance Weakness is 1-5%, a lot of hits are required to complete a successful partial plaintext recovery.

There is also a passive variant of this attack, in which the attacker eavesdrops on the inbound traffic to a web application, aiming to steal credit card number info. The attacker is required to wait 1 billion connections for a weak key usage event and when it happens, the Invariance Weakness can be used to predict keystream LSBs. In this way the plaintext LSBs can be recovered from ciphertext LSBs by the attacker. It is important to note that the compromised credit card number or password is of a random victim, with the attacker not having any control over its identity.

The only solution to avoid Bar Mitzvah attack is to not use RC4 stream cipher. Consequently upgrading the version of TLS protocol to the most recent one (TLS 1.3) is a good kind of protection, since the support for weak ciphers, included RC4, has been removed.

RC4 NOMORE [53] attack exploits weaknesses in RC4 encryption algorithm to reveal information which was previously thought to be safely encrypted. It relies on two types of statistical biases present in the keystream. The first one, commonly called the Fluhrer-McGrew biases, is that two consecutive bytes are biased towards certain values. The second type of biases, called the Mantin's ABSAB biases, is that a pair of consecutive bytes is likely to repeat itself. RC4 NOMORE attack can be mounted exploiting a combination of these two types of biases.

In particular a group of researchers showed that, using RC4 NOMORE, an attacker can decrypt web cookies, which are normally protected by the HTTPS protocol, that are used to identify users and authorize their actions. If an attacker can recover a cookie, he will be able to impersonate the victim to the server, gaining access to personal information and performing actions as if he were the victim.

In order to execute this kind of attack the attacker has to insert malicious JavaScript code inside the website visited by the victim. This code will induce the victim to transmit encrypted requests which contain the victim's web cookie. By monitoring and analysing all the requests sent by the victim, the attacker can generate a list containing possible values for the cookie. Then, testing all the possibilities, the attacker will be able to recover the correct cookie in about 75 hours. With respect to previous attacks again RC4 stream cipher, this short execution time increase a lot the feasibility of this attack. In addition any kind of information that is sent encrypted with RC4 repeatedly on a TLS connection can be recovered, not only web cookies.

Also in this case, the only countermeasure for this attack, is the removal of the support for RC4, as has been done with TLS 1.3.

5.4.6 Other attacks

In March 2014, some researchers found an important vulnerability in the TLS implementation of OpenSSL, named Heartbleed vulnerability [54]. Heartbleed allows attackers to read memory from vulnerable servers, including cryptographic keys, login credentials, and other private data. In addition it was one of the most impactful vulnerability, considering OpenSSL, since it was easy to exploit and a lot of server were using HTTPS and other TLS services.

The Heartbleed attack is based on a bad implementation of the Heartbeat Extension by OpenSSL. This extension allows the two end-point of a connection to detect whether its peer is still present. During the handshake each one of the two peers can send a *Heartbeat Request* message to verify connectivity. This message is composed by four different parts: a *Type* filed (1 byte), a *Payload Length* field (2 bytes), the payload and and at least 16 bytes of random padding. When an endpoint receives an *Heartbeat Request* message, it answers with an *Heartbeat Response* message, containing the received payload and its own random padding.

The OpenSSL implementation of the Heartbeat Extension contained a vulnerability that allowed either end-point to read data following the payload message in its other's memory by specifying a *Payload Length* larger than the amount of data in the Heartbeat Request message. Because the *Payload Length* field is two bytes, the peer responds with up to 2^{16} bytes ($\sim$64 KB) of memory.

The countermeasure for this attack is represented by the patch for the vulnerability of OpenSSL library, the revocation of the compromised keys and the redistribution of new keys.

Security Losses from Obsolete and Truncated Transcript Hashes (SLOTH [55]) refers a class of transcript collision attacks due to the use of obsolete and truncated hash constructions (such as MD5) on popular cryptographic protocols such as TLS. Cryptographic protocols widely use hash functions to implement authentication and integrity security properties. Indeed they are used to generate digital signatures and message authentication codes (MAC) and in combination with key-derivation functions (KDF). In 2005 MD5, a popular cryptographic hash function, was broken the first time by a collision attack. From that moment other attacks against cryptographic hash functions arose, increasing their speed and danger more and more. SLOTH attack forces a hash-construction downgrade to MD5, reducing the expected security.

In TLS, the client authenticates itself by presenting an X.509 certificate and then signing a hash of the entire handshake transcript (up to that point) with the private key corresponding to the certificate. In TLS 1.0 the signature was implemented with SHA1+MD5, while in version 1.1 SHA1 was used. Instead TLS 1.2 allowed clients and servers to negotiate the signature and hash algorithms they support. Consequently when a TLS handshake is executed, it could happen that a strong hash algorithm, such as SHA-256 or SHA-512 is negotiated, but also a weak hash algorithm such as MD5 could be chosen.

If a TLS client C and server S both support RSA-MD5 signatures for client authentication and C is willing to use the same certificate to authenticate at some malicious server M, M then can mount a man-in-the-middle transcript collision attack that allows it to impersonate C at S. In order to execute this attack, shown in Fig. 5.5, the attacker M must compute a chosen-prefix MD5 collision between two handshake transcripts, one between C and M, and the other between M and S. The attack complexity depends on the difficulty of finding such collisions. For MD5, such collisions are known to require computing 2^{39} hashes.

The version of TLS that is vulnerable to SLOTH attack is TLS 1.2. The only possible solution to counteract this vulnerability is to remove support for MD5 and SHA-1, as has been done with TLS 1.3 standardization.

The SWEET32 attack [56] is based on a security weakness in the block ciphers used in cryptographic protocols, such as TLS 1.2. It's similar to the attacks that exploit RC4 vulnerabilities in terms of computational complexity. Block ciphers are often used by TLS protocol to protect the confidentiality of data exchanged between client and server. First data to be transferred is divided into chunks of a certain length, called blocks, and then each block is encrypted following a specific mode of operation. Another parameter that is introduced when a block cipher is used

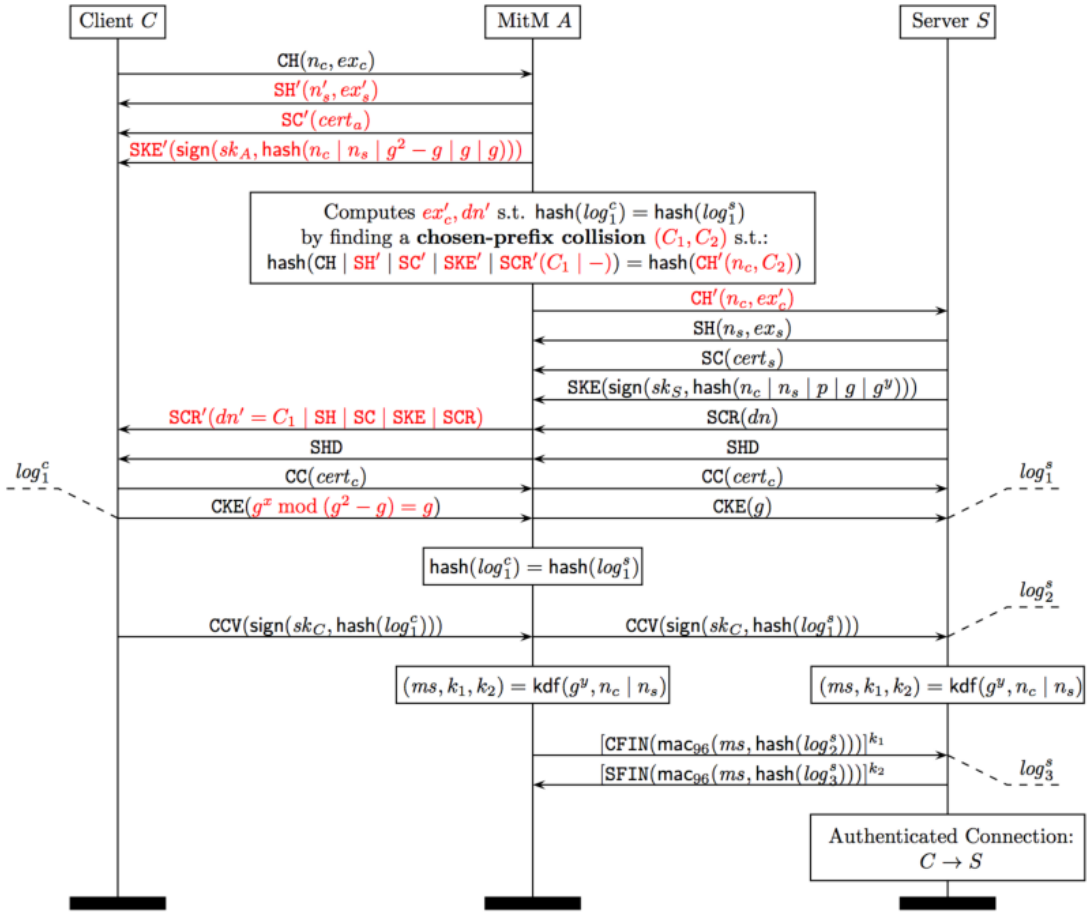


Figure 5.5. The SLOTH attack (source: [55])

is the key length, that is different with respect to the block length. Both these two values are set by the algorithm. In 3DES algorithm, the block size is 64 bits, while for AES, it's 128 bits.

The best attack against a block cipher is a brute force attack to recover the key, that has a complexity of 2^k, where k is the key size. But the security of a block cipher cannot be reduced to the value k, because also the block size n is also an important security parameter, defining the amount of data that can be encrypted under the same key. The shorter a block size is, the more vulnerable it is to a birthday attack, a type of vulnerability based on the birthday problem in probability theory. The birthday bound is defined as the number of attempt needed before a collision is found and with most modes of operation (e.g. CBC, CTR, GCM, OCB, etc.) it's equal to $2^{n/2}$. This problem is well-known by cryptographers, who always require keys to be changed well before $2^n/2$ blocks.

Short block sizes make web servers vulnerable to hitting the same hash for multiple inputs. An attacker can exploit this vulnerability in order to recover secure HTTP cookies, observing data exchange between a web server and a website. With a modern block cipher with 128-bit blocks such as AES, the birthday bound is equal to 256 EB. But if the block size is reduced to 64 bits, the birthday bound decreases to only 32 GB. When the amount of data encrypted under a fixed key approaches this limit, the security guarantees of the mode of operation start to crumble.

Block ciphers are used with a mode of operation in order to deal with messages with a different length than the block size n. For example, with CBC mode the message M is divided into blocks m_i and is encrypted as:

$$c_i = E_k(m_i \oplus c_{i-1})$$

where c_{-1} is an initialization value usually denoted as IV. The birthday bound with CBC mod is equal to $2^{n/2}$. Indeed after $2^{n/2}$ message blocks encrypted with the same key, a collision between

two ciphertext blocks $c_i = c_j$ is expected and a birthday attack can be executed. Since E_k is a permutation, a collision in the output means that the inputs are the same ($m_i \oplus c_{i-1} = m_j \oplus c_{j-1}$) which reveals the XOR of two plaintext blocks:

$$m_i \oplus m_j = c_{i-1} \oplus c_{j-1}$$

According to the birthday paradox, with 2^d blocks of data the expected number of collisions is about 2^{2d-n-1}. In this way an attacker can recover the XOR between two plaintext, that may be not enough for an attack with practical impact. But if a fixed (HTTP cookie) is sent repeatedly and some fraction of the plaintext is known, there is a chance that a collision leaks the XOR between the fixed secret and the known plaintext and consequently the secret is revealed. Assuming 2^s is the number of copies of the secret and 2^t is the number of known encrypted blocks, if $s + t \geq n$ this attack is expected to succeed with high probability.

In the case of 64-bit block cipher algorithms, a collision attack can happen after about 32 GB of data, so if an attacker generates and sends a sufficient amount of data from the browser to the web server with JavaScript, the HTTP session cookies can be discovered. But experiments showed that as much as 785 GB of data might be needed for a practical collision attack. The only way to prevent SWEET32 attacks is to ensure that a block cipher with a big block size is used (such as 128 bits), so that the probability of having a collisions decreases and the birthday bound increases.

The truncation attack [57, 58] is a security attack that can be applied when tearing down an TLS connection and was discovered by researchers Ben Smyth and Alfredo Pironti of the French National Institute for Research in Computer Science and Control (INRIA) in 2013.

In TLS protocol specification two different modes for the termination of a connection are indicated. A graceful connection closure represents the end of a successful connection and in this case all the messages are received as sent. On the other hand, a fatal closure happens at the end of an unsuccessful connection (for example after receiving a corrupt message) and this means that the ordered delivery of only a prefix of all fragments is guaranteed. If an endpoint of a TLS communication wants to close the connection, it must send a specialized close notify alert message to inform the other party that no more messages will be send on the current TLS connection.

In web applications, the server typically notifies the user about its state using some positive feedbacks, when state change is successful, or negative feedbacks, when an error occurred. The lack of feedback can cause confusion and violate basic design principles. For instance, if a user attempts to save a file and no feedback is provided, then the user does not know if the file was saved or not.

In order to execute TLS truncation attack, the attacker need to create a situation in which user's and server's perspective of the application state are de-synchronized. To this purpose the attacker must be placed in the same network of the victim. When the user makes a logout request to the server, the man-in-the-middle blocks it, so that the user remains logged into a web service. In addition, the attacker injects an unencrypted TCP FIN message to close the connection. Consequently the user receives feedback that the sign-out request has been successfully executed, whereas the server is unaware of the user's request. In this way the attacker can get control of the user account without the victim knowing.

5.4.7 Attack against TLS 1.3

All the above mentioned attacks are not executable against the latest version of TLS. Indeed, when in August 2018 TLS 1.3 was standardized, a set of changes has been introduced in the protocol with respect to previous version (TLS 1.2) in order to remove all the known vulnerabilities and to prevent all the known attacks against the protocol. The following elements of the protocol have been removed:

- not-AEAD ciphers, in order to avoid attacks on legacy ciphers;
- RSA key exchange, to always provide perfect forward secrecy to the communication channel;

Attack	TLS 1.0	TLS 1.1	TLS 1.2	DTLS 1.2	Ref.
Bleichenbacher	x				[38]
Renegotiation	x	x	x	x	[39]
3SHAKE	x	x	x	x	[40]
BEAST	x				[41]
LUCKY 13	x	x	x	x	[42]
POODLE	x	x	x		[43]
CRIME	x	x	x	x	[44]
TIME	x	x	x		[45]
BREACH	x	x	x		[46]
DROWN	x	x	x		[49]
FREAK	x	x	x	x	[48]
Logjam		x			[50]
RC4 biases	x	x	x		[51]
Bar Mitzvah	x	x	x		[52]
RC4 NOMORE	x	x	x		[53]
Heartbleed	x	x	x		[54]
SLOTH			x	x	[55]
SWEET32			x	x	[56]
Truncation	x	x	x	x	[57]

Table 5.1. Attacks against TLS and DTLS protocol.

- broken hash algorithms (MD5, SHA-1), to avoid SLOTH and similar attacks;
- *CHangeCipherSpec* message, in order to reduce the handshake size (in terms of number of messages and latency);
- data compression, in order to prevent CRIME attack;
- session renegotiation, to avoid attacks that exploit the renegotiation mechanism.

Moreover two new features have been introduced: *EncryptedExtensions* message, to avoid transmitting preferences in plaintext, and the 0-RTT mode to increase session resumption speed. In addition, with TLS 1.3, all the messages after *ServerHello* are encrypted, leading to certificates encryption. Despite all the improvements and changes made to the design and the implementations of TLS protocol, different authors during recent years have tried to discover whether the latest version (TLS 1.3) is affected by some vulnerabilities or problems that can be exploited to mount an attack.

The authors in [59] complained about PSK-based authentication mode for session resumption, typical of TLS 1.3, and explained that it's characterized by a misuse vulnerability which can lead a node to act as client and server at the same time. The attack described by the authors

is a reflection attack and it is named "Selfie". It requires that the attacker acts as a man-in-the-middle, between a client that has created a session S1 with the server, sending a *ClientHello* message with pre shared key extension. The attacker captures the message and reflects it back to client in another session S2, where client and server of session S1 have swapped their role. In this second session, the server (the client in S1) replies to the client (the server in S1) with *ServerHello* and server *Finished* messages, that are captured again by the attacker and sent to the client in S1. The client of S1, that is convinced to talk with S1 server, will reply with a client *Finished* message, which is again relayed back by the attacker to S2 server (the client in S1). As a result, the client in S1 has opened a "Selfie" session with himself, meaning that any piece of data that is sent by him in S1 will be echoed back to him through S2.

TLS, as indicated above, is often vulnerable to version downgrade attacks, where a man-in-the-middle attacker interferes with the handshake and leads the communicating parties to fall back from a higher version of the protocol to lower ones. Different mechanism have been designed in the last years in order to prevent this kind of attacks. One of them is called SCSV (Signaling Cipher Suite Value) and it consists in mentioning the latest supported TLS version of the client in the *ClientHello* message. For this solution to be used and effective, both the client and the server must support it. Another downgrade protection mechanism introduced in TLS 1.3 uses a version-specific value in *ServerHello* message. The authors of [60] studied the SCSV mechanism adoption in TLS 1.3 and its support in 10 major Web browsers, including Firefox, Chrome, and Edge, in five operating systems. As a result, they identified that two network stacks of Microsoft and Apple are vulnerable to downgrade attacks, due to not supporting the adequate protection mechanism. The authors showed that TLS version can be downgraded till 1.0 in MS Windows, while in iOS and macOS only to 1.2. Consequently a downgrade attack against TLS 1.3 is practically feasible and could be quite easily mounted due to the lack of SCSV protection.

The extension of the encryption to all the messages in the handshake of TLS 1.3 after the *ServerHello* not only provides stronger security guarantees, but it was also introduced to protect identities of the server and client (certificates are then encrypted). The authors of [61] tested the privacy of the full handshake and session resumption mechanisms of TLS 1.3. They stated that TLS 1.3 offers an rather optimal degree of privacy, especially during a full handshake, during which the protocol provides a perception of server unlinkability. The level of privacy is slightly decreased when an old session is resumed, because a method of linking sessions between two objects is introduced. Moreover, the use of session tickets for session resumption contributes to the privacy reduction.

In work [62] the authors studied the way protocols supporting 0-RTT mode, such as TLS 1.3 and Google QUIC, are affected by replay attacks. The 0-RTT mode enables a client to establish a fresh provisional key without interaction, based only on cryptographic material obtained in previous connections. With this solution client and server can exchange early application data before the fully secure keys are negotiated, protecting them with the provisional key. The authors showed that, at least at time of writing, replay attacks were accepted as inevitable by the protocol creators.

5.4.8 Attacks against DTLS

DTLS can be intended as an adaptation of TLS protocol to run on top of UDP, with TLS 1.1 corresponding to DTLS 1.0 and TLS 1.2 corresponding to DTLS 1.2. DTLS 1.3, standardized recently on April 2022 by IETF with RFC 9147 [37], is based on TLS 1.3 and provides equal security guarantees with the exception of anti replay protection. Consequently, all the attacks presented above against TLS protocol, can be executed also to break DTLS protocol, accordingly with versions correspondence. The only exception is represented by the attacks against RC4 stream cipher, because this encryption algorithm is not allowed in DTLS.

Because DTLS protocol runs on top of UDP, it's potentially vulnerable to two denial of service (DoS) attacks: the standard resource consumption attack and the amplification attack. To prevent these attacks, the cookie exchange process must be implemented during the DTLS handshake. When the client sends its *ClientHello*, the server responds with an *HelloVerifyRequest* message containing an opaque cookie. In order to demonstrate it's not using a fake IP address, the client must send back the cookie to the server in a second *ClientHello* message.

5.5 QUIC attacks

This section only includes QUIC-related attacks. Attacks against the TLS 1.3 handshake are pertinent given that TLS acts as a security component of QUIC; basically, the two protocols cooperate. Specifically, QUIC uses the keys derived from the TLS handshake for providing confidentiality and integrity to the exchanged packets through its transport layer. The various attacks are split into five categories, namely cryptographic, handshake, fuzzing, transport-layer, and privacy. In all the subsections, the discussed works are given in a chronological order.

5.5.1 Attacks against Google QUIC

In the previous section the Bleichenbacher attack against TLS has been presented. It allows an attacker to decrypt PKCS#1 v1.5 ciphertexts exploiting the responses of an "oracle", that can be given by a TLS server. By analysing the time necessary to receive responses from the "oracle", the attacker can understand if a ciphertext is correctly padded or not.

In 2015, the authors of [63] presented two attacks against TLS 1.3 and Google QUIC, based on the Bleichenbacher attack against TLS. In particular they identified a way to transfer the potential weakness of old TLS versions to these two protocols, that don't even support PKCS#1 v1.5 encryption by design. The target of this attack is to impersonate a server by generating valid signature for some handshake messages with the help of "signing oracle", represented by a vulnerable TLS-RSA server implementation.

The first attack against TLS 1.3, that has a reduced real impact, need the attacker to forge requires the TLS *CertificateVerify* message before the client releases the connection with the aid of a quick "Bleichenbacher-oracle". This attack demonstrates that, even if the exploited legacy algorithm has been removed form TLS 1.3 standard, it's not enough to protect it against its weaknesses. The problem is that, TLS 1.3 is not directly susceptible to Bleichenbacher-type attacks, but it becomes vulnerable if the server supports static RSA in the context of previous versions of TLS.

The second version of the attack against Google QUIC protocol is more practical in terms of time. Indeed, thanks to the possibility of using pre-computation, the attacker may take a great amount of time to use the "Bleichenbacher-oracle" to generate the needed signature and the attack is anyway effective. Since signed values in QUIC are independent of the client's connection request, the attacker can decide to pre-compute the signature before the client initiates the connection.

As described above, TLS protocol is vulnerable to an attack called DROWN, that let an attacker break a passively collected RSA key for any TLS server if this key was also used for SSL 2.0. The author of [49] showed that also Google QUIC is vulnerable to DROWN attack, if it is executed in a man-in-the-middle variant, enabling the attacker to impersonate a server. This attack requires that the target server exposes its RSA public keys through SSL 2.0. The key point of this version of the attack is the *server config* message sent by a Google QUIC server to the client during a connection attempt. This message is signed by the server and contains the server Diffie-Hellman public value used for key establishment. In order to execute the attack, the MitM needs to forge a valid *server config*, counterfeiting the server signature. To this purpose the attacker has to find a valid SSL 2.0 ciphertext. Therefore, the next time the client tries to connect to the server using Google QUIC, the attacker will used the forged *server config* message to execute the attack. According to the authors, this attack against Google QUIC requires 2^{17} SSL 2.0 connections and 2^{58} offline work. Since the IETF version of QUIC protocol uses TLS 1.3 in order to execute the cryptographic handshake, instead of Google QUIC CRYPTO, it is not vulnerable to this attack.

The authors in [64] contributed the first study that focused on Google QUIC security and provided the definition of a forma security model, named QACCE. They found that Google QUIC does not provide perfect forward secrecy as it's done instead by TLS employed with DHE, and that QUIC may be unable to achieve one of its main goals, such as 0-RTT connections, in the presence of attackers. In addition they identified three different attacks affecting Google QUIC protocol, that in the end could easily prevent QUIC from achieving minimal latency advantages

either by having it fall back to TCP or by making the client and server to have an inconsistent view of their handshake and then preventing the connection establishment to complete correctly. They stated that these QUIC security weaknesses are caused by the mechanism used to reduce latency.

The first one is a replay attack that requires the attacker to be a man-in-the-middle. The target of the attack is causing an handshake failure with a DoS attack mounted through the replay of either some key establishment parameters to the client or the packet that contained the source address token (*stk*) of the client to the server, where the *stk* is an authenticated-encryption block of the client's IP address and a timestamp. This attack can be related with the TCP SYN flood attack.

The second attack they presented consists in manipulating unprotected parameters inside Google QUIC packets, such as the CID or the *stk*, in order to force a downgrade to a TLS connection. Also this kind of attack requires a MitM and is related by the authors with the attack against TLS that exploits SSL downgrade vulnerability. Also in this case the attack results in an handshake failure, preventing the client to connect to the server.

There is also another way to make the handshake fail and it's represented by the third attack, named Crypto Stream Offset attack. In this case the attacker need to inject a four character string, that is, "REJ\0" into the handshake message. With this attack, the opponent can deny the client access to the requested application or force the client to fall back to a TLS connection. The authors compared this attack with the TCP ACK flooding attack, in which an attacker sends multiple ACKs to the server to confuse it and drop the connection.

The work in [26] provided a comparable study between TLS 1.2 and Google QUIC from a security point of view. In particular the authors evaluated the impact of the two protocols vulnerabilities when they are employed to provide an HTTP/2-based web services. Specifically for Google QUIC, the authors described the same attacks presented in [64] in the context of HTTP/2 services. They showed that attacks against Google QUIC have impact mainly on the client, indeed the server can not crash or reach a state where the web service offered to end-users is largely degraded. On the other hand, TLS attacks target more precisely the server and for this reason QUIC is a more reliable solution to implement an HTTP/2-based web server, according to the authors.

The authors in [65] during their analysis on Google QUIC identified two 0-RTT oriented attacks, namely QUIC RST and Version Forgery. The first attack consists in tricking the client to give up the connection, sending him a reset packet, as if the server refused to connect. On the other hand, in Version Forgery attack, the attacker impersonate the server and sends to the client a version negotiation packet with a not supported version. Also in this case the target is to make the client give up the connection. Thanks to the use of ARP spoofing, in order to associate the attacker's MAC address with the IP address of another host, the authors stated that both these attacks are more easily mounted in a LAN environment.

J. Zhang et al. [66] presented a modeling and verification method for cryptographic protocols in order to verify the security of the QUIC handshake protocol. They revealed that if an attacker is able to forge the *CHLO* message and the client ephemeral Diffie-Hellman public value and send them to the server, it can impersonate the client. Moreover, the authors proposed a revised QUIC handshake protocol which uses the client private key to sign the *CHLO* message and the public value of ephemeral Diffie-Hellman, consequently preventing this kind of attack.

5.5.2 Attacks against IETF QUIC

Gagliardi and Levillain [67] analysed QUIC protocol in the Internet drafts from version 18 to version 23. Using the Scapy Python library, they tested the IETF QUIC handshake protocol of the various existing implementation and also four different attacks in ten live demo servers.

The first attack was a sort of version negotiation, where the attacker sends a different protocol version and they found that the only implementation that is vulnerable to this kind of attack and leads to a timeout was ngtcp2.

The second attack is an attempt to exploit the behaviour of implementations against the client *Initial* packet length. Indeed a mitigation method against amplification attacks states that the client *Initial* packet must have a length of at least 1200 bytes and the response of the server should be at most 3 times the size of the received packet. The authors demonstrated that only two implementations replied to short *Initial* client packet (300 bytes) and always with a message tha respect the mitigation method rule, providing in this way protection against amplification attacks.

During the analysis on the third attack, named missing parameters, the authors showed that some servers accepted packets with missing mandatory parameters (as describe in QUIC RFC), such as the maximum size of each packet.

Finally, they mounted frame mangling attacks by sending forbidden frames to the demo servers. In QUIC, an *Initial* packet could contain only four different types of packets, i.e., crypto, ack, connection close, and padding frames. During their tests the authors found that also four different packets, including a ping packet, could be encapsulated inside an *Initial* QUIC packet and the result is unexpectedly a successful handshake instead of an error.

In 2021 Nawrocki et al. [68] presented two type of attacks against IETF QUIC handshake protocol, named state-overflow and reflective amplification, in order to prove that QUIC is vulnerable to resource exhaustion attacks. State-overflow attacks exploit the fact that QUIC keeps a state from each connected client. This kind of attack is similar to TCP SYN flood attack. The authors revealed that a defensive mechanism has been introduced in the RFC of QUIC, but it introduces additional delays in the authentication procedure and for this reason is considered in contrast with the target of QUIC protocol of reducing latency. This mitigation method consists in retrying authentication if the IP address of the client cannot be verified.

On the other hand, a reflective amplification attack can be executed against QUIC, since it is built on top of UDP. An attacker can send an *Initial* packet including a spoofed source IP address to a QUIC server. The server then replies with an *Initial* QUIC message and an *Handshake* one. Since the last message includes the server certificates, it is larger compared to the client request. Consequently, the server is only allowed to triple the bytes if the request in its response, as long as the client is unverified.

The authors showed that on average, the Internet is exposed to four QUIC floods per hour and half of these attacks occur concurrently with other common attack types such as TCP/ICMP floods.

5.6 Recommendations

As it was explained all along this chapter, there are different solutions to protect a communication between two parties. Among all the analysed protocols, the best solutions turned out to be TLS 1.3 and IETF QUIC, that offer comparable and adequately strong level of security. The security guarantees provided by DTLS 1.3 are the same of TLS 1.3, but, if the cookie exchange process is not implemented correctly, this protocol is vulnerable to DoS attacks. The oldest versions of TLS, DTLS and QUIC protocol are affected by known vulnerabilities discovered during years and in some cases have also been deprecated. Consequently the first recommendation is to use always the latest version of each security protocol, that is the most secure one. Another recommendation is to configure the system to use the security protocol in the most secure way, whatever the version. An example for this tip is to prevent the use of weak cipher and always prefer Authenticated Encryption with Associated Data (AEAD) algorithm for data protection.

Chapter 6

Conclusion

Protecting the communication between two endpoints has grown in importance over the last few decades and to this purpose a lot of security protocols were born. TLS (Transport Layer Security), a protocol to provide security on a TCP channel, became the most important and widely adopted standard in this field. Throughout all its versions, till the most recent one in 2018 (TLS 1.3), a lot of changes and improvements have been gradually introduced in order to meet the need for a ever higher level of security, providing countermeasures for new attacks and strengthening the security properties.

Since a growing number of application layer protocols were designed to use UDP transport, a new solution had to be introduced. Indeed TLS cannot go directly on top of UDP because it's designed to run on top of a reliable transport protocol, such as TCP, and it's unable to cope with the packet loss or reordering that may occur. For this reason DTLS (Datagram Transport Layer Security) was developed, in particular for applications with an unreliable transport layer, such as in the case of the IoT, video conferencing, VoIP, VPN and online gaming. When the protocol was designed, security experts kept it as close to TLS as possible, preserving much of the code and infrastructure to minimise the number of new security techniques required. This means that DTLS is basically a datagram-compatible version of TLS and offers the same security guarantees. In April 2022 the latest version of DTLS protocol (1.3), corresponding to TLS 1.3, was standardized by IETF (Internet Engineering Task Force).

QUIC (Quick UDP Internet Connections) is an encrypted-by-default transport protocol introduced by Google several years ago. This protocol was designed in order to make HTTP traffic faster, reducing latency, and at the same time provide at least the same security as TLS. QUIC resulted to be a real competitor of the TCP+TLS protocol stack on the Internet. In 2015 QUIC was adopted by IETF and its standardization process started. There were some differences introduced in the new version of the protocol that made Google QUIC and IETF QUIC diverge: the packet format was changed, as well as the handshake phase, for which the proprietary QUIC Crypto protocol was abandoned and substituted by the TLS 1.3 handshake protocol. In addition also the mapping of HTTP was modified. IETF QUIC has improved the original Google QUIC design thanks to open collaboration from many organizations and individuals, with the shared goal of making the Internet faster and more secure. The target of this thesis was to analyse all these three protocols, in particular keeping attention on the performance and the provided security.

For what concerns the performance different experiments have been conducted in order to measure different metrics that can be meaningful to quantify and describe the behaviour of each protocol in different scenarios. Since QUIC has been adopted, firstly by Google Chrome and subsequently also by several other browsers, as the protocol used to download a webpage, the first group of experiments is focused on analysing a situation in which the client, represented by the browser, tries to download a webpage with both TCP+TLS and QUIC. This is one of the most used approach to compare QUIC and TLS performance also by other authors and researchers in past works. On average QUIC resulted in better performance with respect to the TCP+TLS stack, being faster in both receiving the first byte of a webpage tested and carrying out the

download of the entire requested resource. The only exception to this behaviour is when QUIC has to deal with big pages composed by an high number of elements: in this situation TLS is able to provide at least comparable, if not better, performance with respect to QUIC.

Another parameter that has been measured during the experiments, using different libraries, is the handshake time. Also in this case there were no unexpected results: QUIC is the fastest protocol to establish a connection between a client and a server, providing on average an handshake time around 1-RTT. For the other protocol the handshake phase takes on average a longer time. Indeed, for what concerns TLS, the negotiation of the security parameters is added to the TCP 3-way handshake, while DTLS has an handshake phase with the same messages as TLS, to which the cookie exchange procedure must be added.

As a last point, the three security protocols are compared in terms of performance when they have to deal with file download. During this last group of experiments the behaviour of each protocol has been tested under different network conditions, varying the bandwidth and the delay of the links of the network. As a result, QUIC provided good performance almost in all scenarios, especially in the environment with poor network conditions. Instead, when a network with high bandwidth and low delay is considered and the file to be downloaded is big, the performance of QUIC deteriorates and it's no more the best choice. In addition, in almost all the scenarios, on average the file transfer executed with QUIC resulted to be faster also with respect to the one with HTTP in plaintext, without any kind of security introduced in the communication between client and server.

For what concerns TLS protocol, version 1.3 performed on average a little bit better than version 1.2. Partially, this is probably due to the fact that the latest version of the protocol is characterized by a reduced handshake latency. Anyway, TLS resulted to be not the best solution for file download compared to the other protocols (in particular QUIC) and in many situations this protocol provided the slowest performance. DTLS protocol instead could represent a good solution to transfer a file, especially in a network with high bandwidth, but it presents and important drawback: the lack of a mechanism to manage packet loss. Indeed, with DTLS protocol, if packet loss occurs, the lack of an ACK mechanism (since DTLS runs on top of UDP) leads to only partial reception of the file, since packet lost will not be sent again.

From the security point of view the evaluation has been done throughout all the versions that have been designed and standardized for all the protocols. In particular TLS has been enhanced and changed during the last years, in order to reduce the handshake latency required and to improve the implementations of the security properties. Indeed this protocol has been analysed in detail during years and some vulnerabilities and possible attacks has been discovered. The target of the protocol designers was to strengthen the security guarantees provided and to introduce countermeasures for all the documented attacks against the protocol. As a result, TLS 1.3 is not vulnerable to the majority of the attacks presented in this thesis.

Since DTLS is designed as TLS but running on top of UDP, it offers the same security guarantees and it's vulnerable to the same kind of attacks (with only few exceptions). Given its nature, the most important difference of DTLS with respect to TLS is that, since DTLS handshake takes place over datagram transport, it's potentially vulnerable to two denial of service (DoS) attacks: the standard resource consumption attack and the amplification attack. To prevent these attacks, DTLS uses the cookie exchange technique.

QUIC was proposed by Google in order to reduce the latency needed for the connection establishment. From the security perspective, the target of Google working group was to introduce in the communication security guarantees comparable to TLS. To this purpose the proprietary QUIC Crypto protocol has been designed and adopted. When QUIC was adopted by IETF, the major change applied to the protocol regards the handshake. Indeed the IETF version of QUIC encapsulates TLS 1.3 packet for the connection establishment. This makes QUIC protocol completely comparable with TLS 1.3 from the security point of view.

In order to conclude, during this work it has been demonstrated that QUIC offers a very good and efficient solution to protect the communication between two endpoints. Indeed this protocol is able to quickly create the channel between the two peers, protect in the proper way the exchanged data and provide high performance in data transfer. The security introduced by IETF QUIC is

the same as TLS 1.3, indeed both protocols are vulnerable to a very reduced number of attacks while writing. Maybe QUIC will not substitute completely TLS in the next future, since it's a very important and working standard, but its diffusion and adoption for the communication management on Internet will almost certainly grow more and more in the coming years.

Bibliography

[1] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2." RFC-5246, August 2008, DOI 10.17487/RFC5246

[2] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1." RFC-4346, April 2006, DOI 10.17487/RFC4346

[3] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3." RFC-8446, August 2018, DOI 10.17487/RFC8446

[4] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2." RFC-6347, January 2012, DOI 10.17487/RFC6347

[5] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, "The QUIC Transport Protocol: Design and Internet-Scale Deployment", Proceedings of the Conference of the ACM Special Interest Group on Data Communication, Los Angeles (California, USA), 2017, pp. 183–196, DOI 10.1145/3098822.3098842

[6] A. Langley and W.-T. Chang, "QUIC Crypto", December 2016, https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/edit

[7] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport." RFC-9000, May 2021, DOI 10.17487/RFC9000

[8] G. Carlucci, L. De Cicco, and S. Mascolo, "HTTP over UDP: an experimental investigation of QUIC", Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca (Spain), 2015, pp. 609–614, DOI 10.1145/2695664.2695706

[9] P. Megyesi, Z. Krämer, and S. Molnár, "How quick is QUIC?", 2016 IEEE International Conference on Communications (ICC), Kuala Lumpur (Malaysia), 2016, pp. 1–6, DOI 10.1109/ICC.2016.7510788

[10] Andrea Cardaci, https://github.com/cyrus-and/chrome-har-capturer

[11] P. Biswal and O. Gnawali, "Does QUIC Make the Web Faster?", 2016 IEEE Global Communications Conference (GLOBECOM), Washington (DC, USA), 2016, pp. 1–6, DOI 10.1109/GLOCOM.2016.7841749

[12] A. Kakhki, S. Jero, D. Choffnes, C. Nita-Rotaru, and A. Mislove, "Taking a Long Look at QUIC: An Approach for Rigorous Evaluation of Rapidly Evolving Transport Protocols", Proceedings of the 2017 Internet Measurement Conference, London (United Kingdom), 2017, pp. 290–303, DOI 10.1145/3131365.3131368

[13] S. Cook, B. Mathieu, P. Truong, and I. Hamchaoui, "QUIC: Better for what and for whom?", 2017 IEEE International Conference on Communications (ICC), Paris (France), 2017, pp. 1–6, DOI 10.1109/ICC.2017.7997281

[14] K. Nepomuceno, I. N. d. Oliveira, R. R. Aschoff, D. Bezerra, M. S. Ito, W. Melo, D. Sadok, and G. Szabó, "QUIC and TCP: A Performance Evaluation", 2018 IEEE Symposium on Computers and Communications (ISCC), Natal (Brazil), 2018, pp. 45–51, DOI 10.1109/ISCC.2018.8538687

[15] Ravi Netravali, Anirudh Sivaraman, Greg D. Hill, and Keith Winstein, http://mahimahi.mit.edu/

[16] K. Wolsing, J. Rüth, K. Wehrle, and O. Hohlfeld, "A Performance Perspective on Web Optimized Protocol Stacks: TCP+TLS+HTTP/2 vs. QUIC", Proceedings of the Applied Networking Research Workshop, Montreal (Quebec, Canada), 2019, pp. 1–7, DOI 10.1145/3340301.3341123

[17] sitespeed.io, https://github.com/sitespeedio/browsertime
[18] T. Shreedhar, R. Panda, S. Podanev, and V. Bajpai, "Evaluating QUIC Performance Over Web, Cloud Storage, and Video Workloads", IEEE Transactions on Network and Service Management, vol. 19, June 2022, pp. 1366–1381, DOI 10.1109/TNSM.2021.3134562
[19] LiteSpeed Technologies Inc, https://github.com/litespeedtech/lsquic
[20] P. Miranda, M. Siekkinen, and H. Waris, "TLS and energy consumption on a mobile device: A measurement study", 2011 IEEE Symposium on Computers and Communications (ISCC), Kerkyra (Greece), 2011, pp. 983–989, DOI 10.1109/ISCC.2011.5983970
[21] S. Bossi, T. Anselmo, and G. Bertoni, "On TLS 1.3 - Early Performance Analysis in the IoT Field", Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP), Rome (Italy), January 2016, pp. 117–125, DOI 10.5220/0005688901170125
[22] M. T. Paracha, D. J. Dubois, N. Vallina-Rodriguez, and D. Choffnes, "IoTLS: Understanding TLS Usage in Consumer IoT Devices", Proceedings of the 21st ACM Internet Measurement Conference, Virtual Event, 2021, pp. 165–178, DOI 10.1145/3487552.3487830
[23] M. Vucinic, B. Tourancheau, T. Watteyne, F. Rousseau, A. Duda, R. Guizzetti, and L. Damon, "DTLS performance in duty-cycled networks", 2015 IEEE 26th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC), Hong Kong, 2015, pp. 1333–1338, DOI 10.1109/PIMRC.2015.7343505
[24] S. Gallenmüller, D. Schöffmann, D. Scholz, F. Geyer, and G. Carle, "DTLS Performance - How Expensive is Security?", CoRR, vol. abs/1904.11423, April 2019, DOI 10.48550/ARXIV.1904.11423
[25] G. Restuccia, H. Tschofenig, and E. Baccelli, "Low-Power IoT Communication Security: On the Performance of DTLS and TLS 1.3", 2020 9th IFIP International Conference on Performance Evaluation and Modeling in Wireless Networks (PEMWN), Virtual Event, 2020, pp. 1–6, DOI 10.23919/PEMWN50727.2020.9293085
[26] A. Saverimoutou, B. Mathieu, and S. Vaton, "Which secure transport protocol for a reliable HTTP/2-based web service: TLS or QUIC?", 2017 IEEE Symposium on Computers and Communications (ISCC), Heraklion (Greece), 2017, pp. 879–884, DOI 10.1109/ISCC.2017.8024637
[27] The OpenSSL project, http://www.openssl.org/
[28] wolfSSL Inc., https://www.wolfssl.com/
[29] Olaf Bergmann (TZI) and others, https://github.com/eclipse/tinydtls
[30] Sebastian Gallenmüller, https://github.com/gallenmu/MoonState
[31] https://github.com/Mbed-TLS/mbedtls
[32] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport." RFC-9000, May 2021, DOI 10.17487/RFC9000
[33] A. Freier, P. Karlton, and P. Kocher, "The secure sockets layer (ssl) protocol version 3.0." RFC-6101, August 2011, DOI 10.17487/RFC6101
[34] T. Dierks and C. Allen, "The TLS Protocol Version 1.0." RFC-2246, January 1999, DOI 10.17487/RFC2246
[35] K. Moriarty and S. Farrell, "Deprecating tls 1.0 and tls 1.1." RFC-8996, March 2021, DOI 10.17487/RFC8996
[36] E. Rescorla and N. Modadugu, "Datagram transport layer security." RFC-4347, April 2006, DOI 10.17487/RFC4347
[37] E. Rescorla, H. Tschofenig, and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3." RFC 9147, April 2022, DOI 10.17487/RFC9147
[38] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1", Advances in Cryptology — CRYPTO '98, Santa Barbara (California, USA), 1998, pp. 1–12, DOI 10.1007/BFb0055716
[39] M. Ray and S. Dispensa, "Renegotiating TLS", November 2009, https://kryptera.se/Renegotiating%20TLS.pdf
[40] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti, and P. Y. Strub, "Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS", 2014 IEEE Symposium on Security and Privacy, San Jose (California, USA), 2014, pp. 98–113, DOI 10.1109/SP.2014.14
[41] T. Duong and J. Rizzo, "Here come the ⊕ ninjas", 2011. http://www.hpcc.ecs.soton.ac.uk/dan/talks/bullrun/Beast.pdf

[42] N. J. Al Fardan and K. G. Paterson, "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols", 2013 IEEE Symposium on Security and Privacy, San Francisco (California, USA), 2013, pp. 526–540, DOI 10.1109/SP.2013.42

[43] B. Möller, T. Duong, and K. Kotowicz, "This POODLE Bites: Exploiting The SSL 3.0 Fallback", September 2014, http://www.bmoeller.de/pdf/ssl-poodle.pdf

[44] J. Rizzo and T. Duong, "The CRIME attack", ekoparty Security Conference, Buenos Aires (Argentina), 2012

[45] A. Shulman, "A Perfect CRIME? Only TIME Will Tell", BlackHat, Amsterdam (Netherlands), 2013. https://paper.bobylive.com/Meeting_Papers/BlackHat/Europe-2013/bh-eu-13-a-perfect-crime-beery-wp.pdf

[46] Y. Gluck, N. Harris, and A. Prado, "BREACH: reviving the CRIME attack", 2013, http://css.csail.mit.edu/6.858/2020/readings/breach.pdf

[47] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of tls", Commun. ACM, vol. 60, Jan 2017, pp. 99–107, DOI 10.1145/3023357

[48] "FREAK attack", 2015, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0204

[49] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Käsper, S. N. Cohney, S. Engels, C. Paar, and Y. Shavitt, "DROWN: Breaking TLS Using SSLv2", 25th USENIX Security Symposium (USENIX Security 16), Austin (Texas, USA), 2016, pp. 689–706. https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_aviram.pdf

[50] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann, "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice", Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver (Colorado, USA), 2015, pp. 5–17, DOI 10.1145/2810103.2813707

[51] N. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. N. Schuldt, "On the Security of RC4 in TLS", 22nd USENIX Security Symposium (USENIX Security 13), Washington (DC, USA, 2013, pp. 305–320. https://www.usenix.org/system/files/conference/usenixsecurity13/sec13-paper_alfardan.pdf

[52] H. I. Initiative, "Attacking SSL when using RC4", 2015, https://www.imperva.com/docs/HII_Attacking_SSL_when_using_RC4.pdf

[53] M. Vanhoef and F. Piessens, "All Your Biases Belong to Us: Breaking RC4 in WPA-TKIP and TLS", 24th USENIX Security Symposium (USENIX Security 15), Washington (DC, USA), 2015, pp. 97–112. https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-vanhoef.pdf

[54] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman, "The Matter of Heartbleed", Proceedings of the 2014 Conference on Internet Measurement Conference, Vancouver (BC, Canada), 2014, pp. 475–488, DOI 10.1145/2663716.2663755

[55] K. Bhargavan and G. Leurent, "Transcript collision attacks: Breaking authentication in TLS, IKE, and SSH", Network and Distributed System Security Symposium, San Diego (California, USA), 2016, DOI 10.14722/NDSS.2016.23418

[56] K. Bhargavan and G. Leurent, "On the Practical (In-)Security of 64-Bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN", Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna (Austria), 2016, pp. 456–467, DOI 10.1145/2976749.2978423

[57] B. Smyth and A. Pironti, "Truncating TLS Connections to Violate Beliefs in Web Applications", 7th USENIX Workshop on Offensive Technologies (WOOT 13), Washington (D.C., USA), 2013. https://www.usenix.org/system/files/conference/woot13/woot13-smyth.pdf

[58] D. Berbecaru and A. Lioy, "On the Robustness of Applications Based on the SSL and TLS Security Protocols", Public Key Infrastructure, 2007, pp. 248–264, DOI 10.1007/978-3-540-73408-6_18

[59] N. Drucker and S. Gueron, "Selfie: Reflections on TLS 1.3 with PSK", J. Cryptol., vol. 34, Jul 2021, DOI 10.1007/s00145-021-09387-y

[60] S. Lee, Y. Shin, and J. Hur, "Return of Version Downgrade Attack in the Era of TLS 1.3", Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies, Barcelona (Spain), 2020, pp. 157–168, DOI 10.1145/3386367.3431310

[61] G. Arfaoui, X. Bultel, P.-A. Fouque, A. Nedelcu, and C. Onete, "The privacy of the TLS 1.3 protocol", Proceedings on Privacy Enhancing Technologies, vol. 2019, July 2019, DOI 10.2478/popets-2019-0065

[62] M. Fischlin and F. Günther, "Replay Attacks on Zero Round-Trip Time: The Case of the TLS 1.3 Handshake Candidates", 2017 IEEE European Symposium on Security and Privacy (EuroS&P), Paris (France), 2017, pp. 60–75, DOI 10.1109/EuroSP.2017.18

[63] T. Jager, J. Schwenk, and J. Somorovsky, "On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption", Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver (Colorado, USA), 2015, pp. 1185–1196, DOI 10.1145/2810103.2813657

[64] R. Lychev, S. Jero, A. Boldyreva, and C. Nita-Rotaru, "How Secure and Quick is QUIC? Provable Security and Performance Analyses", 2015 IEEE Symposium on Security and Privacy, San Jose (California, USA), 2015, pp. 214–231, DOI 10.1109/SP.2015.21

[65] X. Cao, S. Zhao, and Y. Zhang, "0-RTT Attack and Defense of QUIC Protocol", 2019 IEEE Globecom Workshops (GC Wkshps), Waikoloa (Hawaii, USA), 2019, pp. 1–6, DOI 10.1109/GCWkshps45667.2019.9024637

[66] J. Zhang, X. Gao, L. Yang, T. Feng, D. Li, Q. Wang, and R. Amin, "A Systematic Approach to Formal Analysis of QUIC Handshake Protocol Using Symbolic Model Checking", Security and Communication Networks, vol. 2021, Jan 2021, DOI 10.1155/2021/1630223

[67] E. Gagliardi and O. Levillain, "Analysis of QUIC session establishment and its implementations", 13th IFIP International Conference on Information Security Theory and Practice (WISTP), Paris (France), Dec 2019, pp. 169–184, DOI 10.1007/978-3-030-41702-4_11

[68] M. Nawrocki, R. Hiesgen, T. C. Schmidt, and M. Wählisch, "QUICsand: Quantifying QUIC Reconnaissance Scans and DoS Flooding Events", Proceedings of the 21st ACM Internet Measurement Conference, Virtual Event, 2021, pp. 283–291, DOI 10.1145/3487552.3487840

Appendix A

User's Manual

All the code written and used during this thesis is contained inside a set of folders, divided by protocol ('HTTP', 'TLS', 'DTLS', 'QUIC'). For each protocol, with the only exception of HTTP, there is a folder for each library through which the protocol is implement for the experiments. In this section how to use the code developed for the thesis in order to reproduce the experiments is explained. Each set of experiments that used some pieces of code is considered separately.

A.1 Connection Time and Time To First Byte measurements

A.1.1 TLS

The code to execute to experiments concerning Connection Time and TTFB with TLS protocol is contained inside the folder /TLS/libcurl. The source code is inside the 'src' subfolder (tls_perf.c). In order to use this code, it's necessary to install 3 libraries:

- BoringSSL
- curl
- nghttp2

The installation paths of each one of the library must be inserted in the correct place inside the 'CMakeLists.txt' file in order to compile. Subsequently the following commands must be executed:

```
$ cmake .
$ make
```

The obtained program can receive the following command line arguments:

-4 to use IPv4

-2 to use HTTP/2

-3 to use TLS 1.3 (the default is TLS 1.2)

-u [**website**] to select the url to which the client tries to connect

-p [**port**] to select the port for the connection

The output of the execution of this program is a set of 2 files: the first one containing the TTFB measured value and the second with the Connection Time measured value. Moreover, the program prints on the standard output other information about the established connection. The folder contains also a bash script ('run_experiments.sh') that can be used to perform 50 connections to each one of the 3 webpages considered during the experiments conducted for the thesis (Google, Youtube and Facebook).

A.1.2 QUIC

The code to execute to experiments concerning Connection Time and TTFB with QUIC protocol is contained inside the folder /QUIC/lsquic/bin. The client program is inside the file 'http_client.c'. The main folder /QUIC/lsquic contains the all the code of the lsquic library, adequately modified in order to make possible the execution of the experiments. In order to execute the code for the experiments it's necessary to install lsquic library and all its dependencies. To this purpose the instruction in the thesis or the guide at the link https://github.com/litespeedtech/lsquic must be followed. The folder /QUIC/lsquic/bin contains a *Makefile* that can be used, with the *make* command, in order to build all the code.

In order to reproduce the thesis experiments, the http_client program can be used with the following parameters:

-t to redirect the output of the program on the standard output

-s [**website**] for the webpage to visit

-p [**path**] for the path requested

-v [**version**] to specify the version to use between Q043, Q046, Q050, ID27, ID29 and I001

-K to discard server response

The program gives as output also two additional files containing the values of TTFB and Connection Time measured during the connection to the website. The bash script 'run_experiments.sh' inside the folder /QUIC/lsquic/bin contains the necessary code to automate the tests and execute 50 connections towards the 3 website tested during the thesis.

A.2 Handshake measurements

A.2.1 TLS

The code to execute to experiments concerning the Handshake Time with TLS protocol is contained inside the folder /TLS/openssl. In order to execute this code is necessary to install the OpenSSL library on your system, as it's explained in the thesis. The client and server programs to perform and measure a TLS handshake are respectively contained in 'client_tls_hs.c' and 'server_tls_hs.c'. These are the commands to compile client and server programs:

```
$ gcc -Wall -o client_hs client_tls_hs.c -L/usr/lib -lssl -lcrypto
$ gcc -Wall -o server_hs server_tls_hs.c -L/usr/lib -lssl -lcrypto
```

Once the programs are compiled, you can run them in the following way:

```
$ ./client_hs <hostname> <version> <portnum> <ca_cert> <ciphersuite>
```

where:

hostname is the IP address to which the client is trying to connect

version is the TLS selected version ('2' for TLS 1.2 and '3' for TLS 1.3)

portnum is the number of the port on which the server is listening

ca_cert is the path to the certificate of the CA that issued the server certificate

ciphersuite is the list of ciphersuite that you want to enable for the handshake (must be separated by ':')

```
$ ./server_hs <version> <portnum> <cert_file> <key_file>
```

where:

version is the TLS selected version (the same as before)

portnum is the port number for the server

cert_file is the path to certificate file for the server

key_file is the path to the file that contains the private key of the server (corresponding to the public one in the certificate)

The file 'tls_1_2_supported_ciphersuites' contains all the possible ciphersuites for TLS 1.2 and the same file is available also for version 1.3.

The main folder contains also 4 sub-folder: ca_rsa_cert contains the RSA certificate file of the CA that issued the certificate of the server, while server_rsa_cert contains the RSA certificate and the RSA private key of the server. There are also 2 folders with the same structure but containing DSA certificates. These 4 folders and their content can be used to test the programs. The file 'dh_param_2048.pem' has been generated by the command:

```
$ openssl dhparam -out dh_param_2048.pem 2048
```

and contains additional DH-parameters for the server, that are used in case a DHE ciphersuite is selected by the client.

The bash script 'run_experiments.sh' can be used to automate the experiments about the handshake. Once the server is started, this script can be executed (it receives the same parameters as the client program) in order to try 10 connections to the server. The handshake time can be measured through Wireshark as explained in the thesis.

A.2.2 DTLS

The code to execute to experiments concerning the Handshake Time with DTLS protocol is contained inside the folder /DTLS/openssl. In order to execute this code is necessary to install the OpenSSL library on your system, as it's explained in the thesis. The client and server programs to perform and measure a DTLS handshake are respectively contained in 'client_dtls_hs.c' and 'server_dtls_hs.c'. These are the commands to compile client and server programs:

```
$ gcc -Wall -o client_hs client_dtls_hs.c -L/usr/lib -lssl -lcrypto
$ gcc -Wall -o server_hs server_dtls_hs.c -L/usr/lib -lssl -lcrypto
```

Once the programs are compiled, you can run them in the following way:

```
$ ./client_hs <hostname> <portnum> <ca_cert> <ciphersuite>
```

where:

hostname is the IP address to which the client is trying to connect

portnum is the number of the port on which the server is listening

ca_cert is the path to the certificate of the CA that issued the server certificate

ciphersuite is the list of ciphersuite that you want to enable for the handshake (must be separated by ':')

```
$ ./server_hs <portnum> <cert_file> <key_file>
```

where:

portnum is the port number for the server

cert_file is the path to certificate file for the server

key_file is the path to the file that contains the private key of the server (corresponding to the public one in the certificate)

The file 'dtls_1_2_supported_ciphersuites' contains all the possible ciphersuites for DTLS 1.2.

The main folder contains also 4 sub-folder: ca_rsa_cert contains the RSA certificate file of the CA that issued the certificate of the server, while server_rsa_cert contains the RSA certificate and the RSA private key of the server. There are also 2 folders with the same structure but containing DSA certificates. These 4 folders and their content can be used to test the programs. The file 'dh_param_2048.pem' has been generated by the command:

```
$ openssl dhparam -out dh_param_2048.pem 2048
```

and contains additional DH-parameters for the server, that are used in case a DHE ciphersuite is selected by the client.

The bash script 'run_experiments.sh' can be used to automate the experiments about the handshake. Once the server is started, this script can be executed (it receives the same parameters as the client program) in order to try 10 connections to the server. The handshake time can be measured through Wireshark as explained in the thesis.

A.2.3 QUIC

The code needed for the handshake measurements with QUIC protocol is contained in two files, 'echo_client_hs.c' and 'echo_server_hs.c', inside /QUIC/lsquic/bin folder. The main folder /QUIC/lsquic contains the all the code of the lsquic library, adequately modified in order to make possible the execution of the experiments. In order to execute the code for the experiments it's necessary to install lsquic library and all its dependencies. To this purpose the instruction in the thesis or the guide at the link https://github.com/litespeedtech/lsquic must be followed. The folder /QUIC/lsquic/bin contains a *Makefile* that can be used, with the *make* command, in order to build all the code.

Once the programs are compiled, the server code for the handshake measurement can be executed as following:

```
$ ./echo_server_hs -c [domain_name,cert_file,private_key_file]
```

With no additional parameter the server is executed on port 12345. On the other hand the client code can be executed as following:

```
$ ./echo_client_hs -s [server_addr]:12345
```

As showed in the corresponding thesis chapter, the measurement of the handshake time can be executed with the help of Wireshark.

A.3 File Transfer Time measurements

This section refers to the code generated and used to execute the experiments for file transfer time measurements. To this purpose file1, file2 and file3 in '/file_samples' folder are the files used during the experiments executed to measure file transfer time (respectively of 500kB, 5MB and 50MB).

A.3.1 HTTP

The code in folder ('http_client.c' and 'http_server.c') folder '/HTTP' is used to implement the file transfer with HTTP protocol. The two programs can be compiled using the following commands:

```
$ gcc -Wall -o client client_http.c
$ gcc -Wall -o server server_http.c
```

Once the programs are compiled, you can run the server program in the following way:

```
$ ./server <port>
```

where:

port is the port number for the server

In order to run the HTTP client, the following command must be executed:

```
$ ./client <hostname> <port> <request path>
```

where:

hostname is the IP address to which the client is trying to connect

port is the number of the port on which the server is listening

request path is the name of the requested file

The time needed to download a file, once server and client are executed, can be measured with Wireshark as explained in the thesis.

A.3.2 TLS

The code to execute to experiments concerning the File Transfer Time with TLS protocol is contained inside the folder /TLS/openssl. In order to execute this code is necessary to install the OpenSSL library on your system, as it's explained in the thesis. The client and server programs to perform and measure the time needed to exchange a file with TLS are respectively contained in 'client_tls_ft.c' and 'server_tls_ft.c'. These are the commands to compile client and server programs:

```
$ gcc -Wall -o client_ft client_tls_ft.c -L/usr/lib -lssl -lcrypto
$ gcc -Wall -o server_ft server_tls_ft.c -L/usr/lib -lssl -lcrypto
```

Once the programs are compiled, you can run them in the following way:

```
$ ./client_ft <hostname> <version> <portnum> <ca_cert> <ciphersuite>
    <requested file>
```

where:

hostname is the IP address to which the client is trying to connect

version is the TLS selected version ('2' for TLS 1.2 and '3' for TLS 1.3)

portnum is the number of the port on which the server is listening

ca_cert is the path to the certificate of the CA that issued the server certificate

ciphersuite is the list of ciphersuite that you want to enable for the handshake (must be separated by ':')

requested file is the the name of the file the client wants to download

```
$ ./server_ft <version> <portnum> <cert_file> <key_file>
```

where:

version is the TLS selected version (the same as before)

portnum is the port number for the server

cert_file is the path to certificate file for the server

key_file is the path to the file that contains the private key of the server (corresponding to the public one in the certificate)

The time needed to download a file, once server and client are executed, can be measured with Wireshark as explained in the thesis.

A.3.3 DTLS

The code to execute to experiments concerning the File Transfer Time with DTLS protocol is contained inside the folder /DTLS/openssl. In order to execute this code is necessary to install the OpenSSL library on your system, as it's explained in the thesis. The client and server programs to perform and measure the time needed to exchange a file with DTLS protocol are respectively contained in 'client_dtls_ft.c' and 'server_dtls_ft.c'. These are the commands to compile client and server programs:

```
$ gcc -Wall -o client_ft client_dtls_ft.c -L/usr/lib -lssl -lcrypto
$ gcc -Wall -o server_ft server_dtls_ft.c -L/usr/lib -lssl -lcrypto
```

Once the programs are compiled, you can run them in the following way:

```
$ ./client_ft <hostname> <portnum> <ca_cert> <ciphersuite> <requested file>
```

where:

hostname is the IP address to which the client is trying to connect

portnum is the number of the port on which the server is listening

ca_cert is the path to the certificate of the CA that issued the server certificate

ciphersuite is the list of ciphersuite that you want to enable for the handshake (must be separated by ':')

requested file is the the name of the file the client wants to download

```
$ ./server_ft <portnum> <cert_file> <key_file>
```

where:

portnum is the port number for the server

cert_file is the path to certificate file for the server

key_file is the path to the file that contains the private key of the server (corresponding to the public one in the certificate)

The time needed to download a file, once server and client are executed, can be measured with Wireshark as explained in the thesis.

A.3.4 QUIC

The code needed for the file transfer time measurements with QUIC protocol is contained in two files, 'echo_client_ft.c' and 'echo_server_ft.c', inside /QUIC/lsquic/bin folder. The main folder /QUIC/lsquic contains the all the code of the lsquic library, adequately modified in order to make possible the execution of the experiments. In order to execute the code for the experiments it's necessary to install lsquic library and all its dependencies. To this purpose the instruction in the thesis or the guide at the link https://github.com/litespeedtech/lsquic must be followed. The folder /QUIC/lsquic/bin contains a *Makefile* that can be used, with the *make* command, in order to build all the code.

Once the programs are compiled, the server code for the file transfer measurement can be executed as following:

```
$ ./echo_server_ft -c [domain_name,cert_file,private_key_file]
```

With no additional parameter the server is executed on port 12345. On the other hand the client code can be executed as following:

```
$ ./echo_client_ft -s [server_addr]:12345 -f [requested_file]
```

As showed in the corresponding thesis chapter, the measurement of the file transfer time can be executed with the help of Wireshark.