POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master Thesis

Certificate Validation and TLS Interception

Advisors
prof. Antonio Lioy
prof.ssa Diana Berbecaru

Candidate
Matteo Simone

Academic year 2021-2022

Summary

Transport Layer Security (TLS) is the protocol mostly used nowadays to protect communications between a client and a server. It uses X.509 certificate chains to guarantee server and optionally client authentication. It is the client, the Relying Party (RP), that must validate that chain.

TLS affected all those legal, security and performance use cases that required access to plain HTTP traffic. That led to the introduction of TLS interception solutions. This thesis aims to study the different behavior of TLS clients and TLS interception products facing misconfigured X.509 certificates in a testing/enterprise environment. Additionally, wants to collect the actual Certificate Transparency (CT) usage, the TLS version negotiated, OCSP Stapling and OCSP Must-Staple support in the top 1 million domains.

After a theoretical background of X.509 and PKI standards and of TLS protocol (Section 2 and Section 3), the works of D. McLuskie and X. Belleken[1], Ahmad Samer Wazan et al.[2] and Louis Waked et al.[3] have been described and used as reference point for misconfigured certificate generation (Section 4). In the same section are collected the results obtained by Yabing Liu et al.[4] Johanna Amann et al.[5] with respect to what concerns the certificate validation process and OCSP Stapling and Must-Staple.

In Section 5 is described the methodology of this project. This work's tests have been run against a multitude of desktop/mobile browsers (Google Chrome, Mozilla Firefox, Safari, Opera, Microsoft Edge) and TLS interception products (Mitmproxy, Squid, Kaspersky Total Security, ESET Smart Security) running on the major operating systems (Windows, macOS, Ubuntu, iOS, Android). To be more exhaustive in the analysis, an Apache, Nginx and Lighttpd server has been configured for each test. Tests consist in a set of certificates generated assigning wrong values to their attributes and extensions which do not respect what is stated by the RFC 5280[6] and/or the CA/Browser Forum[7]. The extensions involved are: Subject Alternative Name, Basic Constraints, Key Usage, Extended Key Usage, TLS Feature, Authority Information Access and CRL Distribution Point. Tests and their server configurations are generated automatically through a set of Python scripts, using PyOpenSSL and cryptography libraries.

After deliberately malformed certificates tests, a statistical measure of the top 1 million domains is performed. For each of them, the TLS version negotiated, OCSP Stapling and OCSP Must-Staple support and all the Signed Certificate Timestamps (SCTs) embedded in their certificate are retrieved.

Results obtained have been critically discussed and compared with related works. They show that there is still inconsistency among the different TLS interception products. This is less visible in browsers that follows in most of the cases the suggestions of the CA/Browser Forum. The major issue still remains the certificate revocation status check. The support for OCSP Stapling in TLS interception products is still limited. Its adoption in the top domains increased from the 27% in 2019 to 31% according to this thesis measures. OCSP Must-Staple remained unused. There is,

then, an increased usage of TLS 1.3, negotiated in about the 75% of the connections established. From the measures obtained the CT trust is concentrated on Google, DigiCert, Cloudflare and Let's Encrypt logs. Almost the 98% of the certificate retrieved contained 2 (55.6%) and 3 (41.7%) SCTs embedded in them. There is a 2.2% of the domains that does not embed SCTs in their certificates. They probably prefer to provide the SCTs via TLS extension (privacy aware solution) or via OCSP response.

This report can be taken as a good insight of the current state of certificate validation process by major browsers and TLS interception products and of the actual global CT usage, but, also, as the base for further analysis in the future. It suggests to extend the software produced in this work and to study other TLS interception solutions such as ETS and CDNs caching.

Acknowledgements

I would like to acknowledge and give my warmest thanks to my supervisors Antonio Lioy and Diana Berbecaru, whose guidance and suggestions carried me through all the stages of this work.

I would like also to thank my family. All the sacrifices done by my parents allowed me to arrive at this point of my carrier. Without their support I wouldn't managed to become who I am now. I would like to give a special thank, also, to my brother Stefano. He was always there, ready to help he, when I needed it.

Finally, I want to thank all my friends and colleagues with who I shared the "university experience" and that where always there to sustain and encourage me since the beginning of this journey.

Contents

Chapter 1

Introduction

Transport Layer Security (TLS) is widely used nowadays to provide secure connections between clients and servers. TLS is the evolution of the old Secure Socket Layer (SSL) protocol and consists in the generation of a secure communication channel. This channel is configured accordingly to an handshake phase in which, both the client and the server, exchange information about the cipher suites (cryptographic algorithms) they support, random numbers used to generate a master secret and other cryptographic parameters to negotiate a symmetric key to be used to protect data in transit.
Server authentication, one of the main feature of TLS, is achieved by means of X.509 certificates, the key building block of the Public Key Infrastructure (PKI), which has the task to verify the identity of the subject (server, person, software) to be bond to the public-private key-pair before to issue the certificate itself. A certificate is a signed attestation and who signs it is the so called Certification Authority (CA), which has its own signed certificate as well.
Generally, a server, during the TLS handshake phase, sends to clients a logical chain of certificates: at the end there is its own leaf certificate signed by an intermediate CA; the certificate of the intermediate CA could be signed by another intermediate CA and so on until a root CA self-signed certificate. Root CAs are trusted a priori and are installed at client-side. It is the client, the Relying Party (RP), who has to validate the whole certificate chain, from the leaf certificate to the second last one.

The certificate is generated in the X.509 format following the rules and constraints documented in the RFC-5280 [8]. In the RFC-5280 are, in particular, described the version 3 extensions to be used in addition to the standard Distinguished Name values that identify the certificate's subject and the certificate's issuer and certificate's attributes. It states also how to distribute Certificate Revocation Lists (CRLs). Each extension introduces a constraint that could be marked as critical or not. If the case is the former, then the TLS client that does not recognize the object identifier (OID) of that particular extension must reject the certificate and, so, fail the handshake phase.

The validation process follow a well defined procedure: firstly the digital signature of the issuer of the certificate is verified; then, the RP checks if the certificate is within its validity period; next, it checks whether the certificate is revoked or not and lastly it verifies that all the constraints introduced by the certificate extensions are met. This process is repeated for each of the certificates in the logical chain obtained by the server up to the root certificate.

Other than CRLs, which give a history of all the revoked certificates issued by a specific CA, the principal alternative is the Online Certificate Status Protocol (OCSP), a client-server solution that allows the RP to obtain the actual status of a certificate. Enhanced versions of this

last protocol are also studied in this thesis. They are OCSP Stapling and OCSP Must-Staple. Respectively they allow the client to request directly the status of the certificate during the TLS handshake and impose the presence of the certificate status in the TLS handshake. A further solution used nowadays is CRLSet, a small portion of CRLs, directly pushed to the browsers.

The advent of TLS affected all those use cases, such as application level traffic filtering, QoS analysis, government data retention regulations and so on. In all those cases, the access to plain HTTP traffic is required. For this reason, TLS interception solutions have been introduced. This thesis mainly concentrates on those which split the TLS session into two separate sessions, following the man-in-the-middle pattern.

After the incident of DigiNotar CA in 2011 [9], when its key got compromised and used to issue fraudulent certificates, Certificate Transparency (CT) was introduced. It is based of publicly accessible log servers that allow everyone to verify that the certificate received has been not issued by a compromised CA.

Since the PKI standards state only whether a certificate should be considered valid or not and there is not a strict universal set of rules to follow, the clients' programmers are free to implement their software according to their interpretation of the standards.

This thesis studies the different behavior of TLS clients and TLS interception products facing deliberately misconfigured X.509 certificates in a testing/enterprise environment. Additionally, wants to collect the actual Certificate Transparency (CT)[10] usage, the TLS versions negotiated, OCSP Stapling and OCSP Must-Staple support in the Top-1-Million domains.

This study takes inspiration from past works. *X.509 Certificate Error testing* by D. McLuskie and X. Belleken[1], *On the Validation of Web X.509 Certificates by TLS interception products* by Ahmad Samer Wazan et al.[2] and *To intercept or not to Intercept: Analyzing TLS Interception in Network Appliances* by Louis Waked et al.[3] have been used as reference point for misconfigured certificate generation. They created certificates varying the Subject Alternative Name, Key Usage, Extended Key Usage, CRL Distribution Point, Authority Information Access and TLS Feature extensions' values and the Distinguished Name description of the certificate's subject.

This work's tests have been run against desktop/mobile browsers (Google Chrome, Mozilla Firefox, Safari, Opera, Microsoft Edge) and TLS interception products (Mitmproxy, Squid, Kaspersky Total Security, ESET Smart Security) running on the major operating systems (Windows, macOS, Ubuntu, iOS, Android). To be more exhaustive in the analysis, an Apache, Nginx and Lighttpd server have been configured for each test. Tests consist in a set of certificates automatically generated assigning values to their attributes and extensions which do not respect the state-of-the-art configuration suggested by the RFC 5280[6] and/or the CA/Browser Forum[7] and in automatic server configurations for each of the server applications tested.

Certificate revocation status check is an important step in the certificate validation process. However, is still incorrectly performed. For instance, if a revoked certificate is still considered valid by a client, an attacker could easily trick a victim to connect to a malicious web-server exploiting it.

Yabing Liu et al.[4] in their paper *An End-to-End Measurement of Certificate Revocation in the Web's PKI* went through the various revocation status check techniques including CRLs, OCSP, its enhanced versions OCSP Stapling and OCSP Must-Staple and CRLSets. Together with the analysis performed by Johanna Amann et al. in *Mission Accomplished? HTTPS security after DigiNotar* [5], by Taejoong Chung et al. in *Is the Web Ready for OCSP Must-Staple?*[11] and, again, by Ahmad Samer Wazan et al., this work have been used as inspiration to produce a script to measure CT usage, OCSP Stapling and OCSP Must-Staple support and TLS versions negotiated in the real world. Connecting to the Majestic Top-1-Million domains[12], it first collects

all the needed data and then analyzes them producing, also, charts with its results. The purposes of this analysis is to discover what are the major CT log server providers nowadays, how many Signed Certificate Timestamps (SCTs) are embedded in each certificates and which specific log server they refer to. In addition this statistical analysis wants to give an insight of the evolution of the TLS 1.3, OCSP Stapling and OCSP Must-Staple adoption.

Chapter 2

X.509 certificate validation

2.1 X.509 certificate

The X.509 is a standard format for digitally signed certificates used to bind a public-private key-pair to a specific subject. The subject could be a person, a server or a software. This kind of certificate is indispensable during the TLS handshake phase in order to identify the server the user wants to connect to. Optionally, it is used also to identify the client who wants to connect to a specific server. More detailed discussion on what and how TLS works can be found in Chapter 3. The standard is described in the RFC-5280 [6]. In this RFC are described the basic format of a certificate, its extensions and how to generate a proper Certificate Revocation List (CRL).

The basic structure of a certificate is the following:

- *Version*: it is the version of the certificate and could be 1, 2 or 3.
- *Serial number*: it is a set of at most 20 octets that identify a certificate.
- *Signature algorithm*: contains the signature algorithm and its parameters used to sign the certificate.
- *Issuer*: identifies the entity that has signed and issued the certificate. It must contain a non-empty distinguished name (DN).
- *Validity*: composed by two timestamps, NotBefore and NotAfter, that express the validity period of the certificate.
- *Subject*: identifies the entity associated with the public key stored in the subject public key field.
- *SubjectPublicKeyInfo*: the algorithm used, the length and the public key associated to the subject.
- *DigitalSignature*: the digital signature of the certificate.

The subject and the Issuer of the certificate are described trough:

- *CommonName* (CN). Contains a string identifying the subject/issuer of the certificate. Can be a person name, a DNS name, an IP address version 4 or an IP address version 6.

- *OrganizationName* (O). It is a string that contains the organization name of the subject or the issuer. Must be present and equal in both the subject and the issuer. Should not be longer than 64 characters.
- *OrganizationalUnitName* (OU). It is a string that better specifies the organization. Should not be longer than 64 characters.
- *Country* (C). Two uppercase characters that identify the country. Must be present and equal in both the subject and the issuer.
- *StateOrProvince* (ST). It is a string that specifies the name of the State or province name. Must be present and equal in both the subject and the issuer.

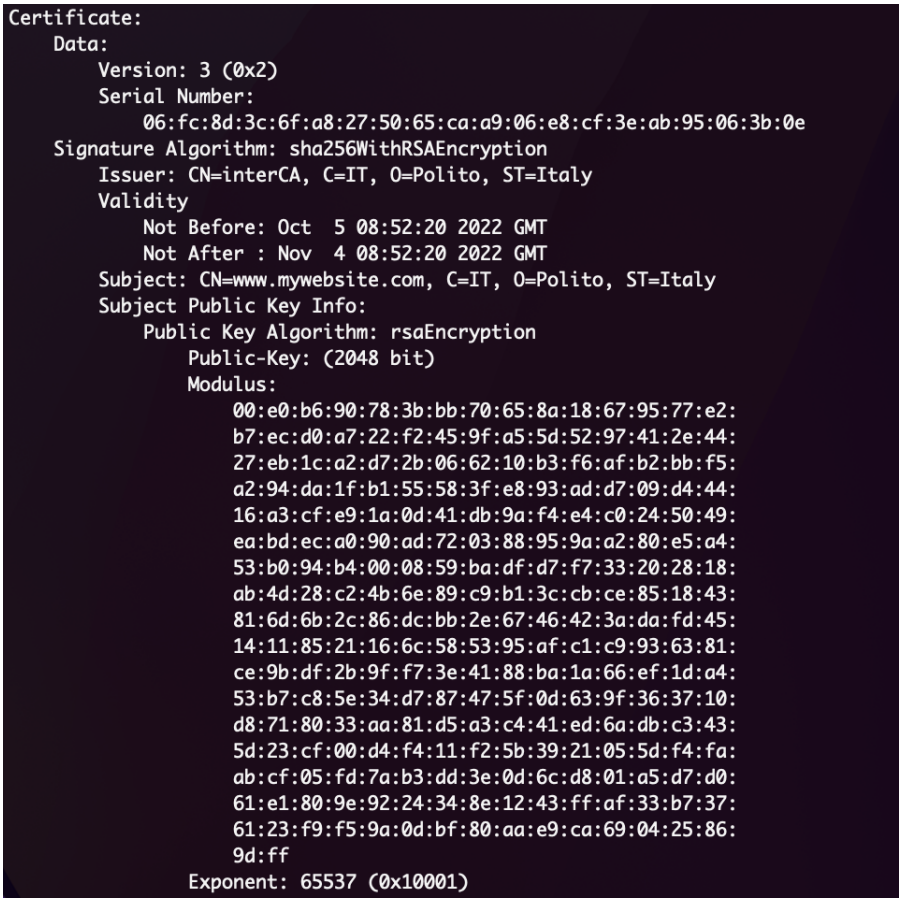

Figure 2.1. X.509 standard format example.

According to RFC-5280 [6], the X.509v3 extensions used in this thesis can be described as follow:

- *SubjectKeyIdentifier*. Identifies the certificate that contains a particular public key. In CA certificates, the value of the *Subject Key Identifier* must be the value placed in the key identifier field of the *Authority Key Identifier* extension of certificates issued by the subject

of this certificate. Applications are not required to verify that key identifiers match when performing certification path validation.

- *AuthorityKeyIdentifier*. Identifies the public key corresponding to the private key used to sign a certificate. The *keyIdentifier* field of the extension must be included in all certificates generated by conforming CAs to facilitate certification path construction. In case of self-signed certificate it may be omitted.
- *SubjectAlternativeNames*. Specifies different names for the subject, generally in the format of a DNS name or a public IP address. If used together with subject CN it must match exactly one entry of this extension.
- *KeyUsage*. Extension that defines the purpose of the key contained in the certificate. It is a nine-bit string. Possible values are:
  - *digitalSignature* (bit 0), when set specifies that the key must be used for digital signature.
  - *nonRepudiation*(bit 1), when set the subject public key is used to verify digital signatures.
  - *keyEncipherment*(bit 2), when set the subject public key is used for enciphering private or secret keys.
  - *dataEncipherment*(bit 3), when set the subject public key is used to encipher directly data.
  - *keyAgreement*(bit 4), when set the subject public key is used for key agreement.
  - *keyCertSign*(bit 5), when set the subject public key is used to verify signatures on public key certificates. If this bit is set then the certificate's *BasicConstraints* extension must be set with CA equal to true.
  - *cRLSign*(bit 6), when set the subject public key is used to verify signatures on certificate revocation lists.
  - *encipherOnly*(bit 7), when set the *keyAgreement* bit is also set. The subject public key may be used only for enciphering data while performing key agreement.
  - *decipherOnly*(bit 8), when set the *keyAgreement* bit is also set. The subject public key may be used only for deciphering data while performing key agreement.
- *ExtendedKeyUsage*. Expresses other possible purposes for the key contained in the certificate. It is handled separately from *KeyUsage* extension but their values must be consistent. If not, then, the certificate must not be used for any purpose. The extension could be set to:
  - *serverAuth*. Used for TLS server authentication, compliant with *digitalSignature*, *keyEncipherment* or *keyAgreement* bits.
  - *clientAuth*. Used for TLS client authentication, compliant with *digitalSignature* and/or *keyAgreement* bits.
  - *codeSigning*. Used to sign downloadable executable code, compliant with *digitalSignature* bit.
  - *emailProtection*. Used for email protection, compliant with *digitalSignature*, *nonRepudiation* and/or *keyEncipherment* or *keyAgreement* bits.
  - *timeStamping*. Used to bind the hash of an object to a time, compliant with *digitalSignature* and/or *nonRepudiation* bits.

  - *OCSPSigning*. Used to sign OCSP responses, compliant with *digitalSignature* and/or *nonRepudiation* bits.

- *BasicConstraints*: used to specify if the certificate subject is a Certification Authority or not and how many intermediate CA certificates there could be after that in the certificate chain.
- *CRLDistributionPoint*: used to insert a URL where it is possible to access to the CRL emitted by the issuer of the certificate.
- *AutorityInformationAccess*: used to provide a URL where is possible to download the certificate of the issuer and/or a URL towards it is possible to send OCSP queries to verify the status of the certificate.
- *TLSFeature*: is an extension that is used to require an OCSP stapled response with the status of the certificate during the TLS handshake phase.

2.2 Certificate revocation

A certificate may be revoked before its expiration date due to several reasons: the CA that has issued that certificate has been compromised, the identity of the subject has not been formerly proven before to issue the certificate or the website private key got compromised.

It is up to the entity that accepts the certificate, the Relying Party (RP), to verify the current status of a certificate when connecting to a server. It must not just consider the leaf certificate but the whole chain of certificates up to the root one. There are two possible solutions to perform this kind of verification: Certificate Revocation Lists (CRL) and Online Certificate Status Protocol (OCSP).

2.2.1 CRL, OCSP

Certificate Revocation List (CRL) is a list of all non-expired revoked certificates, with date and reason of revocation, issued periodically and maintained by the certificate issuers. The list is signed by the CA that issued the certificates or by a Revocation Authority (RA) delegated by the CA. If the case is the latter the CRL becomes an indirect CRL (iCRL). It is important to re-issue periodically the CRL even if there are no new revoked certificates to add in order to guarantee the *freshness* of the list and avoid replay attacks due to the use of an old CRL.

The steps in revocation checking using CRL are summarized in Figure 2.2. The client tries to connect to the web-server starting a TLS handshake; during this phase the web-server provides to the client its own certificate; the client pause the channel negotiation and contact the CA to download a potentially large CRL. If the certificate serial number does not appear in the list, that means the certificate is still valid and the channel negotiation could proceed.

If a key gets compromised the time between its revocation and the issuing of the new version of the CRL may be not negligible. The process in time is represented by Figure 2.3. From the moment a web administrator realizes its certificate must be revoked to revocation time, the usage of the certificate in question by an attacker goes completely unrecognized (red zone). At revocation time the certificate has not been inserted in the new version of the CRL yet. This means that it is still possible for a RP to accept it (yellow zone). This time gap between revocation time and the issuing of the newer CRL depends on the CA policy. Subsequently (green zone) the

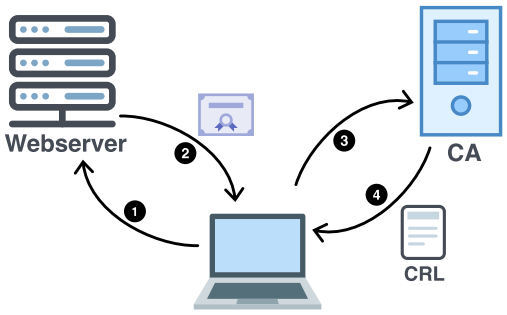

Figure 2.2. Client connecting to a web-server and fetching the CRL from CA after obtaining the certificate in the TLS handshake phase.

usage of the invalid key by an attacker is easily detected if the transaction is an online one. If the RP receives an old digitally signed document, it may still accept it since the verification of the signature is performed at signature time, when the certificate was still valid.

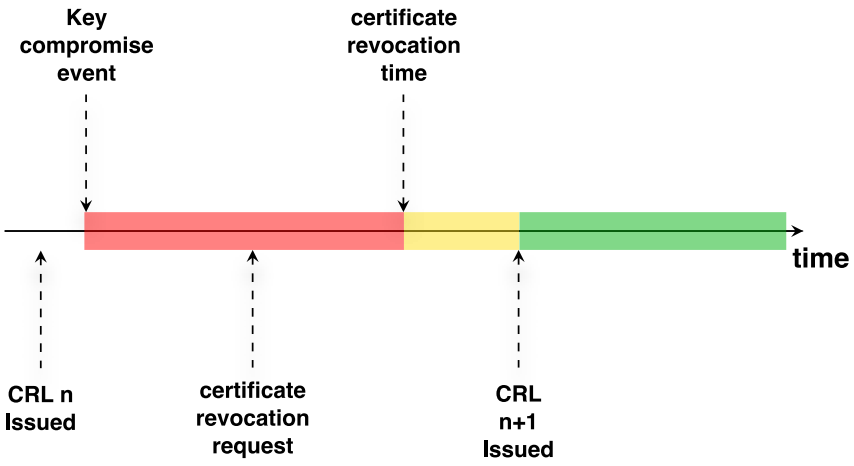

Figure 2.3. Certificate revocation timeline.

According to the X.509 CRLs version 2 specification in RFC-5280 [13], each CRL contains:

- *issuer* field, which specifies the issuer of the revocation list;
- *thisUpdate* field, that indicates when the CRL is issued;
- *nextUpdate* optional field, when the next CRL must be issued even if there are no changes with respect to the current one;
- *revokedCertificates* optional field, that is the list of certificates to be considered as revoked.

It is possible to add the following extensions to the CRL through the *crlExtensions* field:

- *Autority Key Identifier*, extension that identifies the public key corresponding to the private key used to sign a CRL.
- *Issuer Alternative Name*, extension that allows additional identities to be associated with the issuer of the CRL.
- *CRL number*, non-critical CRL extension that conveys a monotonically increasing sequence number for a given CRL scope and CRL issuer.
- *Delta CRL indicator*, critical CRL extension that identifies a CRL as being a delta CRL.
- *Issuing Distribution Point*, critical CRL extension that identifies the CRL distribution point and scope for a particular CRL, it indicates whether the CRL covers revocation for end entity certificates only, CA certificates only, attribute certificates only, or a limited set of reason codes.
- *Freshest CRL*, non-critical extension that identifies how delta CRL information for this complete CRL is obtained.
- *Authority Information Access*, non-critical extension used to specify a URL to get the CRL issuer certificate.

Each entry in the CRL is characterized by a *userCertificate* field that indicates the serial number of the revoked certificate, by a *revicationTime* field and by optional *crlEntryExtensions*, here listed:

- *Reason Code*, non-critical CRL entry extension that identifies the reason for the certificate revocation.
- *Invalidity Date*, non-critical CRL entry extension that provides the date on which it is known or suspected that the private key was compromised or that the certificate otherwise became invalid.
- *Certificate Issuer*, CRL entry extension that identifies the certificate issuer associated with an entry in an indirect CRL, that is, a CRL that has the *indirectCRL* indicator set in its issuing distribution point extension.
- *Hold Instruction Code*, deprecated non-critical CRL entry extension that provides a registered instruction identifier which indicates the action to be taken after encountering a certificate that has been placed on hold.[8]

Base CRLs, list of all the non-expired revoked certificates, may become considerable in term of size. For this reason, they could introduce an annoying delay during the connection to a server during the TLS handshake phase. To avoid that, 3 different options could be taken in consideration:

1. eliminate the certificate from the newer version of the CRL after its expiration date;
2. publish a complete CRL (base CRL number n) and then a CRL with just the differences with respect to it (delta CRL);
3. partition the CRL in different groups using the *CRL Distribution Point* extension in the certificate.

Option 1 reduces the size of the list, but, to be able to say if that certificate has been revoked or not, it is needed an archive of all past CRLs, otherwise it appears just as an expired certificate. Option 2 has faster downloads but introduces a burden to build the whole CRL. Option 3 allows the RP to download only the portion of CRL it needs but since the certificate issuer must maintain several CRLs it is more complex.

The alternative to CRLs is Online Certificate Status Protocol (OCSP), a client-server protocol used to check the status of a certificate *now*. It has the advantage to be fast with respect to CRLs but it does not give the possibility to answer for the validity in the past of the certificate.
As well as for CRL, the TLS handshake phase is paused to contact the OCSP server (Figure 2.4).

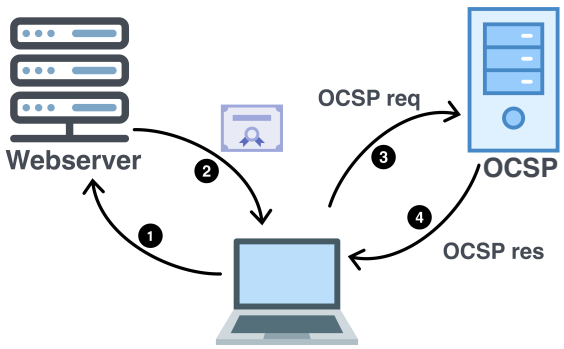

Figure 2.4. Client contacts the CA as OCSP responder to obtain information about the current status of the certificate it has fetched during TLS handshake.

The request could contain the service requested, the identifier of the certificate or chain of certificates whose status has to be verified and possibly some extensions the responder may process. When an OCSP responder receives a request it determines if:

1. the request is well formed;
2. the responder could provide the requested service;
3. the request contains all the information needed by the responder.

If just one of these conditions is not met, the OCSP responder generates an error message in the answer to the requestor. Otherwise, it will provide a definitive response.

All definitive responses should be signed by the CA who issued the certificate or by a Trusted Responder, whose public key is trusted by the requestor and it is independent from the CA for which it is responding or, even, by a CA Designated Responder who signs them using a different key based on the CA for which it is responding. The OCSP server signature must not be verified by the OCSP server itserlf.
The OSCP servers may take their information from public CRL stores, directly from the CA database or other OCSP servers.
RFC-6960 [14] states that the definitive response may have 3 possible values: *good*, *revoked* or *unknown*.

- *good*: positive response; indicates that no certificate with the requested certificate serial number, currently within its validity period, is revoked; this state does not necessarily means that the certificate was ever issued or that the time at which the response was produced is within the certificate's validity period.
- *revoked*: indicates that the certificate has been revoked, either temporarily (the revocation reason is certificateHold) or permanently; a revocation reason and revocation time has to be supplied.
- *unknown*: indicates that the responder does not know about the certificate being requested, usually because the request indicates an unrecognized issuer that is not served by that responder.

If a requestor receives as response *revoked*, it must reject the certificate; if the response contains *unknown*, it is up to the client to decide whether to reject it or try other possible mechanism to verify its validity, using CRLs for example.

OCSP may be subject to replay attacks or DoS attacks. If it protects against the former it is vulnerable to the latter and vice versa. To protect against replay attacks, in which the attacker saves a positive response of a valid certificate and re-use it after the certificate is revoked, the requestor may insert a nonce in the request to be included in the signed response; to face DoS, such as flood the server with requests, each requiring a real-time online digital signature, the server could use pre-computed responses with three timestamps:

- *thisUpdate*: the most recent time at which the status being indicated is known by the responder to have been correct.
- *nextUpdate*: the time at or before which newer information will be available about the status of the certificate.
- *producedAt*: the time at which the OCSP responder signed this response.

This solution allows replay attacks since it is not possible to insert a specific nonce into pre-computed responses.

Almost all web-browsers, nowadays, do not really rely on CRLs or OCSP queries. Those kind of revocation checks happen through the network, so they introduce potentially a performance penalty and if the CA services are unreachable due to some network problem the whole revocation check fails. An attacker could exploit this kind of situation to manipulate the responses by those services to make a certain certificate pass or fail the revocation check. OCSP could need up to 300 ms to perform a single query. Moreover, the CA managing this service can track the IP addresses of the users and the websites they are visiting. For those reasons, web-browsers rather prefer to

use special kind of CRL centralized sets, which are directly pushed periodically to the browsers. Example of them are CRLset used by the Chromium Project[15] and OneCRL used by Mozilla Firefox[16]. Those sets of revoked certificates are obtained from the major CAs or other CRLs and contains all the certificates that after a serious incident need to be revoked. This kind of solution is quicker than releasing a new update of the browser and wait until all the users have installed it. Furthermore, it complicates the life of an attacker. With this technique, in fact, he should block constantly the update of the CRLset on the victim web-browser, which is quite complex. The only web-browser that still allows to perform the classic OCSP queries is Mozilla Firefox. All other browsers directly blocked this kind of option.

2.2.2 OCSP Stapling and OCSP Must-Staple

To address the latency introduced by the OCSP request while connecting to a server, OCSP Stapling has been outlined in RFC-6066 [17] and then extended in RFC-6961 [18]. With OCSP Stapling it is not anymore the client that checks the status of a certificate each time, but it is the server that periodically makes an OCSP request to the OCSP server and caches the response together with a timestamp. It will provide the freshest, valid OCSP cached response in the *Certificate Status Request* extension during the TLS handshake phase so that could be seen as "stapled" to the certificate. If the client, who is expecting a stapled response, does not receive it, it could try to contact directly the OCSP server itself or decide to go for a soft-fail of the connection (showing a warning message to the user). OCSP Stapling helps also the client privacy to stay protected. In fact, by analyzing the OCSP requests or the CRL downloads a CA could keep trace of all the websites visited by each client. Since with OCSP Stapling it is the server that performs the request the CA could see just the server requests (Figure 2.5).

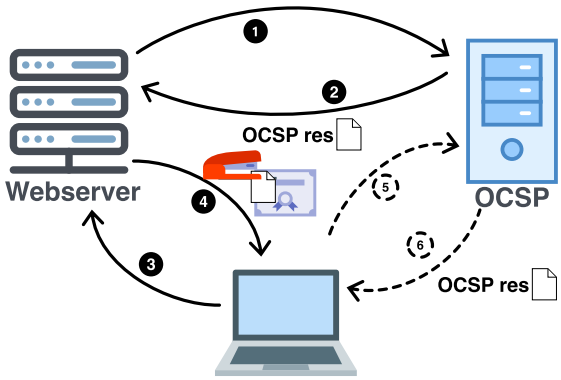

Figure 2.5. Client receives the stapled OCSP response directly with the certificate during TLS handshake. Optionally can contact the OCSP server (steps 5-6).

OCSP Must-Staple has been introduced to force the rejection of the certificate if the stapled response is not present when connecting to a server. It is an X.509 certificate extension[19] that informs the client to hard-fail the connection if the server does not provide a fresh and valid OCSP stapled response in the *Certificate Status Request* extension (CSR) during the TLS handshake (Figure 2.6). In order to be used, several conditions are needed:

- Certification Authorities must issue certificates with the *TLS Feature* extension set;
- Clients must support and recognize the *TLS Feature* extension;
- Web administrators must enable OCSP Stapling in their servers and must request certificates with *TLS Feature* extension;
- Servers must properly query the OCSP server and cache the response;
- OCSP responders must be always online and reliable.

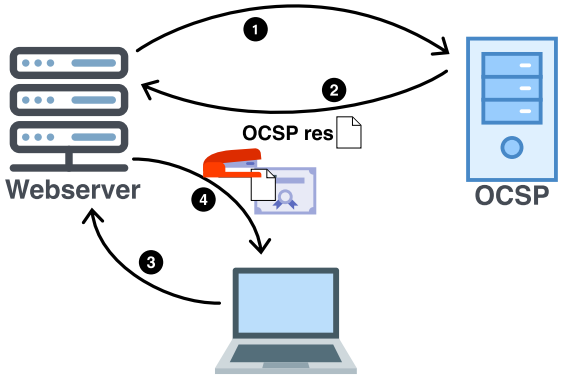

Figure 2.6. Client must receive the stapled OCSP response directly with the certificate during TLS handshake.

2.3 Certificate Transparency

Certificate Transparency (CT)[10] is an open, global auditing and monitoring system. It is based on public log of issued certificates and allows domain owners to verify that a fraudulent certificate has not been issued for their domains.

CT was born as a response to the DigiNotar attack in 2011[9], when an attacker was able to compromise the Dutch CA and issue over 500 fake rogue certificates that were used for man-in-the-middle attacks on traffic from Iran. Those attacks showed how the lack of transparency in CA modes of operation was a serious risk for the Web PKI. The first CT log was launched by Google in March 2013 filled with the certificates their web crawler had found. Then, in June of the same year, CT was standardized by IETF in RFC-6962 [20].

Log servers are the cornerstone of CT. They maintain a secure, public log of TLS certificates. Secure means that each log must be append-only: a certificate can only be added to the log at the end. Integrity and authenticity of the logs are obtained through Merkle Tree hashes[21] that prevents tampering and misbehavior. Anyone is able to query the logs (via HTTPS) and to verify that it is well behaved or that a specific TLS certificate has been properly added to the server (publicly auditable).

The core ideas of CT are to make impossible or, at least, very difficult, for a CA to issue certificates for a domain without making them publicly visible to the domain owner; the domain owner or the CA can verify whether the certificate have been mistakenly or maliciously issued through the open auditing and monitoring system CT must provide. In this way, CT pursuits the aim of protecting the users from fraudulent certificate issuance. CT framework counts several actors:

- Submitter, the one who submit the certificates (or partially completed certificates) to a log server receiving a promise to log the certificate within a certain amount of time called Signed Certificate Timestamp (SCT) in response;
- Logger, who manages the log server and provides each submitter a SCT;
- Monitor, a public or private service that watches for misbehaving logs or suspicious certificates, periodically downloads information from log servers, analyzes every new entry in the log, keeps copies of the entire log and verifies the consistency between published revisions of the log;
- Auditor, a lightweight software component that verifies the overall integrity of logs by periodically fetching a signed cryptographic hash (log proof) from the log server and verifies whether a particular certificate appears in the log.

The SCT created by the log server goes with the certificate throughout its whole life. There are three possible modes in which the SCT is delivered with the certificate. The most used one is represented in Figure 2.7: initially the CA submits a pre-certificate to the log server and receives the SCT as response; next, the CA attaches to the pre-certificate the SCT as an X.509v3 extension, signs the certificate and delivers it to the server operator. Figure 2.8 describes how the SCT could be delivered via TLS extension: the CA issues directly the certificate to the server operator; then, the server operator, submits it to the log server receiving the SCT; the server sends the SCT in the *signed_certificate_timestamp* TLS extension during TLS handshake with the client. Finally, as third alternative, SCT may be sent via OCSP Stapling (Figure 2.9): the CA provides the issued certificate to both the log server and the server operator; the log server sends back to the CA the SCT; the server make an OCSP query to the CA and gets the SCT in the OCSP response; the server includes the OCSP response in the *Certificate Status Request* extension during TLS handshake.

The format of the SCT according the RFC-6962 is described as follows:

```
struct {
          Version sct_version;
          LogID id;
          uint64 timestamp;
          CtExtensions extensions;
          digitally-signed struct {
              Version sct_version;
              SignatureType signature_type = certificate_timestamp;
              uint64 timestamp;
              LogEntryType entry_type;
              select(entry_type) {
                  case x509_entry: ASN.1Cert;
                  case precert_entry: PreCert;
              } signed_entry;
```

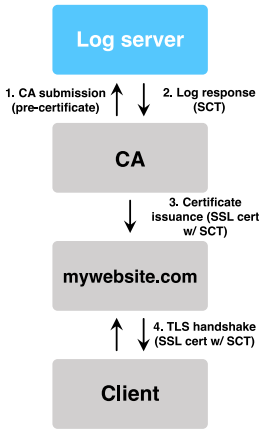

Figure 2.7. SCT via X.509v3 extension.

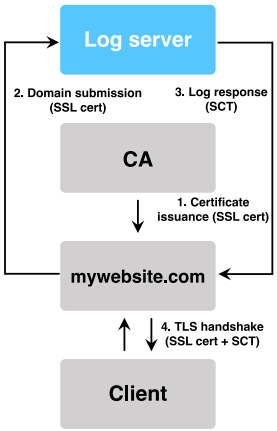

Figure 2.8. SCT via TLS extension.

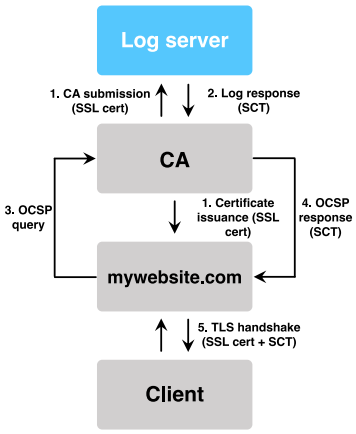

Figure 2.9. SCT via OCSP Stapling.

```
        CtExtensions extensions;
    };
} SignedCertificateTimestamp;
```

Where:

- *sct_version* is the version of the SCT protocol.
- *id* corresponds to the SHA-256 hash of the log's public key.
- *timestamp* is the current Network Time Protocol (NTP) Time measured since the epoch, ignoring leap seconds[1], in milliseconds.
- *entry_type* is the log server entry type.
- *signed_entry* is the leaf certificate or is the pre-certificate.
- *extensions* are future extensions to this protocol version (v1).

CT does not impose to use a particular configuration. In fact, the monitors or the auditors could be, respectively, operated by the CA and part of the web-browser or seen as standalone entities, providing free or paid services to CAs and server operators. As already said, it is possible to choose one of the previous modalities to carry the SCT together with the certificate. A TLS client that recognize, using an auditor, that a specific certificate is not logged, may refuse the connection and can use the SCT as evidence that the log has not behaved correctly. Moreover, auditors and monitors exchange information about logs via a gossip protocol to detect forked or branched logs (Figure 2.10).

[1] A leap second is a one-second adjustment that is occasionally applied to Coordinated Universal Time (UTC)

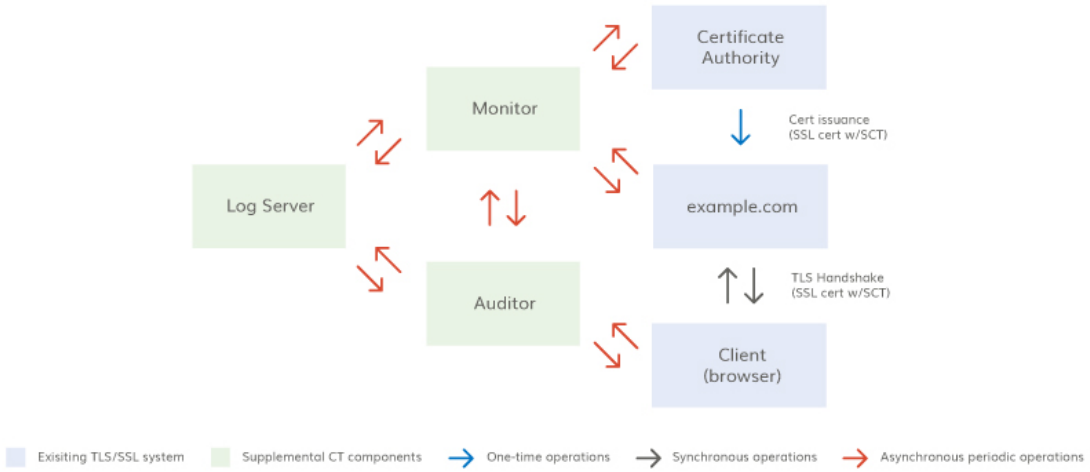

Figure 2.10. Example of CT model (Source [22]).

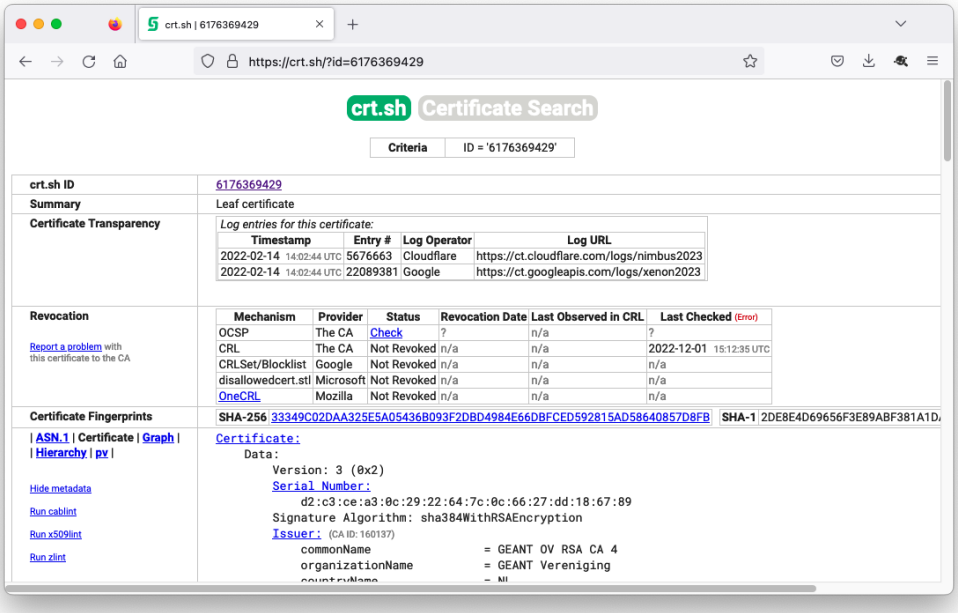

Figure 2.11. Sectigo's tool crt.sh showing the result for www.polito.it certificate.

Browsers requirement for CT are different for each of them. More in detail, Google Chrome requires a certificate with an SCT if has the field *NotBefore* after 30 April 2018[23]. Safari requires more than one SCT to prove the validity of the certificate. If the *NotBefore* field is set after 21 April 2021, the certificate must have 2 SCT in log server operated by different providers in case the validity period is less or equal than 180 days. If the validity period is between 180 and 398

days (the maximum suggested by the CA/Browser Forum[7]), then it must have 3 SCT with maximum two of them in log servers operated by the same provider. Then, the longer is the validity period the higher is the number of SCTs to embed in the certificate[24]. Mozilla Firefox does not require strictly the presence of a SCT.[25]. The 2022 recognized log operators by the Chromium project[26] are: Google, Cloudflare[27], TrustAsia[28], DigiCert[29], Let's Encrypt[30] and Sectigo[31].

At the moment exists a tool managed by Sectigo called *crt.sh*, which allows to check the different revocation status check techniques used for a specific certificate, together with its status information. More over it keeps trace of the log servers in which the certificate can be found (Figure 2.11).

2.4 Root CA Management

Various models of PKI are possible. The simplest one is the hierarchical PKI (Figure 2.12): a tree rooted as a self-signed root CA. Due to its simplicity it is easy to build the certification path between two End Entities (EEs) and it is natively supported by all the applications. The existence of conflicting legal, commercial or political policies across the world caused the presence of a "forest" of these hierarchies.

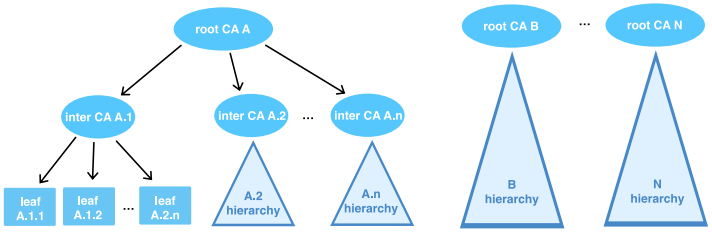

Figure 2.12. Hierarchical PKI model.

Two hierarchical PKI may unilaterally or bilaterally trust each other by issuing one root CA the certificate for the other root CA (cross-certificate). This kind of cross-certificates are not automatically recognized by all the applications (they are not able to determine which certificate chain consider). In addition, to gain complete trust among all the N hierarchical PKIs, $N(N-1)/2$ cross-certificates are needed. A mesh PKI in obtained in this way. For those reasons this solution is rarely adopted.

As main alternative to hierarchical PKIs, to simplify the CA management operations and trust transitivity, the bridge PKI model has been introduced. The core idea is to have a bridge CA, which is a cross-certificate with each root CA that does not certify any other CA or EE. Even though it is not automatically recognized by all applications yet, complete trust is achieved with just N cross-certificates.

2.5 Validation Process

In the RFC-5280 [6] is described the base algorithm that a RP should follow to perform the validation of a certificate chain received during the TLS handshake phase.

Given a certificate i in a chain made of certificates from i to n the following checks must be performed:

1. Verify the digital signature of the certificate i using the key of the i-1 certificate in the chain. If the certificate is a self-signed the key to be used is the subject key.
2. Check whether the current time is between the *notBefore* and *notAfter* timestamps.
3. At the current time the certificate must not be revoked. It is possible to use one of the above described solutions to get this information.
4. Verify the presence of policies to be applied to certificate i.
5. If the certificate is not the leaf certificate, verify that the *Basic Constraints* extension is set to `CA:True` and decrement the path length value. Moreover, if *Key Usage* extension is defined check whether is set the bit corresponding to *keyCertSign*.
6. Process the critical extensions and then the non-critical extensions.
7. repeat for certificate i+1 with issuer set to the subject of certificate i.

If all the checks are successful up to the certificate n, the leaf certificate, it means the validation succeeded and the TLS handshake can continue.

Chapter 3

TLS Interception

3.1 Transport Layer Security

Transport Layer Security (TLS) protocol is the evolution of the old Security Socket Layer (SSL) protocol and was released in 1999. This protocol is used to establish a secure communication channel over a Transmission Control Protocol (TCP) connection by means of an handshake. This protocol achieves server authentication and optionally client authentication through an asymmetric challenge-response mechanism; optionally message confidentiality using symmetric encryption; message authentication and integrity computing a Message Authentication Code (MAC) and protection against filtering and replay attacks since it is used over TCP and exploits it segment sequence numbers. The last version of TLS is 1.3 and it got standardized in RFC-8446 [32] in 2018, but TLS version 1.2[33] is still quietly used at the moment.

The TLS handshake has the purpose to make the client and the server agree on a set of algorithms for confidentiality and integrity, exchange true random numbers between them to generate a subsequent symmetric key by means of public key operations, negotiate a session ID for the connection and exchange the necessary certificates (server must always send its certificate). In TLS 1.2 the messages during this phase are:

- *ClientHello*, this message contains the SSL/TLS preferred version by the client, a 28 pseudo-random bytes number, a session ID (if is the starting message of a session, then it is 0, otherwise different from 0), a list of the cipher suites[1] supported by the client and a list of compression methods supported by the client.
- *ServerHello*, this message contains the SSL/TLS preferred version by the server, a 28 pseudo-random bytes number, a session ID (a new one if the *ClientHello* message session ID is 0, otherwise the same session ID proposed by the client), the highest in common cipher suite supported by both and the compression method chosen by the server.
- *Certificate*, if this message is sent by the server, it contains the server X.509 certificate, whose Subject or Subject Alternative Names must be the same of the identity of the server;

[1] A cipher suite identifies a set of algorithms. It contains a key exchange algorithm, a symmetric encryption algorithm and an hash algorithm.

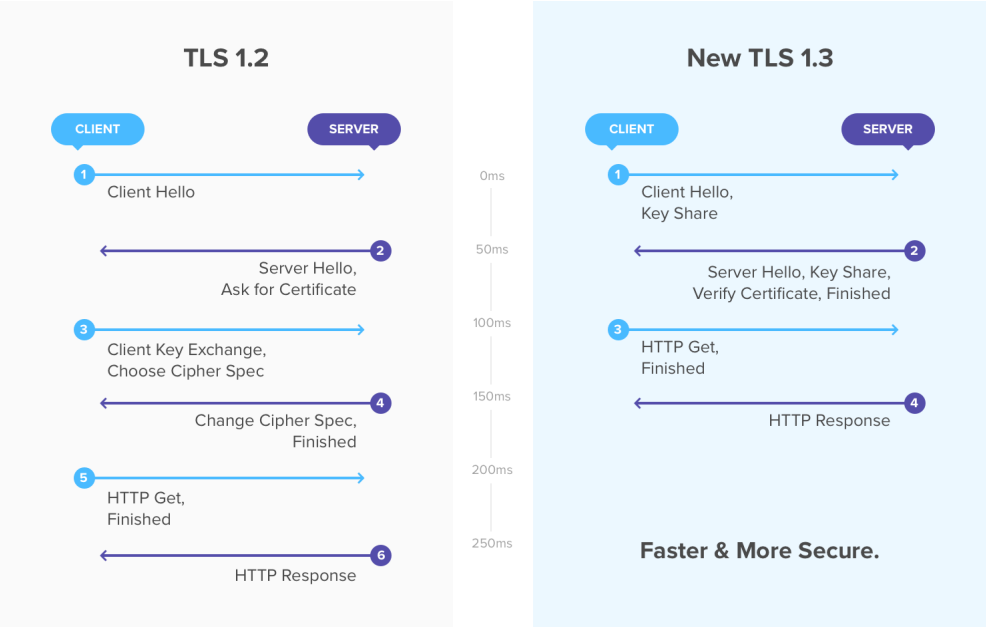

Figure 3.1. TLS handshake. On the left version 1.2 handshake. On the right the version 1.3 handshake (Source: [34]).

it must be used for digital signature and if specified in the Key Usage extension it could be used also for encryption. On the other hand, if this message comes from the client, it contains a certificate identifying it that must be digitally signed by one of the CA in the *Certificate Request* message sent by the server.

- *Certificate Request*, is used to request client authentication by the server. It specifies a list of trusted CAs by the server. Only a client certificate signed by one of them is considered allowed.
- *Server Key Exchange*, it carries the server public key used for key exchange in case the server key can be used only for digital signature, anonymous or ephemeral Diffie Helmann (DH) is used to establish the pre-master secret[2], there are export problems that force the usage of ephemeral RSA or DH keys or Fortezza[3] is used. This is the only message directly signed by the server.
- *Client Key Exchange*, is the client generated symmetric keys using a pre-master secret encrypted with the server RSA public key or using DH or Fortezza.
- *Certificate Verify*, this message is present only if the client is sending its certificate to the server. It is an explicit test signature performed by the client. It consists in the hash of all the previous messages in the handshake and encrypted with the client private key. The purpose of this message is to identify the client and reject fake ones.

[2]The master secret used to generate the different keys per TLS session is obtained by a pre-master secret generated with public key cryptography in combination with the two random numbers sent by the client a the server

[3]An old key agreement protocol

- *Change Cipher Spec*, theoretically is a protocol on its own and not part of the handshake. It triggers the change of the algorithms (just negotiated) to be used to protect the next messages in the TLS session.
- *Finished*, it is the first message protected by the algorithms negotiated between client and server. It contains all the hash of all the previous messages but the *Change Cipher Spec* one encrypted with the master secret just generated. This message is different for the client and the server.

Although has been proved that TLS 1.2 worked fine, there are security and privacy concerns after years of patching and revisions. Moreover, this version of the protocol implies unnecessary network and performance overheads[35].

Researchers and attackers, have discovered several vulnerabilities in many TLS 1.2 components such as ciphers, digital signatures and key exchange algorithms. Some are implementation bugs like *Heartbleed*[4], others are protocol vulnerabilities due to bad design decision in earlier TLS versions, such as *ROBOT*[5][36]. The majority of the bugs have been fixed in TLS 1.2 but nothing could be done for protocol design mistakes. This is the reason why in TLS 1.3 the handshake phase has been completely redesigned. The most alarming flaw in TLS 1.2 protocol is that it is subject to downgrade attacks. That because it does not protect with digital signature the messages before the *Change Cipher Spec* message. Consequently, an attacker, could manipulate the cipher suite negotiation and force the channel to use a vulnerable algorithm.

The version 1.2 of the protocol, as seen from its handshake, requires the negotiation of different parameters before to start a protected communication. Generally, in term of time, it requires 2 additional Round Trip Time (RTT) with respect to the normal TCP unprotected communication. That is problematic for those devices which requires a low bandwidth and low power consumption.

Lastly, TLS 1.2 is seen problematic for the privacy of the users over the web. To allow the usage of different certificates for the different virtual host running on the same public IP address, the *Server Name Indication* (SNI) handshake extension has been introduced. That contains the hostname of the server the client wants to connect to. The problem is that this name is sent in clear.

TLS 1.3 solves these issues. It gives up backward compatibility in favor of proper security design. It has been projected from scratch to provide similar functionalities of TLS 1.2, but better form performance, privacy and security point of view.

First of all, it does not use anymore all key exchange algorithms but DH Ephemeral (DHE) or Elliptic Curve DHE (ECDHE) with a set of well known to be secure parameters. This avoids the need to negotiate those parameters with the server when starting a connection. Secondly, it does not support anymore vulnerable cipher suites such as those with Cipher Block Chaining (CBC) mode or RC4 algorithm, but only the modern cryptographic algorithms with Authenticated Encryption with Associated Data (AEAD) mode. Those cipher suites are still used in TLS 1.2 for legacy support. The handshake has been made more secure too. The newest version of the protocol requires the server to digitally sign the whole handshake, including the cryptographic

[4]The Heartbleed bug exploits a vulnerability in the old version of OpenSSL (CVE-2014-0160) which allowed an attacker to read the content of the RAM to steal sensitive data such as username and password or/and server's private key

[5]ROBOT is a vulnerability that allows performing RSA decryption and signing operations with the private key of a TLS server. To protect against it RSA should not be used for encryption.

negotiation, protecting the communication against downgrade attacks. The digital signature has been also improved implementing the RSA Probabilistic Signature Scheme (RSA-PSS), which is immune to cryptographic attacks that affect previous TLS versions signatures.

The handshake has been simplified according to the simplified set of ciphers and key negotiation algorithms. Instead of a 2-RTT handshake as in TLS 1.2, in TLS 1.3 there is a 1-RTT handshake. In this case the *ClientHello* message contains the client random number, the supported protocol version and the supported cipher suites. In addition, the client could send in this message the DH key shares because can almost be sure about what will be the server's parameters. If the server, unlikely, does not support the parameters chosen by the client it will respond with a *HelloRetryRequest* error. The client, at this point, will start a new handshake with new parameters.

A further optimization introduced in TLS 1.3 is the 0-RTT handshake. This handshake is made possible since both, the client and the server, store a pre-shared key generated from a complete handshake at the first connection. In this way, from the second connection on, there is no need for a new handshake phase. The client could send in its first message data protected with a key generated from the pre-shared key. The communication is protected with no additional performance loss with respect to unprotected transmissions. Since the key used to protect the interactions between the client and the server depends on a pre-shared key, the 0-RTT handshake could not provide forward secrecy[6] and requires that data stored at both sides to be synchronized.

The handshake messages exchanged in TLS 1.3 are the following:

- *ClientHello*. In this message there are the client random number, the supported version of TLS and the supported cipher suites. It contains some extensions used for key negotiation. There is a *key_share* extension which contains the (EC)DHE share of the client; *signature_algorithms*, that is the list of supported algorithms for digital signature; *psk_key_exchange_modes*, a list of supported modes for a PSK and *pre_shared_key* if there is one.

- *ServerHello*. This message contains the server random number, the selected version of the protocol and the selected cipher suite. The *key_share* extension contains the server (EC)DHE share. The *pre_shared_key* extension, if present, contains the selected PSK.

- *EncryptedExtensions*. Contains answers to other non-cartographic extensions of the client. It is sent encrypted.

- *CertificateRequest*. This is the request that the server could send to the client to require its certificate. It is sent encrypted.

- *Certificate*. The X.509 certificate that has to be sent from the server and optionally by the client. It is sent encrypted.

- *CertificateVerify*. It is the digital signature of all the handshake up to this message signed with the private key of the party who is sending the message. It is sent encrypted.

- *Finished*. It is the MAC computed over the whole handshake up to this message. It is sent encrypted.

[6] If the key used for encryption get compromised it does not affect the past or the future traffic encrypted with it

Lastly, in TLS 1.3 the user privacy protection has been enhanced too. Now also the SNI extension is sent encrypted. In this way an attacker can not track the user HTTPS history from leaking information from the server.

3.2 Why TLS interception

In this section use cases in which is needed to access to plain HTTP traffic are described. The advent of TLS affected negatively those cases and that is why the practice of the TLS interception earned its importance.

To start, a government could impose to an Internet Service Provider (ISP) to provide to it a portion of its traffic. An ISP may be also legally obliged to store its customers traffic for a certain amount of time by the government. To block the access to some illegal website, a government could also impose content filtering policies. Filtering could happen at different level. TLS does not affect filtering up to Transport Layer, but filtering at higher level requires the access to the plain traffic.

A part from government driven use cases, there are also security reasons why the clear HTTP traffic should be analyzed. First of all, to allow Intrusion Detection Systems (IDS) or Intrusion Prevention Systems (IPS) to work properly. If the content of the traffic is not visible it is impossible for them to apply their policies. It is the same if some firewall policies are used to filter the traffic at application level. Companies could also retain their traffic to demonstrate compliance with standards or want to monitor outgoing traffic in order to prevent data loss.

The analysis of the HTTP traffic is useful for troubleshooting in case of errors or to launch Quality of Service (QoS) analysis or load balance the traffic. The content traffic must be in clear also if some other parental control filtering should be applied or if the service offered must be paid based on the websites visited by the customer.

All those cases are impossible or partially possible if TLS comes to play. For this reason, TLS interception has been introduced. How TLS interception works it is explained in the next section.

3.3 How TLS interception works

TLS, as already said, is a way in which a secure communication channel is negotiated between a client and a server. That make impossible to perform all those security, privacy and performance analysis of the traffic incoming and outgoing a network. This thesis studies the solutions based on the TLS session splitting, exploiting proxy applications and antivirus tools. This kind of TLS interception solutions' core concept is really simple. The idea is to break the channel in two channels and place in the middle of those two channel a third entity, a proxy machine. The proxy machine will negotiate both of the channels. It will act as client towards the server and as server towards the client (Figure 3.2).

In order to make everything works fine, a further step is necessary. Each interception product have a private-public key-pair, together with a self-signed certificate. This certificate must be installed in the client's Trusted Certification Authorities store. If this step is skipped, when the client tries to connect to the server (using a browser configured to use the proxy on a specific port) will always show a warning message saying that the issuer of the website certificate is unknown and so untrusted.

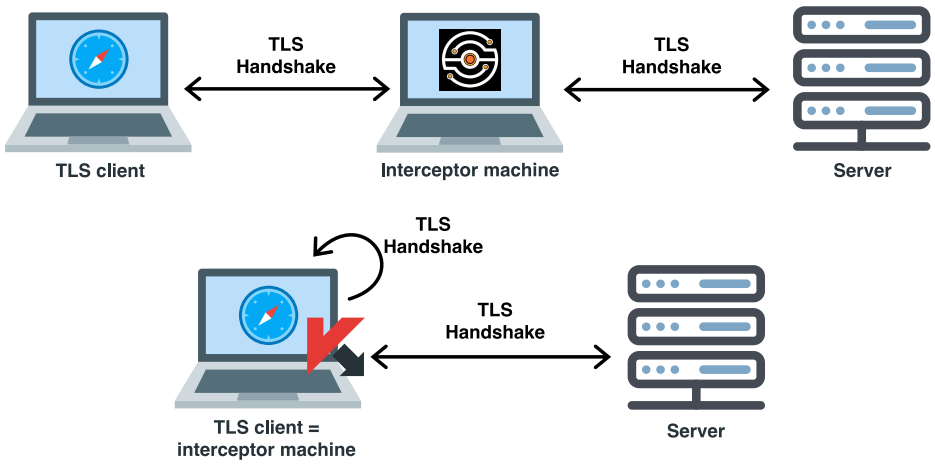

Figure 3.2. TLS client machine connecting through its browser to the server passing trough a proxy machine above. TLS client connecting to the server using the antivirus as TLS interception product below.

When the proxy machine receives the connection request from the client, it will send a *ClientHello* message to the server and complete a proper TLS handshake. Then, on-the-fly, it will create a new certificate starting from the server original one and sign it with its own private key. It will use this just created X.509 certificate to negotiate a channel with the client. The client and the sever will not recognize this manipulation.

The trick is that the proxy machine can read the plain traffic passing through it. So in that point of the network could be performed any kind of analysis described in the previous section.

The interception of the TLS traffic is not performed only by proxies. There are also antivirus tools that do this kind of operation. They are directly installed on the client machine instead of on a different one. In this thesis have been tested both the categories of TLS interception products.

Chapter 4

Related work

4.1 Certificate validation

Yabing Liu et al.[4] in their paper *An End-to-End Measurement of Certificate Revocation in the Web's PKI* published at the end of 2015 analyzed the certificate revocation process from the point of view of the web administrator and from the one of the clients.

Initially they studied the website administrators' behavior when they revoke a certificate. They used 74 full IPv4 scans collected by Rapid7[37] between October 2013 and March 2015. Selecting just the valid certificates from the scans, they checked everyday their revocation status starting from October 2014, discovering that over the 8% of them has been revoked and 1% of them continues to be advertised anyway. They also discovered that OCSP Stapling was supported just by the 3% of the certificates.

Subsequently they observed the delay introduced by the certificate revocation check in case the CRL method is used or the OCSP one. They showed that in both of the cases there is a delay, but significantly less in the case of OCSP.

Furthermore, they moved their attention on the revocation check performed by the clients. They wanted to analyze both, desktop and mobile browsers' certificate revocation check features. They developed a test suite of 244 distinct certificate configurations. The results evidenced that the revocation status check is not performed so many times. Only Mozilla Firefox checked the validity status of the leaf certificate, only if this one had an OCSP responder embedded in it. No one of the mobile browsers checked the revocation status of the certificates.

Lastly they wanted to study more in detail the Google Chrome revocation status check process. They focused on CRLSets used by it. They discovered that only the 0.35% of revoked certificates was present it the CRLSet.

D. McLuskie and X. Belleken in their paper *X.509 Certificate Error testing* in 2018 [1] try to analyze the different behavior of TLS clients facing wrong certificate configurations. Their methodology starts with the individuation of a set of twelve possible certificate erroneous configuration. Their main focus was on wrong values used in the Distinguished Name (DN) parameters identifying the subject of the certificate, the validity period values, certificate serial number and the certificate digital signature.

To generate their tests, they used a configuration file read by a Python script. The script generates a proper directory for each test, an Apache server configuration for each test, an HTML page for each configuration and, obviously, the needed certificates for each test.

They run their tests against two possible configurations. A Windows 10 machine using as TLS client Mozilla Firefox version 60 and a macOS 10 machine using Safari version 11.1.

Table 3: Test configuration 1 results

Test Option	Expected Result	Result
WrongKey	Fail ⊗	Fail ⊗
SwapStartEnd	Fail ⊗	Fail ⊗
MissingStart	Fail ⊗	Fail ⊗
MissingEnd	Fail ⊗	Fail ⊗
LongEnd	Pass √	Pass √
NullCN	Fail ⊗	Pass √
FOOCN	Fail ⊗	Fail ⊗
TabCN	Fail ⊗	Fail ⊗
BackspaceCN	Fail ⊗	Fail ⊗
LongOU	Fail ⊗	Pass √
LongRandomSerial	Fail ⊗	Pass √
SameSerial	Fail ⊗	Fail ⊗

Figure 4.1. Results of test with Windows configuration in *X.509 Certificate Error testing* (Source: [1]).

Table 4: Test Configuration 2 Results

Test Option	Expected Result	Actual Result
WrongKey	Fail ⊗	Fail ⊗
SwapStartEnd	Fail ⊗	Fail ⊗
MissingStart	Fail ⊗	Fail ⊗
MissingEnd	Fail ⊗	Fail ⊗
LongEnd	Pass √	Pass √
NullCN	Fail ⊗	Incomplete
FOOCN	Fail ⊗	Fail ⊗
TabCN	Fail ⊗	Fail ⊗
BackspaceCN	Fail ⊗	Fail ⊗
LongOU	Fail ⊗	Pass √
LongRandomSerial	Fail ⊗	Fail ⊗
SameSerial	Fail ⊗	Pass √

Figure 4.2. Results of test with macOS configuration in *X.509 Certificate Error testing* (Source: [1]).

Their tests results (Figure 4.1 and Figure 4.2) showed how the two different system uses OpenSSL differently since there are no strict rules the programmers must follow. Their major concern was about the fact that OpenSSL allowed the insertion of escape characters in the DN fields of the certificates that could lead to SQL injection attacks. Moreover they are not able to generate the tests equally on the two different configurations.

Johanna Amann et al. conducted an interesting study on the new extensions born after the clamorous case of DigiNotar CA certificate that got compromised. Their research can be found in the paper *Mission Accomplished? HTTPS security after DigiNotar* [5].

Among the new security solution, Certificate Transparency (CT) has been widely analyzed. They retrieved the certificates from a TLS scan based on DNS names for a total of 193M domains. From those certificates they extracted the SCTs and noted who was the CA who issued that certificate, how many SCTs are embedded in that certificate and if those SCTs are valid or not. There are several cases in which the SCTs inserted in the certificates were not valid. In the majority of the times, when the SCTs are valid, they analyzed in which log server they are stored. In this way they had the possibility to obtain a measure of the coverage of CT over the internet websites in 2017.

They discovered that the CAs obtain their SCTs mainly from Symantec[38], Google and DigiCert[29]. This tendency caused a concentration of trust towards them. In addition, 16.4K certificates contained just a single SCT. In the most of the cases pointing to a log server managed by Google.

Analyzing the data from the ICSI Notary, a large scale monitoring system of the TLS ecosystem, they collected information about the version of the protocol (from SSLv3 to TLS 1.3) negotiated across 5 years, from 2012 to 2017. They saw a progressive step of the community towards security. In 2017 the most used version was TLS 1.2.

Taejoong Chung et al. in 2018 conducted an interesting study, *Is the Web Ready for OCSP Must-Staple?*[11], in which they wanted to understand what was the actual support to Must-Staple by CAs, browsers, web-server applications and web-server administrators.

They first focused on a set of 112,841,653 valid certificates, which support OCSP, by Censys[39]. They searched in those certificates for the OCSP URL in the *Authority Information Access* extension. They verified the support for OCSP Must-Staple discovering that just the 0.02% supported it. Most of them (97.3%) were issued by Let's Encrypt [1]. Then they performed the same check on the Alexa Top-1-Million domains showing a increasing usage of OCSP Stapling (Figure 4.3). The OCSP Stapling adoption reached about the 38% in 2018. Only 100 certificates from them supported OCSP Must-Staple.

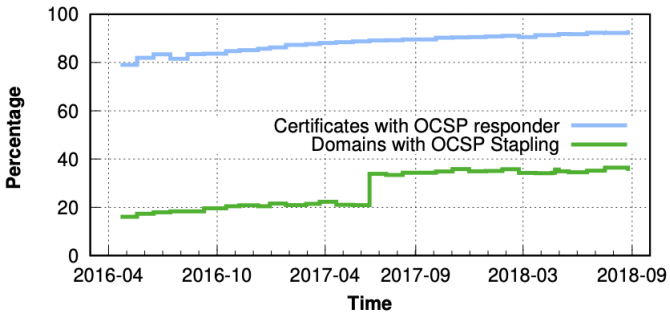

Figure 4.3. OCSP and OCSP Stapling adoption from monthly-fetched Censys Alexa Top-1-Million dataset since May 21st 2016 to August 2018 (Source [11]).

Then, they decided to test the OCSP responders. Their objective was to see how reliable they are. They picked the OCSP responders they obtained previously from the Censys dataset. They took 50 random certificates managed by each responder and periodically (every hour) checked the status of those certificates. They distributed their clients in 6 vantage points over the world: Oregon, Viriginia, São Paulo, Paris, Sydney and Seoul. They discovered that 36.8% of them experienced at least a few hours outage.

Then, they tested the browsers' support for OCSP Must-Staple. They bought a domain and disabled OCSP Stapling on purpose. They observed the behavior of Mozilla Firefox (Windows, Linux, macOS), Google Chrome (Windows, Linux, macOS), Safari (macOS), Opera (Windows, macOS), Internet Explorer (Windows), Microsoft Edge (Windows) as desktop clients and Safari (iOS), Mozilla Firefox (iOS, Android) and Google Chrome (iOS, Android) as mobile clients. They studied whether they insert the Certificate Status Request extension in the TLS handshake or not and if they respected the Must-Staple extension.

	Desktop Browsers											Mobile Browsers				
	Chrome 66			Firefox 60			Opera		Safari	IE	Edge	Safari	Chrome		Firefox	
	OS X	Lin.	Win.	OS X	Lin.	Win.	OS X	Win.	11	11	42	iOS	iOS	And.	iOS	And.
Request OCSP response	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Respect OCSP Must-Staple	✗	✗	✗	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Send own OCSP request	✗	✗	✗	-	-	-	✗	✗	✗	✗	✗	✗	✗	✗	✗	-

Figure 4.4. Table containing Taejoong Chung et al. results of browsers tests. ✓ indicates that the browser requested the status of server certificate in TLS handshake. ✗ indicates that the client accepted anyway the certificate even if no stapled response is received with the server certificate. (Source [11]).

[1] Let's Encrypt is a free, automated, and open certificate authority

Figure 4.4 summarizes the results obtained by Taejoong et al. of Must-Staple tests against browsers. All of them requested the status of server certificate in the TLS handshake. Only Mozilla Firefox, desktop version and Android mobile version, respected the Must-staple extension and refused the connection. None of the browsers selected performed their own request to the OCSP responder.

Lastly, they tested whether Apache and Nginx support OCSP Stapling. They noticed that none of them pre-fetched the OCSP response. That introduced a delay in the connection. Moreover, Apache provided expired OCSP responses from the cache and discards the previous and valid responses when it faces an error communication with the OCSP responder.

4.2 TLS interception

Xavier de Carné de Carnevalet and Paul C. van Oorschot, from Carleton University in Canada, conducted a theoretical study of all the possible TLS interception solutions proposed [40].

They analyzed different categories of technique to obtain access to the plain HTTP traffic. The first category, session splitting and key sharing, is the one which relates the most with this thesis. It is the simplest kind of solution to implement the TLS interception. It just requires one of the endpoint of the communication and one third party (middlebox) to be compatible. Core idea of this category is to split the channel into multiple channels or to share TLS keys or secrets after they have been negotiated. Part of this category are solutions like:

- *TLS session splitting*, the classic man-in-the-middle pattern. In this case the middlebox acts as sever towards the client and as client towards the server. It is trusted by the client to verify the validity and correctness of the server certificate. This is the preferred method by antivirus tools, parental control, malware, student or employee monitoring systems, ad-blocking softwares and enterprise network appliances.
- *Certificate private key sharing*, as the name suggests the private key of the server is shared with a trusted middlebox who can use it to decipher the transmission or to actively impersonate the server.
- *Sharing DH key exchange private shares*, this method gives up Perfect Forward Secrecy (PFS) making the DH shares static. Then shares them among the parties. Every of them could reconstruct the keys from those shares.
- *Client session key sharing*. This solution fits well when splitting the TLS session is not acceptable at client-side. Basically, browsers like Google Chrome or Mozilla Firefox can save TLS secrets into a *SSLKEYLOGFILE*. Some antivirus tool chooses this solution to passively decipher the communication reading that file. In this way they are able to apply some filtering policy without splitting the TLS communication.

Their article discuss also more complex solutions, not so related to what is studied in this thesis.

4.2.1 Certificate validation in TLS interception products

Mostly related to this work are the following articles. *On the Validation of Web X.509 Certificates by TLS interception products*[2] written by Ahmad Samer Wazan et al. is a study of how different

TLS interception products, both proxies and antivirus tools, react to misconfiguration of servers or to server certificates which do not respect the standard.

The TLS interception products they analyzed are the ones in Figure 4.5

TABLE 1
List Of Tested Products

Product	Version (Test Date)
Avast Antivirus Gratuit	17.4.2294 (2017), 19.5.2378 (2019)
Kaspersky Total security	17.0.0.611 (2017), 19.0.0.1088 (2019)
AVG Internet Security	17.4.3014 (2017), 19.5.3093 (2019)
ESET Internet Security	10.1.210.2 (2017), 12.1.34.0 (2019)
Squid	3.3.8 (2017), 4.6(2019)
Charles Web debugging Proxy	4.1.2 (2017), 4.2.8 (2019)
Mitmproxy	0.9.2 (2017), 4.0.4 (2019)
Telerik Fiddler	4.6.20171.14978 (2017), 5.0.20192.25091(2019)

Figure 4.5. TLS interception products used in *On the Validation of Web X.509 Certificates by TLS interception products*(Source: [2]).

Firstly, they varied the values of the *Subject Alternative Names* (SAN) extension of the server certificate. This extension should contain only DNS names or public IP addresses and if used together with *Common Name* (CN) in the subject description section of the certificate, it must contain one entry that matches with that CN value. What they wanted to study is whether the Relying Party (RP), the TLS interception product, would have accepted anyway a certificate that do not respect the standard. They tested all the possible combination of values assigned to the SAN extension and CN. Their results are collected in Figure 4.6. Most of the antivirus products opts for a delegated validation. That means that the validation of the certificate in done together with the browser. Kaspersky, among them, has a different behavior. It accepts the certificate if at least one between Subject CN and SAN extension contains a valid value. It soft-fails all cases in which SAN is not defined. Among the proxies, the differences in their behavior, accepting wrong certificates with respect to the standard, are more evident.

Then they moved their attention to the *Key Usage* and *Extended Key Usage* extensions. As described in Section 2.1, these extensions are used to specify the purpose of the asymmetric key associated to the server. Based on the cipher suite negotiated between the client and the server, possible value are:

- *TLS_RSA**: those cipher suites choose RSA as key exchange algorithm. In this case the value that should be assigned to the KU extension is *keyEncipherment*.
- *TLS_ECDHE_RSA**: if this is the case, the key exchange algorithm is Elliptic Curve Diffie-Hellman Ephemeral (ECDHE). The private RSA key of the server will be used to sign the ECDHE public key and the associated parameters. The KU should be *digitalSignature*.

The EKU is managed separately according to the standard. It must be compliant with the value inserted in the KU. The key of the certificate can be used just for one value inserted in the

TABLE 2
Multiple FQDN Web Server Identities

	Antivirus								Proxies									
	Avast		Kaspersky		AVG		ESET		Squid		Charles		Mitm		Fiddler		Standards	
	S1	S2	S1	S2	S1	S2	S1	S2	S1	S2	S1	S2	S1	S2	S1	S2	S1	S2
i) SCN= sana1.fr, SAN-DNS=sana1dns.fr	dV	dV	W	A	dV	dV	dV	dV	W	A	iV	iV	W → R	A	W	A	I	V
ii) SCN= null, SAN-DNS=sana1dns.fr	dV	dV	W	A	dV	dV	dV	dV	R	A	iV	iV	W → R	A	W	A	I	V
iii) SCN= sana1.fr , no SAN-DNS	dV	dV	A	W	dV	dV	dV	dV	A → W	R	iV	iV	A	W → R	A	W	?	I
iv) SCN=null, no SAN-DNS	dV	dV	W	W	dV	dV	dV	dV	R	R	iV	iV	W → R	W → R	W	W	I	I
v) SCN=null, SAN-DNS=sana1.fr , sana1dns.fr	dV	dV	A	A	dV	dV	dV	dV	A	A	iV	iV	A	A	A	A	V	V

*Where: **S1** = sana1.fr, **S2** = sana1dns.fr, **dV** = delegated Validation, **iV** = incorrect Validation, **A** = Accept, **W** = Warn, **R** = Refuse, **I** = Invalid certificate w.r.t standard, **V** = valid certificate w.r.t standard.*

TABLE 3
IP Address Server and/or fqdn Identities

	Antivirus								Proxies									
	Avast		Kaspersky		AVG		ESET		Squid		Charles		Mitm		Fiddler		Standards	
	S1	IP	S1	IP	S1	IP	S1	IP	S1	IP	S1	IP	S1	IP	S1	IP	S1	IP
vi) SCN= null , SAN-IP=192.168.133.149	dV	dV	W	A	dV	dV	dV	dV	R	R	iV	iV	W → R	W → R	W	W → A	I	I
vii) SCN= sana1.fr ,SAN-IP–192.168.133.149	dV	dV	A	A	dV	dV	dV	dV	W	R	iV	iV	A	W → R	A	W → A	V	I
viii) SCN= null, SAN-IP=192.168.133.149 SAN-DNS=sana1.fr	dV	dV	A	A	dV	dV	dV	dV	A	R	iV	iV	W → A	W → R	A	W → A	V	I
ix) SCN= null ,No SAN-IP ,SAN-DNS=sana1.fr	dV	dV	A	W	dV	dV	dV	dV	A	R	iV	iV	A	W → R	A	W	V	I
x) SCN= null ,No SAN-IP ,SAN-DNS=null	dV	dV	W	W	dV	dV	dV	dV	R	R	iV	iV	W → R	W → R	W	W	I	I
xi) SCN= null , SAN-DNS=null , SAN-IP=192.168.133.149	dV	dV	W	A	dV	dV	dV	dV	R	R	iV	iV	W	W	W	W → A	I	I
xii) SCN= null , SAN-IP=141.115.26.43	dV	dV	? → W	? → A	dV	dV	dV	dV	? → R	? → R	iV	iV	? → R	? → R	? → R	? → A	I	V
xiii) SCN= dane.irit.fr ,SAN-IP=141.115.26.43	dV	dV	? → A	? → A	dV	dV	dV	dV	? → W	? → R	iV	iV	? → A	? → R	? → A	? → A	V	V
xiv) SCN= null ,SAN-IP=141.115.26.43 SAN-DNS=dane.irit.fr	dV	dV	? → A	? → A	dV	dV	dV	dV	? → A	? → R	iV	iV	? → A	? → R	? → A	? → A	V	V
xv) SCN= null , SAN-DNS =null, SAN-IP=141.115.26.43	dV	dV	? → W	? → A	dV	dV	dV	dV	? → R	? → R	iV	iV	? → R	? → R	? → W	? → A	I	V

*Where: **S1** = sana1.fr or dane.irit.fr, **IP** = 192.168.133.149 or 141.115.26.43, **Na** = not applicable, **dV** = delegated Validation, **iV** = incorrect Validation, **A** = Accept, **W** = Warn, **R** = Refuse, **I** = Invalid certificate w.r.t standard, **V** = valid certificate w.r.t standard, ? → = means that we didn't make this specific test in 2017.*

Figure 4.6. Results of SAN extension tests in *On the Validation of Web X.509 Certificates by TLS interception products*(Source: [2]).

KU and EKU. When in KU are inserted *digitalSignature*, *keyEncipherment* or *keyAgreement* the proper value of the EKU should be *serverAuth*. Keeping in mind that, they firstly discovered what was the cipher suite negotiated between the TLS interception product and the server and then, as well as the SAN extension, they tried different possible combination of value to be inserted in the KU and EKU extensions. The results are summarized in Figure 4.7. In this case, almost all of them accepts anyway the certificate, independently from the cipher suite negotiated and the expected value of the KU and EKU extensions. All antivirus tools but Kaspersky, again, chosen to delegate the validation to the client.

TABLE 5
Key Usage Test

Key Usage	Standard Validity	Avast	Kaspersky	AVG	ESET	Squid	Charles	Mitm	Fiddler
key usage extension = KA No Extended key usage extension	I	dV	A	dV	dV	R → A	iV	A	A
key usage extension = DE No Extended key usage extension	I	dV	A	dV	dV	R	iV	A → R	A
key usage extension = KE No Extended key usage extension	I (AA Products) V(BB Products)	dV	A	dV	dV	A	iV	A → A	A → A
key usage extension = DS No Extended key usage extension	I (BB Products) V(AA Products)	dV	A	dV	dV	A	iV	A → A	A → A
key usage extension = KA key Extended usage extension =SA	I	dV	A	dV	dV	R → A	iV	A	A
key usage extension = DE key Extended usage extension =SA	I	dV	A	dV	dV	R	iV	A → R	A
key usage extension = KE key Extended usage extension =SA	I (AA Products) V(BB Products)	dV	A	dV	dV	A	iV	A → A	A → A
key usage extension = DS key Extended usage extension =SA	I (BB Products) V(AA Products)	dV	A	dV	dV	A	iV	A → A	A → A
No key usage extension key Extended usage extension =CA	I	dV	W	dV	dV	R	iV	A → R	A → W
key usage extension = DS key Extended usage extension =CA	I	dV	W	dV	dV	R	iV	A → R	A → W

Where: ***CA*** *= clientAuthentication,* ***DE*** *= dataEncipherment,* ***DS*** *= digitalSignature,* ***KA*** *= keyAgreement,* ***KE*** *= keyEncipherment,* ***dV*** *= delegated Validation,* ***iV*** *= incorrect Validation,* ***SA*** *= serverAuthentication,* ***I*** *= invalid w.r.t standard,* ***V*** *= valid w.r.t standard,* ***A*** *= Accept,* ***W*** *= Warn,* ***R*** *= Refuse,* ***I (AA Products)*** *= means the certificate is invalid for AA products,* ***V(AA Products)*** *= means the certificate is valid for AA products.*

Figure 4.7. Results of Key Usage and Extended Key Usage extensions tests in *On the Validation of Web X.509 Certificates by TLS interception products*(Source: [2]).

Lastly they tested the capabilities of the interception products to check the status of a certificate. They tested their support for CRL checks OCSP, OCSP Stapling and OCSP Must-Staple. The support for the revocation check is rare in those products. They have also written a Java program running on one of their servers that connected to the Alexa Top-1-Million domains and verified the support of OCSP stapling and OCSP Must-Staple on real domains. They repeated the measurements twice, in 2018 and then in 2019. They discovered that the support of OCSP stapling passed from the 19% in 2018 to 27% in 2019. Meanwhile, OCSP Must-Staple passed from the 0.04% to 0%.

Another interesting work is *To intercept or not to Intercept: Analyzing TLS Interception in Network Appliances*[3] by Louis Waked et al. in 2018. They tested different TLS interception products listed in Figure 4.8.

They mainly focused on testing their certificate validation process, including not just the leaf

Table 1: List of the tested appliances

Appliance	Company	Version
Untangle NG Firewall	Untangle	13.0
pfSense	NetGate	2.3.4
WebTitan Gateway	TitanHQ	5.15 build 794
Microsoft TMG	Microsoft	2010 (SP1, SP1 Update, SP2 rollup updates 1 to 5)
UserGate Web Filter	Entensys	4.4.3320601
Cisco Ironport WSA	Cisco	Async OS version 10.5.1 build 270

Figure 4.8. TLS interception tools tested in *to Intercept: Analyzing TLS Interception in Network Appliances* (Source [3]).

certificate but the whole chain sent by the server. The server in question was an Apache one running on an Ubuntu VM. The results of their tests are collected in Figure 4.9, showing, also with those different products, an inconsistent behavior.

WebTitan Gateway and UserGate Web Filter did not respect almost at all the validation process standards. Still, issues were caused by the revocation status check in most of them.

Table 3: Results for certificate validation, part I. ✓ means a faulty certificate is accepted and converted to a valid certificate by the TLS proxy; → means a faulty certificate is passed to the client as-is but not blocked; and blank means certificate blocked.

	Self Signed	Signature Mismatch	Fake Geo-Trust	Wrong CN	Unkn-own Issuer	Non-CA Interm-ediate	X509v1 Interm-ediate	Invalid pathLen-Constraint	Bad Name Constraint Intermediate	Unknown Critical X509v3 Extension	Malformed Extension Values
Untangle				→					✓		✓
pfSense									✓		→
WebTitan Gateway	✓	✓	✓	→	✓	✓	✓	✓	✓	✓	→
Microsoft TMG									✓		✓
UserGate Web Filter	✓	✓	✓	→	✓	✓	✓	✓	✓	✓	→
Cisco Ironport WSA									✓		→

Table 4: Results for certificate validation, part II. For legend, see Table 3; N/A means not tested as the appliance disallows adding the corresponding faulty CA certificate to its trusted store.

	Revoked	Expired Leaf	Expired Interm-ediate	Expired Root	Not Yet Valid Leaf	Not Yet Valid Interm-ediate	Not Yet Valid Root	Leaf keyUsage w/out Key Enciph-erment	Root keyUsage w/out KeyCert-Sign	Leaf extKey-Usage w/ clientAuth	Root extKey-Usage w/ Code Signing
Untangle	✓			✓			✓		✓		✓
pfSense	✓										✓
WebTitan Gateway	✓	→	✓	✓	→	✓	✓	→	✓	→	✓
Microsoft TMG								✓			✓
UserGate Web Filter	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Cisco Ironport WSA		→		N/A			N/A		N/A		N/A

Figure 4.9. Test results in *to Intercept: Analyzing TLS Interception in Network Appliances* (Source [3]).

Chapter 5

Testbed Configuration and tests generation

5.1 Methodology

There are two test phases. In the first one the certificate validation process of web-browsers and TLS interception products is tested while the second one is a statistical measurement of the real world usage of CT, OCSP Must-Staple and Stapling adoption and TLS versions negotiated.

Since the PKI standards state how to identify an invalid certificate but do not state anything strict about what to do with that certificate, it is up to the programmer to choose what to do. That is the reason why in this work have been selected a multitude of TLS clients as test subjects. All the server applications that will host a misconfigured certificate are listed in Table 5.1, the web-browsers chosen are described in Table 5.2 and he TLS interception products in Table 5.3.

The machines that compose the testbed and their interaction during tests can be found in Section 5.1.4.

The certificate profiles used for testing are in section 5.2.1. Those certificates are generated through a set of Python scripts described in Section 5.2.2. They are in charge, also, for the automatic creation of the server configuration files to be loaded by the server products. More details about this test generation scripts can be found in Appendix B.1

The second phase of test aims to collect statistical data (Section 5.2.3). Another script is in charge to connect to each of the domains contained in the Majestic Top-1-Million domains[12]. The objective of this analysis is to obtain a chart of the usage distribution of TLS versions, how many of those domains support OCSP Stapling and OCSP Must-Staple and lastly study the actual CT usage.

This last objective can be tested only in this way since it requires the certificates to be in a public log server. This kind of operation is incompatible with TLS interception products that generates certificates on-the-fly and with this thesis testing environment which rely on fake root CA certificates.

5.1.1 Servers

All the servers that try to authenticate with the anomalous certificates are installed on an Ubuntu 20.04 LTS Virtual Machine (VM) and are listed in Table 5.1. They are Apache 2.4.41, Nginx 1.18 and Lighttpd 1.4.66. The objective is to notice any difference in the TLS connection using different server products. Instructions on how to install and run them are in Appendix A.2.

OS	Server
Ubuntu 20.04.4 LTS	Apache 2.4.41 Nginx 1.18 Lighttpd 1.4.66

Table 5.1. Server used to advertise test certificates.

5.1.2 Clients

All the tests are run against the major desktop and mobile web-browsers, installed on all principal Operating Systems (OSs), to have the highest possible coverage of TLS client behavior facing a wrong certificate. Table 5.2 contains the web-browsers chosen as targets.

Device Type	OS	TLS Clients
Desktop	Ubuntu 20.04.4 LTS	Mozilla Firefox v105 Google Chrome v106 Opera v91
	macOS Monterey 12.6	Mozilla Firefox v105 Google Chrome v106 Safari v15.5 Opera v91 Microsoft Edge v106
	Microsoft Windows 10	Mozilla Firefox v105 Google Chrome v106 Opera v91 Microsoft Edge v106
Mobile	iOS 16	Safari v15.5 Google Chrome v106 Mozilla Firefox v105 Opera v3.3.7 Microsoft Edge v105
	Android 11	Google Chrome v106 Mozilla Firefox v105 Microsoft Edge v105

Table 5.2. Web-browsers target of test.

The web-browsers in question are Safari, Google Chrome, Mozilla Firefox, Microsoft Edge and Opera. Each of the OSs will run the subset of these TLS clients available for them. On Ubuntu 20.04 LTS are installed Mozilla Firefox, Google Chrome and Opera; on Windows 10 are installed Mozilla Firefox, Google Chrome, Opera and Microsoft Edge; on macOS 12.6 are

installed Mozilla Firefox, Google Chrome, Opera, Microsoft Edge and Safari; on iOS 16 device are installed the mobile versions of Mozilla Firefox, Google Chrome, Opera, Microsoft Edge and Safari; on Android 11 are installed the mobile versions of Mozilla Firefox, Google Chrome and Microsoft Edge. Although there is a mobile version of Opera running on Android, it does not allow to use local certificates. So it is pointless to use it for this work's tests. In fact, tests generate fake certificate chains, so to validate them, root CA certificates are inserted into the trusted authorities certificate stores of the target devices.

5.1.3 TLS interception products

The TLS interception products to be tested are listed in Table 5.3.

OS	TLS product
Ubuntu 20.04.4 LTS	Mitmproxy v8.1.1 Squid v5.5
Microsoft Windows 10	Mitmproxy v8.1.1 Squid v4.14 Kaspersky Total Security ESET Smart Security
macOS Monterey 12.6	Mitmproxy v8.1.1 Squid v4.17

Table 5.3. TLS interception products target of test.

Interception products are divided into two categories: proxies and antivirus tools. *Mitmproxy* and *Squid* fall in the former while *Kaspersky Total Security* and *ESET Smart Security* in the latter.
Proxies are installed on all the OSs and follow the three-machines scenario described in Figure 3.2. On the contrary, the remaining two products are installed directly on the client machines, Windows 10 only.[1]

To make the interceptor able to validate for the client the certificate coming from the server, it is necessary to install the root certificate of the product in the client's trusted authorities certificate store. In this way any certificate coming from the TLS interceptor will be automatically accepted by any TLS client installed on client machine. Again, since the certificate chains generated in this thesis are not real ones there is the need to add them to a real CA bundle and configure the interceptor to use them for certificate validation. In case of antivirus tools, it is enough to install them on the interceptor machine's trusted store.

All the commands and operations that have been executed in order to install those softwares are described in Appendix A.1.

5.1.4 Testbed

Initially all the tests want to analyze the behavior of the different web-browsers running on the different devices. The scenarios which describe this situation are depicted in Figures 5.1, 5.2,

[1]Other antivirus tools have been considered: AVG antivirus and AVAST free antivirus. They, unfortunately, are not allowed anymore by the web-browsers but Microsoft Internet Explorer to intercept the TLS traffic. So they are not used in this work for testing purposes.

5.3, 5.4, 5.5. In this phase, there is a physical desktop machine running Ubuntu 20.04 LTS, one running Windows 10, one running macOS 12.6; in addition an iOS 16 mobile device and an Android 11 one are used to test mobile web-browsers results. On the Ubuntu machine a Virtual Machine (VM) running the same OS is used to host an Apache, Nginx and Lighttpd server. In case of public IP address tests a further Ubuntu machine, exposed to the external network, is used to host the server applications.

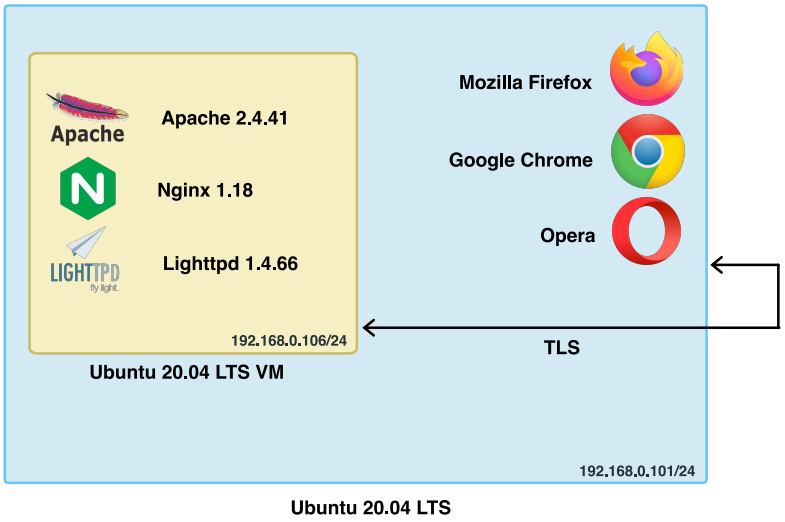

Figure 5.1. Test run between the different web-browsers installed on Ubuntu machine and the different servers installed on the Ubuntu server virtual machine.

Subsequently the same tests are launched in more complex scenarios. In those cases the interception products come to play. Figure 5.6, 5.7 and 5.8 show the tests passing through Mitmproxy. As represented in the figures, there is an additional VM for Ubuntu OS and Windows 10 OS. Each client is configured to connect through the IP address of the machine which runs the proxy listening on port 8080. The macOS case is slightly different. The proxy is installed directly on the physical machine that acts as client too. In this case, in order to connect through the proxy, the macOS machine uses the loopback address[2] and the same port.

Figures 5.9, 5.10 and 5.11 show a similar scenario to the Mitmproxy one. The only differences are the proxy used (Squid) and the port used. In this case is the 3128.

The last two scenarios describe the cases in which the interception product is an antivirus. Figure 5.12 show the tests run directly on the Windows 10 physical machine relying on Kaspersky Total Security. The second one (Figure 5.13) uses two VMs: the server one and a Windows 10 one with ESET Smart Security.

[2] 127.0.0.1

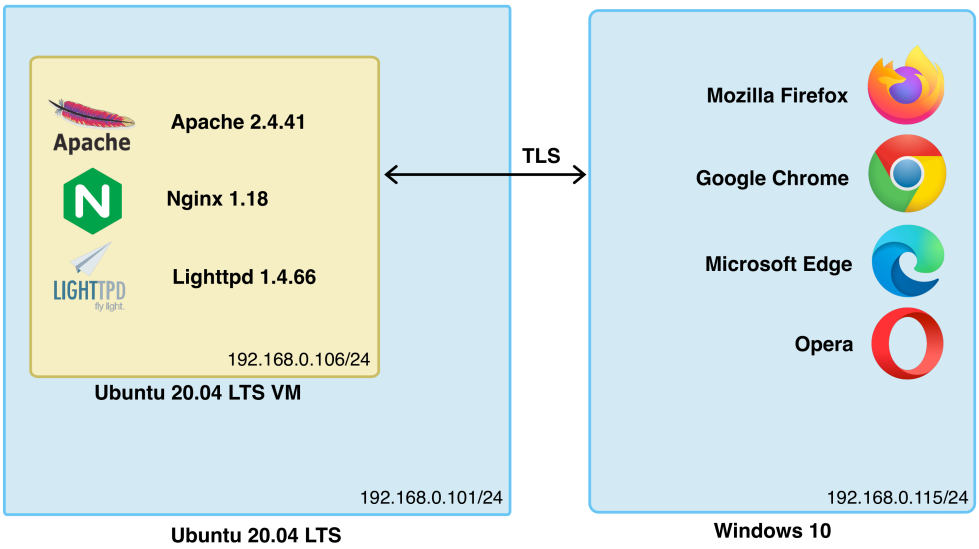

Figure 5.2. Test run between the different web-browsers installed on Windows 10 machine and the different servers installed on the Ubuntu server virtual machine.

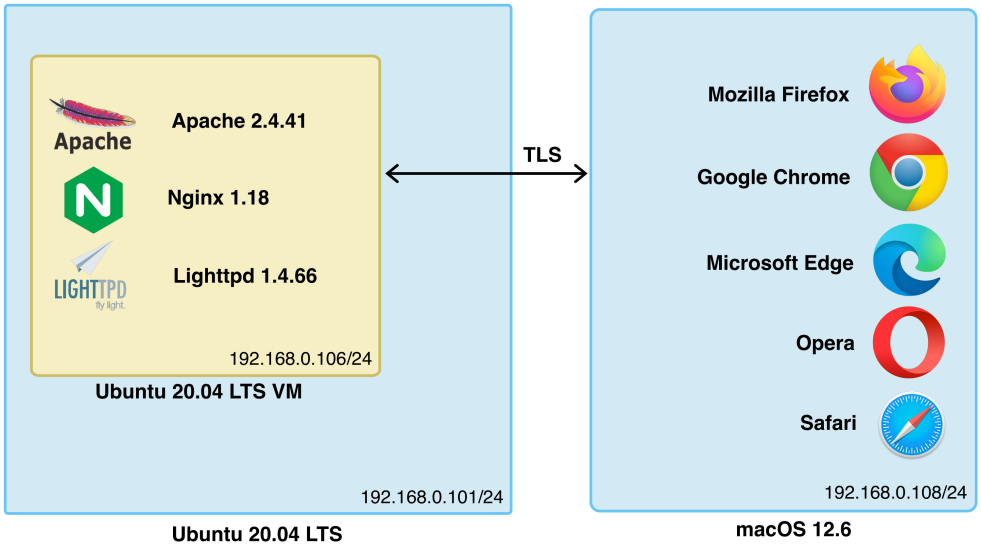

Figure 5.3. Test run between the different web-browsers installed on macOS machine and the different servers installed on the Ubuntu server virtual machine.

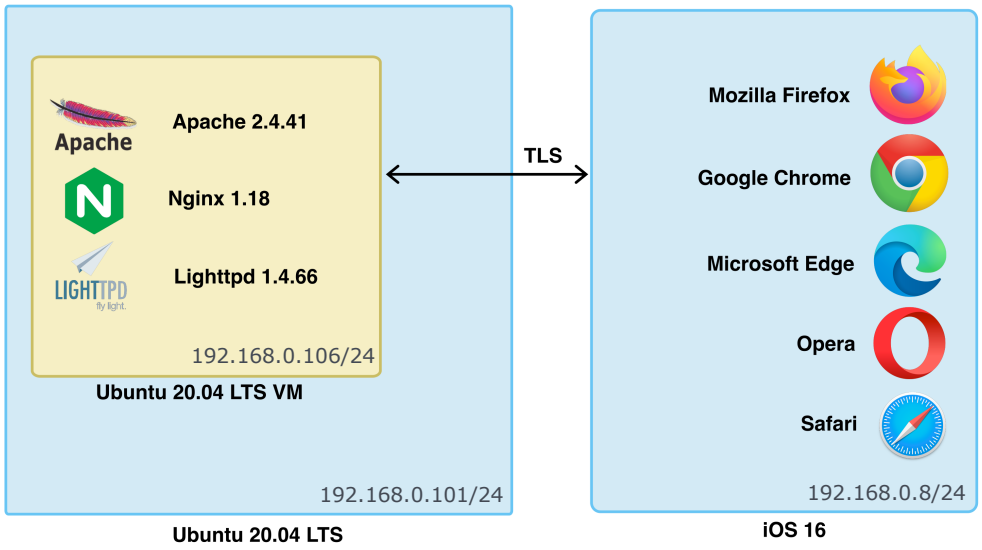

Figure 5.4. Test run between the different web-browsers installed on iOS mobile device and the different servers installed on the Ubuntu server virtual machine.

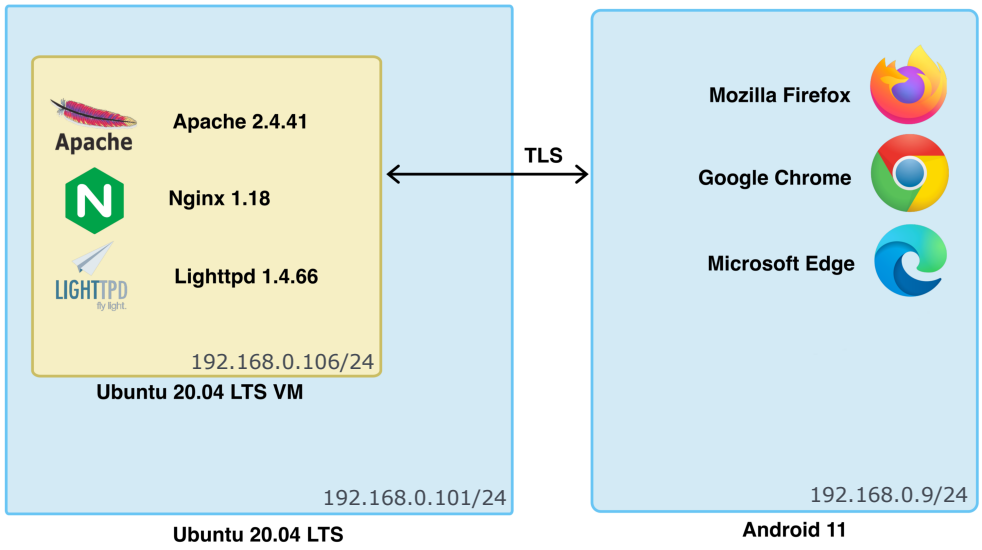

Figure 5.5. Test run between the different web-browsers installed on Android mobile device and the different servers installed on the Ubuntu server virtual machine.

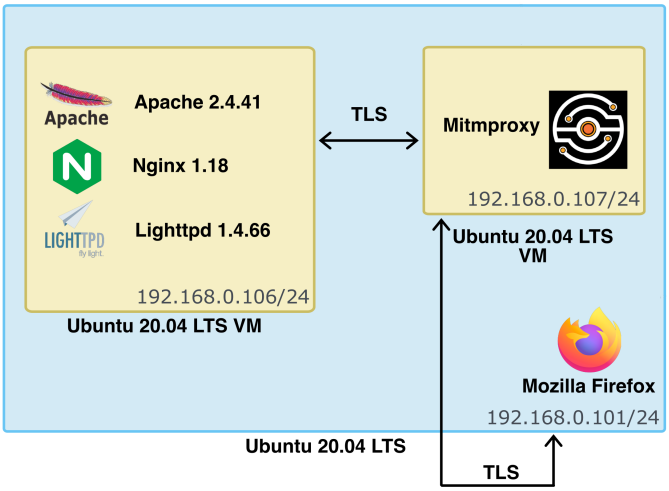

Figure 5.6. Tests using as client Mozilla Firefox on Ubuntu machine connecting to the different servers installed on Ubuntu server virtual machine passing through Mitmproxy installed on an other Ubuntu virtual machine.

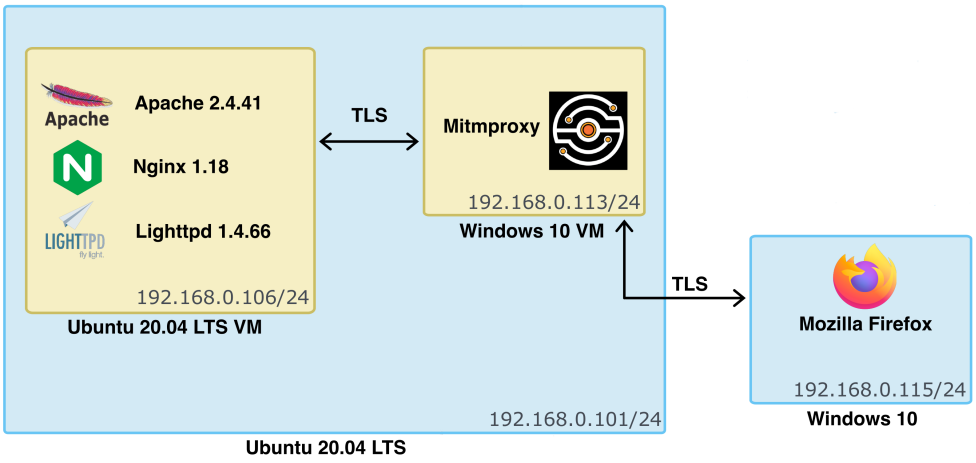

Figure 5.7. Tests using as client Mozilla Firefox on Windows 10 machine connecting to the different servers installed on Ubuntu servers virtual machine passing through Mitmproxy installed on a Windows 10 virtual machine.

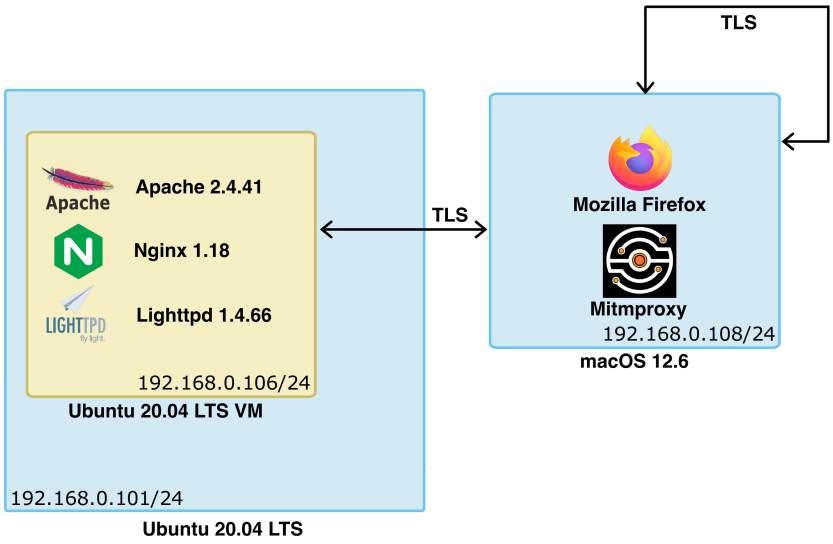

Figure 5.8. Tests using as client Mozilla Firefox on macOS machine connecting to the different servers installed on Ubuntu server virtual machine passing through Mitmproxy installed the macOS physical machine.

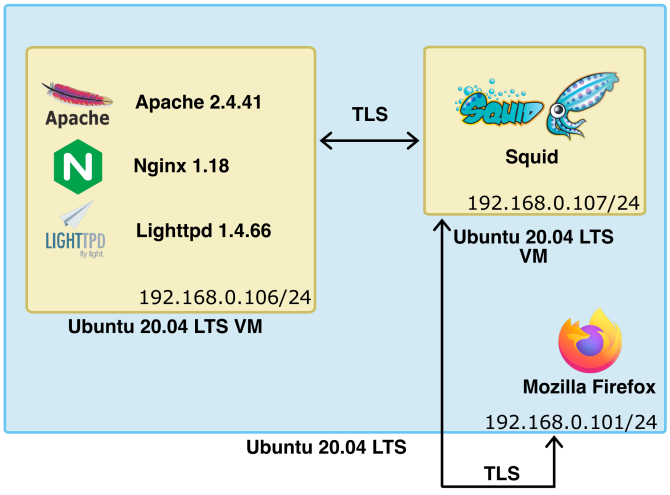

Figure 5.9. Tests using as client Mozilla Firefox on Ubuntu machine connecting to the different servers installed on Ubuntu server virtual machine passing through Squid installed on an other Ubuntu virtual machine.

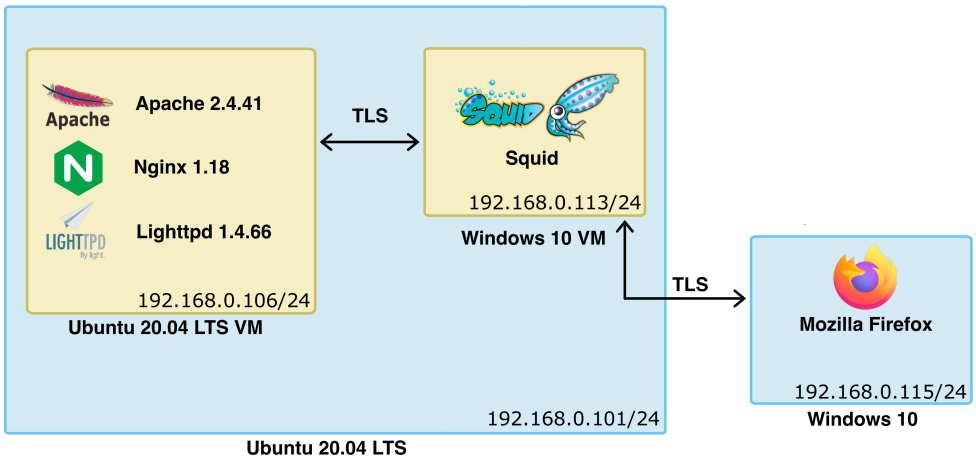

Figure 5.10. Tests using as client Mozilla Firefox on Windows 10 machine connecting to the different servers installed on Ubuntu server virtual machine passing through Squid installed on a Windows 10 virtual machine.

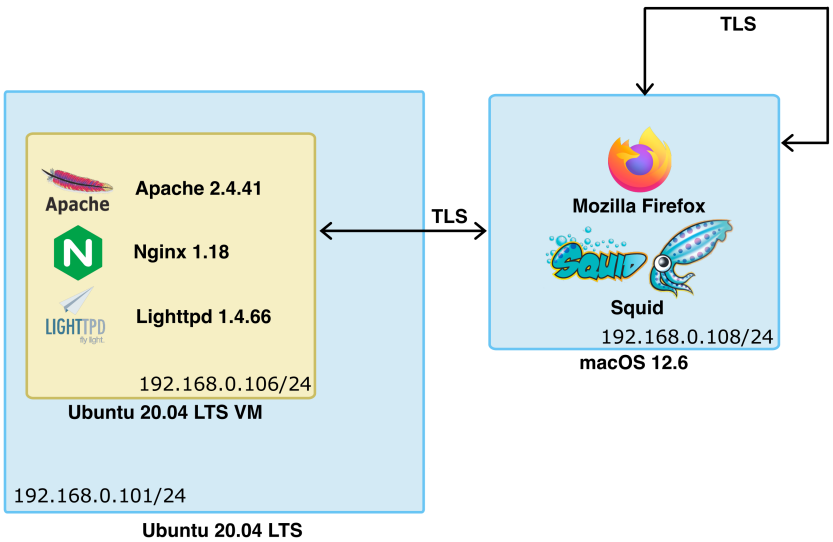

Figure 5.11. Tests using as client Mozilla Firefox on macOS machine connecting to the different servers installed on Ubuntu server virtual machine passing through Squid installed the macOS physical machine.

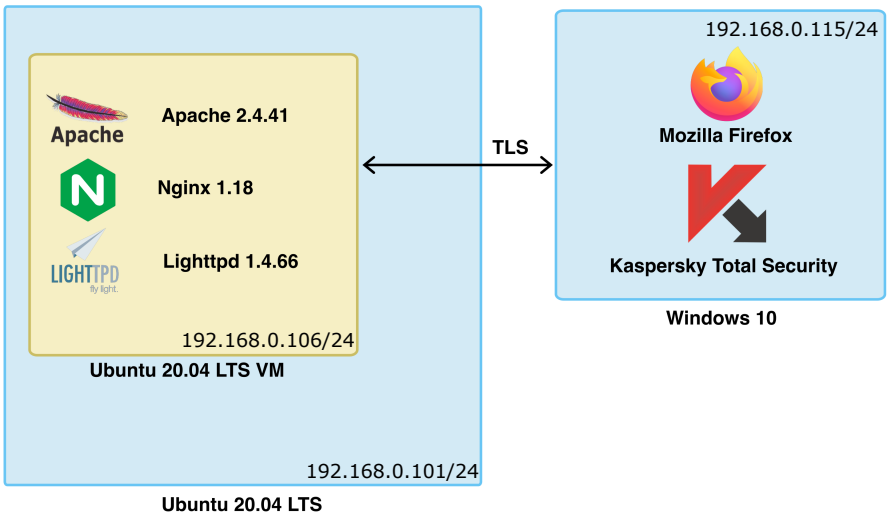

Figure 5.12. Windows 10 machine using as client Mozilla Firefox connecting to the different servers installed on Ubuntu server virtual machine using Kaspersky Total Security installed on it.

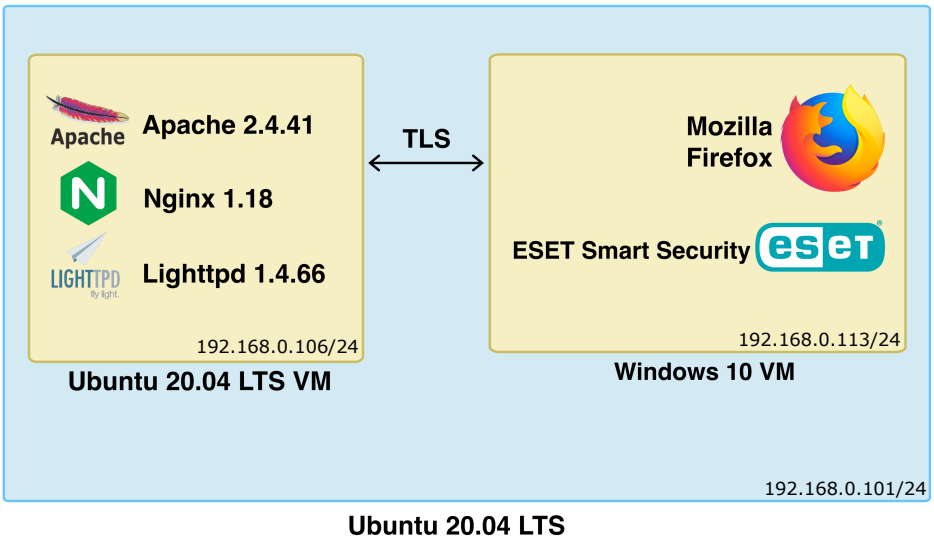

Figure 5.13. Windows 10 virtual machine using as client Mozilla Firefox connecting to the different servers installed on Ubuntu server virtual machine using ESET Smart Security installed on it.

5.2 Tests

TLS handshake phase is used to establish a secure communication channel between client and server. The server has to send to the client its X.509 certificate (in the form of a logical chain of certificates) to authenticate himself. The client, a.k.a. the Relying Party (RP), is in charge to validate it. The RP starts from the root and perform the following check for all the certificates in the chain: first, verifies that the certificate is correct from the cryptographic point of view, so verifies its digital signature; then it verifies the validity period of the certificate; next, checks the revocation status of the certificate; subsequently, checks if the issuer field correspond to the subject field in the previous certificate in the chain (those value are the same in case of root certificate); lastly the constraints introduced by the X.509 version 3 extension are verified. If every of those operations succeed that means the leaf certificate (the one which identifies the web-server) is valid [41].

The purpose of the test profiles chosen is to verify the state-of-the-art of this validation process, not only by the web-browsers but also by the TLS interception products which should guarantee an equivalent level of security since they validate certificates in behalf of the clients.

5.2.1 Test description

The first tests aim to see what happens at client-side (both web-browsers and TLS interception products) if the received server certificate does not fully respect the X.509 format[13]. Table 5.4 collects the test profiles run during this phase.

1. This test simply signs a leaf certificate with a randomly generated key instead of the intermediate CA's private key.
2. The aim of this test is to swap the values of *notBefore* and *notAfter* attributes of the certificate, obtaining an invalid validity period.
3. This test generates a leaf certificate without the *notBefore* value.
4. This test generates a leaf certifcate without the *notAfter* value.
5. The purpose of this test is to go against what is stated in the CA/Browser Forum[7]. The forum states that certificates issued on or after 1 September 2020 should not have a Validity Period greater than 397 days and must not have a Validity Period greater than 398 days. In this case the leaf certificate has a validity period of 50 years.
6. This test inserts as subject Common Name field a `\0x00` (NULL) value.
7. This test inserts as subject Common Name *www.mydifferentwebsite.com* instead of *www.mywebsite.com*.
8. This test inserts as subject Common Name field a `\0x09` (tab escape) value.
9. This test inserts as subject Common Name field a `\0x08` (backspace escape) value.
10. The Organization Unit Name, whose usage is deprecated according the CA/Browser Forum, contains 200 `A` characters. The objective is to cause a buffer overflow at client-side.
11. From what is stated in the RFC-5280 [6], the maximum length of the serial number bit string in a certificate is 20 octets. In this case the leaf certificate has been generated with a 30 octets serial number.

12. This test inserts a completely invented X.509v3 extension marked as critical in the leaf certificate. That kind of certificate should be rejected by the RP.

Test no.	Profile	Description
1	WrongKey	Leaf certificate signed using the wrong key.
2	SwapStartEnd	Leaf certificate with start and end date swapped.
3	MissingStart	No start date in the leaf certificate.
4	MissingEnd	No end date in the leaf certificate.
5	LongEnd	Leaf certificate with very distant end date in the future (50 years).
6	NullCN	NULL character in subject CN field of the leaf certificate.
7	FooCN	Different CN from what is expected in subject CN field of the leaf certificate.
8	TabCN	Tab escape character in subject CN field of the leaf certificate.
9	BackspaceCN	Backspace escape character in subject CN field of the leaf certificate.
10	LongOU	Long character string to overflow the string buffer used for OU field.
11	LongRandomSerial	Serial number longer than expected.
12	UnknownCriticalExt	Leaf certificate with a critical X.509v3 unknown extension

Table 5.4. Test profiles for X.509 format anomalies.

The subsequent set of tests wants to analyze the effects of a deliberately erroneous usage Subject Alternative Names (SAN) extension in combination with the subject Common Name field of the certificate. The CA/Browser Forum states that this extension must be present and must contain at least one entry that could be a DNS name or a public IP address. The usage of a private address in it is prohibited. Moreover, the usage of Common Name (CN) field is discouraged. If it is present, then must contain exactly one of the entries in the SAN extension.

13. This test inserts in subject CN field *www.mywebsite.com* of the leaf certificate, while in the SAN extension of the same certificate *www.mydifferentwebsite.com*.

14. This test insert `\0x00` character in the subject CN field of the leaf certificate and *www.mywebsite.com* in SAN of the same certificate.

15. Both the subject CN field and the SAN extension of the leaf certificate contain `0x00` character.

16. In the leaf certificate subject CN is inserted *www.mywebsite.com*. There is no SAN extension in that certificate.

17. In the leaf certificate subject CN field is inserted `0x00`, while the SAN extension of the same certificate contains two entries, *www.mywebsite.com* and *www.mydifferentwebsite.com*.

18. The subject CN field of the leaf certificate contains `0x00` character and there is no SAN extension in the same certificate.

19. The subject CN field of the leaf certificate contains `0x00` character and the SAN extension in the same certificate contains *192.168.0.106* private IP address.

20. The subject CN field of the leaf certificate contains *www.mywebsite.com* and the SAN extension in the same certificate contains *192.168.0.106* private IP address.

21. In the leaf certificate subject CN field is inserted `0x00`, while the SAN extension of the same certificate contains two entries, *www.mywebsite.com* and *192.168.0.106* private IP address.

22. The subject CN field of the leaf certificate contains `0x00` character and the SAN extension in the same certificate contains *130.192.1.42* public IP address.

23. The subject CN field of the leaf certificate contains *service-test-eid4u.polito.it* and the SAN extension in the same certificate contains *130.192.1.42* public IP address.

24. In the leaf certificate subject CN field is inserted `0x00`, while the SAN extension of the same certificate contains two entries, *service-test-eid4u.polito.it* and *130.192.1.42* public IP address.

25. In the leaf certificate subject CN field is inserted `0x00`, while the SAN extension of the same certificate contains two entries, `0x00` character and *130.192.1.42* public IP address.

Those set of test profiles are described in Table 5.5.

Test no.	Profile	Description
13	CN-SAN-diff-value	Subject CN field and SAN with different value.
14	nullCN-SAN_DNS	NULL value in subject CN field and DNS name in SAN.
15	nullCN-nullSAN_DNS	NULL value in subject CN and NULL value in SAN.
16	CN-noSAN_DNS	Subject CN specified and no SAN.
17	nullCN-2SAN_DNS	NULL value in subject CN and two entries for SAN.
18	nullCN-noSAN	NULL value in subject CN and no SAN.
19	nullCN-SAN_privIP	NULL value in subject CN and private IP address in SAN.
20	CN-SAN-diff-value-IP	Subject CN field with DNS name and SAN with private IP address.
21	nullCN-SAN_DNS-SAN_IP	NULL value in subject CN and DNS name plus private IP address in SAN.
22	nullCN-SAN_pubIP	NULL value in subject CN and public IP address in SAN.
23	CN-SAN_pubIP	DNS name in subject CN and public IP address in SAN.
24	nullCN-SAN_DNS-and-SAN_pubIP	NULL value in subject CN, SAN with DNS name and public IP address.
25	nullCN-nullSAN-and-SAN_pubIP	NULL value in subject CN, SAN with NULL value and public IP address.

Table 5.5. Test profiles for SAN extension anomalies.

Tests from 22 to 25 in Table 5.6 are meant to test what the RP decides to do in case he can not properly get revocation information from the certificate.

26. In this test the leaf certificate generated does not contain the URI of the intermediate CA CRL in the *CRL Distribution Point* (CDP) extension and the OCSP URI in the *Authority Information Access* (AIA) extension.

27. The leaf certificate in this test contains properly configured the CDP and AIA extensions to get the revocation info but the URI of the CRL is unreachable.

28. The leaf certificate in this test contains properly configured the CDP and AIA extensions to get the revocation info but the URI of the OCSP server is unreachable.

29. The leaf certificate in this test contains properly configured the CDP and AIA extensions. The certificate is revoked. The server is configured to send stapled revocation status of the certificate.

30. The leaf certificate in this test contains properly configured the CDP and AIA extensions. The certificate is endowed of *TLS Feature* extension, a.k.a. Must-Staple extension. The server is configured to do not send the stapled status of the certificate.

Test no.	Profile	Description
26	NoRevocationInfo	No CRL and OCSP server indicated in the leaf certificate.
27	NoCRL	CRL not obtained but OCSP server up and running
28	DownOCSP	CRL reachable but OCSP server down
29	OCSPStapleRevoked	OCSP stapled response of revoked certificate
30	OCSPMustStapleNoResponse	Must-Staple extension set but no stapled response obtained

Table 5.6. Test profiles for revocation check of certificates.

The set of tests in Table 5.7 analyzes the behavior in case of anomalous certificate chains retrieved from web-server during the TLS handshake phase.

31. This test send to the client a self-signed leaf certificate that should be rejected.

32. In this test a new root CA certificate has been generated but it is not installed at client-side trusted authorities store.

33. This test intermediate CA certificate has the *Basic Constraints* extension set to `CA:False`. It should not be used to sign other certificates.

34. This test has two intermediate CA certificates. The first has the *Basic Constraints* extension set to `CA:True` and the maximum path length set to 0. That means that that specific CA cert can issue only leaf certificates.

35. This test contains a leaf certificate issued by a revoked intermediate CA certificate.

36. In this test the validity period of the leaf certificate is postponed by a month.

37. In this test the validity period of the intermediate CA certificate is postponed by a month.

38. In this test the validity period of the root CA certificate is postponed by a month.

39. The whole chain of certificates, containing also the root one, is sent to the client for validation. That root certificate is installed at client-side though.

Test no.	Profile	Description
31	SelfSignedCertificate	Self-signed leaf certificate
32	UnknownIssuer	Root CA certificate unknown
33	NonCAIntermediate	Intermediate CA certificate with Basic Constraint CA option set to False
34	InvalidPathLenConstraint	Intermediate CA issues certificate for second intermediate CA but path length set to 0
35	RevokedIssuer	Intermediate CA has been revoked
36	NotYetValidLeaf	Not yet valid leaf certificate
37	NotYetValidInter	Not yet valid intermediate CA certificate
38	NotYetValidRoot	Not yet valid root CA certificate
39	ChainWithRoot	Complete chain with root CA certificate

Table 5.7. Test profiles for misconfiguration in certificate chain.

The *Key Usage* (KU) and the *Extended Key Usage* (EKU) extensions express the purpose of the key associated to the certificate. The key can be used just for one of the purposes expressed in the KU and EKU extensions. The RP must handle those extensions separately but their values have to be compatible. A better insight of the possible value has already been produced in Section 2.1. Tests that analyze wrong configuration of those extensions are in Table 5.8.

40. The leaf certificate has Key Agreement as KU extension and no EKU set.

41. The leaf certificate has Data Encipherment as KU extension and no EKU set.

42. The leaf certificate has Key Encipherment as KU extension and no EKU set.

43. The leaf certificate has Digital Signature as KU extension and no EKU set.

44. The leaf certificate has Key Agreement as KU extension and EKU extension set to Server Authentication.

45. The leaf certificate has Data Encipherment as KU extension and EKU extension set to Server Authentication.

46. The leaf certificate has Key Encipherment as KU extension and EKU extension set to Server Authentication.

47. The leaf certificate has Digital Signature as KU extension and EKU extension set to Server Authentication.

48. The leaf certificate has Digital Signature as KU extension and EKU extension set to Client Authentication.

49. The leaf certificate has no KU extension set and EKU extension set to Client Authentication.

Test no.	Profile	Description
40	KU_KA_noEKU	KU in leaf certificate set to Key Agreement, no EKU extension.
41	KU_DE_noEKU	KU in leaf certificate set to Data Encipherment, no EKU extension.
42	KU_KE_noEKU	KU in leaf certificate set to Key Encipherment, no EKU extension.
43	KU_DS_noEKU	KU in leaf certificate set to Digital Signature, no EKU extension.
44	KU_KA_EKU_SA	KU in leaf certificate set to Key Agreement, EKU extension set to Server Authentication.
45	KU_DE_EKU_SA	KU in leaf certificate set to Data Encipherment, EKU extension set to Server Authentication.
46	KU_KE_EKU_SA	KU in leaf certificate set to Key Encipherment, EKU extension set to Server Authentication.
47	KU_DS_EKU_SA	KU in leaf certificate set to Digital Signature, EKU extension set to Server Authentication.
48	KU_DS_EKU_CA	KU in leaf certificate set to Digital Signature, EKU extension set to Client Authentication.
49	noKU_EKU_CA	No KU in leaf certificate, EKU extension set to Client Authentication.

Table 5.8. Test profiles for malformed Key Usage and Extended Key Usage extensions in certificates.

5.2.2 Certificates and server configuration generation

To automate and simplify the generation of all the certificates together with server configurations for each of them, a set of Python scripts have been produced. They combine the usage of two Python libraries: PyOpenSSL version 22 [42] and cryptography version 39 [43].

The whole software has been realized to be modular and extendable. To generate tests, a configuration YAML file is used. The format to identify a test profile is:

```
test_<#>:
    profile: <test profile name>
```

The files are organized as follow:

```
.
|__config/
| |__configuration.yml
|
|__profile_generation.py
|__profiles.py
|__server_config.py
|__test_generation.py
|__run.sh
```

The algorithm can be summarized in 4 steps:

1. Read the whole configuration file and saves it into a list.

2. Creates a *tests* folder. Pick a test from the list and creates a so structured directory tree (Figure 5.14): the root folder is named after the `test_<#>` field of the current test; a first sub-folder *certificates* and its three sub-folders, *rootCA*, *interCA* and *leafCRT*; a sub-folder for each server configuration, *apache*, *nginx* and *lighttpd*.

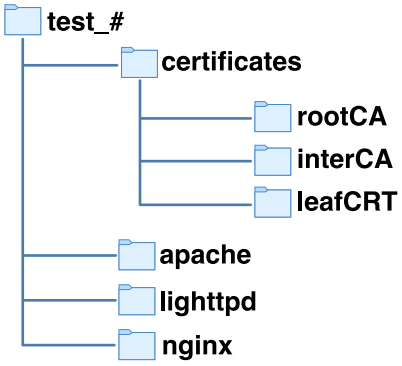

Figure 5.14. Directory tree generated per each test.

3. According to the profile, it calls a second script to create the proper certificate chain. Saves each certificate and its private key in the corresponding folder. At this step concatenates the leaf certificate and the intermediate CA certificate into the *chain.crt* file (used for server configuration), concatenate the CA certificates in the *ca-bundle.pem* file and saves them in the *certificates* folder.

4. Create a configuration per server and save them properly.

Step 3 has also another effect. Since the aim of this thesis is to study the behavior of the clients in as more realistic as possible scenario, each CA, root or intermediate, is endowed with an OCSP server and a its own server to make the client able to download the corresponding CRL or certificate and use OCSP protocol through the CRL Distribution Point and Authority Information Access extensions.

In order to run the test generation, it is sufficient to call run *run.sh* script as superuser after having installed the required dependencies (see Appendix B.1).

In order to make the OCSP server work, the script generates a *certs* folder. Within that, a proper CA directory is created with a *demoCA* and an *OCSP* sub-folder. In the latter are stored the OCSP server private key and its certificate, whose Extended Key Usage extension is set to *OSCPSigning*. The *demoCA* folder is needed for compatibility with OpenSSL [44]. It contains a *newcerts* folder in which are stored all the certificates issued by the CA named with their hexadecimal serial number in PEM format; an *index.txt* file, keeping trace of the status of the issued certificates [45]; a file *serial* with the hexadecimal value of the serial number of the CA certificate; the CA certificate in PEM format and its CRL file. To run the OCSP server, the OpenSSL command as in Figure 5.15 is run from terminal for each CA.

```
ubuntu-server@ubuntuserver-VirtualBox:~/test_script$ openssl ocsp -index certs/rootCA/demoCA/index.txt -rsigner certs/
rootCA/OCSP/ocsp.crt -rkey certs/rootCA/OCSP/ocspSigning.key -port 8081 -CA certs/rootCA/demoCA/cacert.pem -out certs/
rootCA/OCSP/log.txt -text
ocsp: waiting for OCSP client connections...
```

Figure 5.15. OCSP server of the root CA running on port 8081.

Two additional simple Python HTTP servers are run on the same IP address, representing the CA servers (Figure 5.16).

```
ubuntu-server@ubuntuserver-VirtualBox:~/test_script$ python3 -m http.server 8080 --directory
 ./certs/rootCA/demoCA/
Serving HTTP on 0.0.0.0 port 8080 (http://0.0.0.0:8080/) ...
```

Figure 5.16. Root CA server running on port 8080.

Finally, one at the time, all the three servers are run with the generated configuration per test, ready to collect results (see Appendix A.2).

5.2.3 Top-1-Million domains analysis

The second phase of this thesis is more oriented to a statistical measure of the actual state of the global top 1 million domains. The domain ranking chosen for this study is the Majestic Million[3][12]. A further Python script has been produced. This script, take as input the list of domains it has to analyze. For each of them, it will perform two different TLS connections: the former through the command `openssl s_client -connect [domain]:443 -status`; the latter using a secure socket implemented in Python.

The first connection is needed to retrieve information about the OCSP Stapling support. In this case, from the result of this command, the PEM encoding of the server certificates is also saved. From the certificate are extracted the SCTs embedded in it and is checked if it is present the *TLS Feature* extension (which expresses the OCSP Must-Staple support of that certificate). For what concerns the SCTs, their log IDs are compared to the ones contained in a JSON file created ad-hoc. This JSON file contains the actual recognized log servers IDs by the Chromium Project[26], grouped by log provider. The actual providers are just 6: Google, Digicert[29], Let's Encrypt[30], Sectigo[31], TrustAsia[28] and Cloudflare[27]. In this way, it is possible to count how many SCTs are embedded in the real top 1 million domains and which log server they refer to.

The second connection is used to retrieve information about the TLS handshake. In details, the TLS version negotiated.

All the information collected is used to produce a CSV output file which contains the results in the format:

```
[rank no.],[SCT no.],[provider1: ... : providerN],[log1: ... :logN],[TLS
    version],[Must-Staple support],[Stapling support]
```

[3]The Top-1-Million ranking used for analysis is the one from Majestic because the Alexa Top-1-Million ranking went out of support. This means that its dataset is not updated at the time this thesis is produced.

If the domain is unreachable, due to a network problem or to the impossibility to perform the domain name translation, an entry with all the CSV fields set to NULL but the rank number will be inserted in the output file.

All the data collected by this script is used by it to automatically generate charts that express the usage of log servers, extract the percentage of support of OCSP Stapling and OCSP Must-Staple. Lastly what are the actual TLS version negotiated globally.

The script and the related files are attached to this thesis and are organized as follow:

```
.
|__./logs.json
|__./majestic_million.txt
|__./tls_client.py
```

Appendix B.2 contains a manual to use this script. It can be used, also, to analyze just part of a set of domains or only plot an already existing set of results.

Chapter 6

Results

6.1 Desktop web-browser results

There are 4 possible results of a TLS handshake:

- the certificate is valid, represented with ✓ in test results;
- the client opts for an Hard-Fail, HF in results, closes the connection;
- the client opts for a Soft-Fail, most popular, indicated with SF in results, shows a warning message to the user but does not close the connection;
- the client waits until timeout of connection in case of failure.

Table 6.1 lists the choices of web-browsers installed on Desktop machines. The first server tested is Apache. As shown from the table, in most of the cases of failure the web-browser chooses to go with a SF approach. *MissingStart* and *MissingEnd* tests have not been run since the server refused to start. From test *LongEnd* it can be observed that only Opera, Microsoft Edge and Safari, on macOS, soft-failed the TLS connection to a server whose certificate's validity period is of 50 years. They are the only ones that act in compliance with what is stated in the CA/Browser Forum[7]. On the contrary, all other web-browsers accepted the certificate as valid. The subsequent two tests showed that every web-browsers considered valid the certificate with a 200 character Organizational Unit Name and with a serial number longer than the maximum allowed by the RFC-5280 [6]. As expected, the last test of this set, has seen the hard-fail of the TLS connection towards a server whose certificate contains a critical X.509 extension unrecognized by the RP in all the web-browsers but Safari, who opted for a soft-fail.

For what concerns test from 13 to 25 (Table 6.2), those that test the usage of Subject Alternative Names extension, all the web-browsers had the same behavior. As expected, if the Subject Common Name is set and has not an entry in common with the Subject Alternative Names extension, the certificate validation fails. To be more specific, all of them opted for a soft-fail in those cases. The same happens, according to the CA/Browser Forum, when a private IP address or NULL value is used as SAN entry. Not using the SAN extension but just the Subject CN has the same result. From tests *NullCN-SAN_DNS*, *NullCN-2SAN* and *NullCN-SAN_DNS-SAN_IP*, it is needed just one valid entry in the SAN extension to make the certificate pass the validation,

	macOS					Windows				Ubuntu		
	F	C	O	E	S	F	C	O	E	F	C	O
WrongKey	HF	SF	SF	SF	SF	HF	SF	SF	SF	HF	SF	SF
SwapStartEnd	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF
MissingStart	-	-	-	-	-	-	-	-	-	-	-	-
MissingEnd	-	-	-	-	-	-	-	-	-	-	-	-
LongEnd	✓	✓	SF	SF	SF	✓	✓	✓	✓	✓	✓	✓
NullCN	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF
FooCN	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF
TabCN	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF
BackspaceCN	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF
LongOU	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LongRandomSerial	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
UnknownCriticalExt	HF	HF	HF	HF	SF	HF	HF	HF	HF	HF	HF	HF

Table 6.1. Apache server desktop web-browser test 1-12 results. ✓: the web-browser accept the certificate; -: Apache server not even started due to error in certificate format; SF: Soft Fail; HF: Hard Fail; F: Mozilla Firefox; C: Google Chrome; O: Opera; E: Microsoft Edge; S: Safari.

even if Subject CN contains NULL value or if the other entry of the SAN is a private IP address. Test *nullCN-SAN_pubIP*, test *CN-SAN_pubIP* and test *nullCN-nullSAN-and-SAN_pubIP* soft-failed in all cases. The reason why that happened, despite the second can be considered a valid configuration, is that there was a mismatch among the URL used and the SAN entries. In fact, the URL used to connect to the server contained the DNS name associated to it, while the SAN extension of those certificate contained just the public IP address. That is corroborated in test *nullCN-SAN_DNS-and-SAN_pubIP*, which shows a successful connection.

	macOS					Windows				Ubuntu		
	F	C	O	E	S	F	C	O	E	F	C	O
CN-SAN-diff-value	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF
NullCN-SAN_DNS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NullCN-NullSAN_DNS	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF
CN-NoSAN	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF
NullCN-2SAN	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NoCN-SAN_privIP	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF
NullCN-SAN_privIP	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF
CN-SAN_privIP-diff-value	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF
NullCN-SAN_DNS-SAN_IP	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
nullCN-SAN_pubIP	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF
CN-SAN_pubIP	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF
nullCN-SAN_DNS-and-SAN_pubIP	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
nullCN-nullSAN-and-SAN_pubIP	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF

Table 6.2. Apache server desktop web-browser test 13-25 results. ✓: the web-browser accept the certificate; SF: Soft Fail; HF: Hard Fail; F: Mozilla Firefox; C: Google Chrome; O: Opera; E: Microsoft Edge; S: Safari.

Tests in Table 6.3 show that, in a testing environment like the one in which these test have been launched, the absence of revocation information does not affect the validation process. Tests *NoRevocationInfo*, *NoCRL* and *DownOCSP* produced, in fact, the same result. The connection has been accepted. In case of revoked status certificate stapled during TLS handshake, only Mozilla Firefox and Google Chrome, on each system and Opera, on Ubuntu, correctly hard-failed the connection. Lastly, as should it be, the absence of a stapled status for a certificate endowed with the *TLS Feature* extension caused the hard-fail of the TLS connection. Also in this case, only Safari soft-failed the handshake.

	macOS					Windows				Ubuntu		
	F	C	O	E	S	F	C	O	E	F	C	O
NoRevocationInfo	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NoCRL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DownOCSP	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
OCSPStapleRevoked	HF	HF	✓	✓	✓	HF	HF	✓	✓	HF	HF	HF
OCSPMustStapleNoResponse	HF	HF	HF	HF	SF	HF	HF	HF	HF	HF	HF	HF

Table 6.3. Apache server desktop web-browser test 26-30 results. ✓: the web-browser accept the certificate; SF: Soft Fail; HF: Hard Fail; F: Mozilla Firefox; C: Google Chrome; O: Opera; E: Microsoft Edge; S: Safari.

Table 6.4 evidences a pretty correct behavior of the browsers. The most noticeable result is produced by the *RevokedIssuer* test. All the web-browsers, but Mozilla Firefox, which in its default configuration contacts the OCSP servers, let the revocation of the issuer go undetected. Test *NonCAIntermediate* and test *InvalidPathLenConstraint* caused almost all the web-browsers to hard-fail. Test *NotYetValidRoot* passed the certificate validation process in Opera, Microsoft Edge and Safari on macOS and in Google Chrome and Opera on Ubuntu. The usage of the full certificate chain, including the root CA certificate, has been accepted as valid by all of the web-browsers. This is probably due to the fact that the root CA certificate is installed in the client trusted certificates store.

	macOS					Windows				Ubuntu		
	F	C	O	E	S	F	C	O	E	F	C	O
SelfSignedCertificate	HF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF
UnknownIssuer	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF
NonCAIntermediate	SF	HF	HF	HF	SF	SF	HF	HF	HF	SF	HF	HF
InvalidPathLenConstraint	HF	HF	HF	HF	SF	HF	HF	HF	HF	HF	HF	HF
RevokedIssuer	HF	✓	✓	✓	✓	HF	✓	✓	✓	✓	✓	✓
NotYetValidLeaf	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF
NotYetValidInter	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF	SF
NotYetValidRoot	SF	SF	✓	✓	✓	SF	SF	SF	SF	SF	✓	✓
ChainWithRoot	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 6.4. Apache server desktop web-browser test 31-39 results. ✓: the web-browser accept the certificate; SF: Soft Fail; HF: Hard Fail; F: Mozilla Firefox; C: Google Chrome; O: Opera; E: Microsoft Edge; S: Safari.

To analyze the following Key Usage (KU) and Extended Key Usage (EKU) results it is needed to keep trace of the cipher suite negotiated by the different web-browsers. For this thesis purposes,

the servers have been configured to use TLS version 1.2[1]. That is done because TLS 1.2 is still used nowadays and, with respect to TLS 1.3, simpler and faster, uses different algorithms to negotiate the symmetric key to be used for message confidentiality.

The cipher suite negotiated by all the web-browsers is:

- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

That means that the private key of the server should be used for digital signature (*digitalSignature* in KU). This implies that a the valid EKU value should be *Server Authentication*.

In Table 6.5 are collected the results of this last set of tests. It seems that just Safari fully respect the constraints imposed by the KU and EKU extensions. It return a successful validation only in tests *KU_DS_noEKU* and *KU_DS_EKU_SA*. It soft-fails the connection in all other cases. Mozilla Firefox on all the OSs hard-failed the connection when the KU was set to *Data Encipherment* showing an *SEC_ERROR_INADEQAUTE_KEY_USAGE* error message (Figure 6.1). In case the EKU is set to *Client Authentication*, the validation process fails in each web-browser.

	macOS					Windows				Ubuntu		
	F	C	O	E	S	F	C	O	E	F	C	O
KU_KA_noEKU	✓	✓	✓	✓	SF	✓	✓	✓	✓	✓	✓	✓
KU_DE_noEKU	HF	✓	✓	✓	SF	HF	✓	✓	✓	HF	✓	✓
KU_KE_noEKU	✓	✓	✓	✓	SF	✓	✓	✓	✓	✓	✓	✓
KU_DS_noEKU	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
KU_KA_EKU_SA	✓	✓	✓	✓	SF	✓	✓	✓	✓	✓	✓	✓
KU_DE_EKU_SA	HF	✓	✓	✓	SF	HF	✓	✓	✓	HF	✓	✓
KU_KE_EKU_SA	✓	✓	✓	✓	SF	✓	✓	✓	✓	✓	✓	✓
KU_DS_EKU_SA	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
KU_DS_EKU_CA	HF	HF	HF	HF	SF	HF	HF	HF	HF	HF	HF	HF
noKU_EKU_CA	HF	HF	HF	HF	SF	HF	HF	HF	HF	HF	HF	HF

Table 6.5. Apache server desktop web-browser test 40-49 results. ✓: the web-browser accept the certificate; -: Apache server not even started due to error in certificate format; SF: Soft Fail; HF: Hard Fail; F: Mozilla Firefox; C: Google Chrome; O: Opera; E: Microsoft Edge; S: Safari; KA: Key Agreement; KE: Key Encipherment; DE: Data Encipherment; DS: Digital Signature; SA: Server Authentication; CA: Client Authentication; KU: Key Usage; EKU: Extended Key Usage.

All the 49 test did not show any different result when launched using Lighttpd server. Things changed a little when it has been the turn of Nginx.

The tests that produced different results are test 29, *OCSPStapleRevoked* and test 35, *RevokedIssuer*. The cause of this different behavior is in the fact that Nginx, despite it recognizes that the revocation status of the certificate is *revoked*, do not send any OCSP response stapled to the client (Figure 6.2).

[1]There is no difference in the results produced using TLS 1.3 since in this thesis we are testing the RP validation process. In addition the cipher suite negotiated is the same that would have been with TLS 1.3.

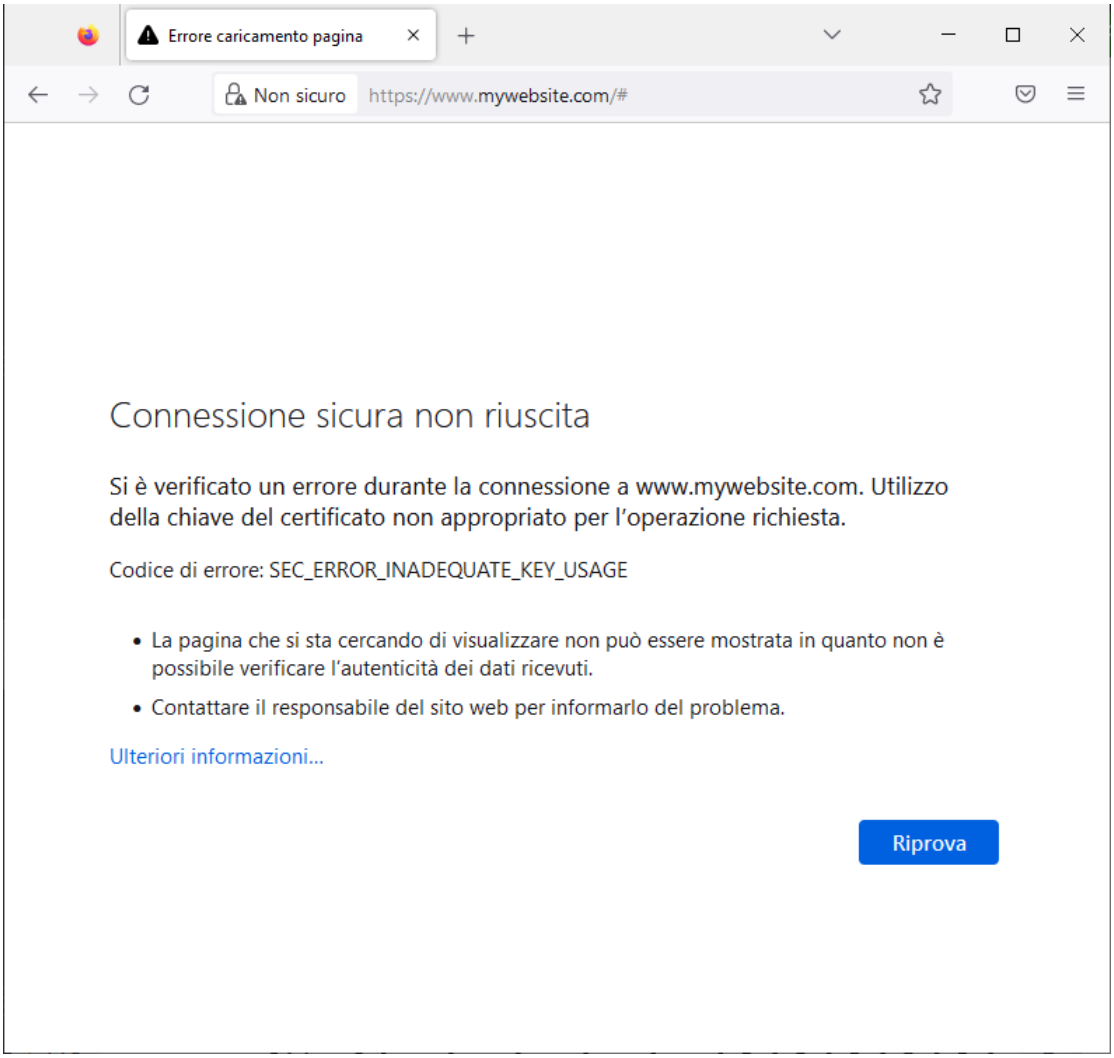

Figure 6.1. Mozilla Firefox Key Usage error message.

```
ubuntu-server@ubuntuserver-VirtualBox:~/test_script$ tail /var/log/nginx/error.log
2022/11/25 16:08:09 [error] 9989#9989: certificate status "revoked" in the OCSP response whi
le requesting certificate status, responder: ocsp.interCA, peer: 127.0.0.1:8081, certificate
: "/home/ubuntu-server/test_script/tests/test_29/certificates/chain.crt"
```

Figure 6.2. Nginx error log entry showing that the revocation status of the certificate of test 29 is revoked.

6.2 Mobile web-browsers results

To test mobile web-browsers' behavior some trick are needed. Since this thesis tests are based on fake certificates that are hosted on fake servers, to make the iOS and the Android devices able to reach them, a DNS spoofing attack via Ettercap has been performed against them. Obviously, on both the devices the fake root certificates have been installed.

The overall behavior of all the interested targets is quite similar. Starting from Table 6.6, it is noticeable that, on iOS, all web-browsers rejected a certificate with a too long validity. In the Android case, the same test passed the validation by every client. Mozilla Firefox, Android version, is the only mobile web-browser that has chosen to hard-fail the connection in test *UnknownCriticalExt*.

	iOS					Android		
	F	C	O	E	S	F	C	E
WrongKey	SF	SF	SF	SF	SF	SF	SF	SF
SwapStartEnd	SF	SF	SF	SF	SF	SF	SF	SF
MissingStart	-	-	-	-	-	-	-	-
MissingEnd	-	-	-	-	-	-	-	-
LongEnd	SF	SF	SF	SF	SF	✓	✓	✓
NullCN	SF	SF	SF	SF	SF	SF	SF	SF
FooCN	SF	SF	SF	SF	SF	SF	SF	SF
TabCN	SF	SF	SF	SF	SF	SF	SF	SF
BackspaceCN	SF	SF	SF	SF	SF	SF	SF	SF
LongOU	✓	✓	✓	✓	✓	✓	✓	✓
LongRandomSerial	✓	✓	✓	✓	✓	✓	✓	✓
UnknownCriticalExt	SF	SF	SF	SF	SF	HF	SF	SF

Table 6.6. Apache server mobile web-browser test 1-12 results. ✓: the web-browser accept the certificate; -: Apache server not even started due to error in certificate format; SF: Soft Fail; HF: Hard Fail; F: Mozilla Firefox; C: Google Chrome; O: Opera; E: Microsoft Edge; S: Safari.

Moving on to Table 6.8 the pattern is exactly the same for tests against desktop web-browsers. A quite interesting difference appears in Table 6.9, in test *OCSPStapleRevoked*. Mozilla Firefox, Android version, is the only one able to recognize the revoked certificate.

The tests' results in Table 6.10 show a quite similar mobile clients' behavior to their desktop versions. None of the mobile clients reject the connection with a server whose certificate has been issued by a revoked CA certificate. The Android versions accepted, also, a certificate whose root CA certificate is yet not valid. Lastly, all of them successfully connected to a server which provided them a certificate chain with the root CA certificate in it. This probably, again, because the root certificates are installed at client-side.

The mobile web-browsers negotiated the same cipher suite as their corresponding desktop versions. There are some differences though. First of all, even though Google Chrome and Microsoft Edge accepted the connection, they marked the website as insecure (absence of the lock icon near the URL) in all tests that has KU different from *Digital Signature*. The Android mobile versions behaved exactly as in the desktop case. The connection failed in all of them in case of EKU set to *Client Authentication*. The results are collected in Table 6.11

Also for mobile tests there is no difference with the results obtained hosting the server with Lighttpd. Always for the same reason exposed above, with Nginx, test 29, *OCSPStapleRevoked* passed the validation also on Mozilla Firefox version Android.

	iOS					Android		
	F	C	O	E	S	F	C	E
CN-SAN-diff-value	SF	SF	SF	SF	SF	SF	SF	SF
NullCN-SAN_DNS	✓	✓	✓	✓	✓	✓	✓	✓
NullCN-NullSAN_DNS	SF	SF	SF	SF	SF	SF	SF	SF
CN-NoSAN	SF	SF	SF	SF	SF	SF	SF	SF
NullCN-2SAN	✓	✓	✓	✓	✓	✓	✓	✓
NoCN-SAN_privIP	SF	SF	SF	SF	SF	SF	SF	SF
NullCN-SAN_privIP	SF	SF	SF	SF	SF	SF	SF	SF
CN-SAN_privIP-diff-value	SF	SF	SF	SF	SF	SF	SF	SF
NullCN-SAN_DNS-SAN_IP	✓	✓	✓	✓	✓	✓	✓	✓
nullCN-SAN_pubIP	SF	SF	SF	SF	SF	SF	SF	SF
CN-SAN_pubIP	SF	SF	SF	SF	SF	SF	SF	SF
nullCN-SAN_DNS-and-SAN_pubIP	✓	✓	✓	✓	✓	✓	✓	✓
nullCN-nullSAN-and-SAN_pubIP	SF	SF	SF	SF	SF	SF	SF	SF

Table 6.8. Apache server mobile web-browser test 13-25 results. ✓: the web-browser accept the certificate; SF: Soft Fail; HF: Hard Fail; F: Mozilla Firefox; C: Google Chrome; O: Opera; E: Microsoft Edge; S: Safari.

	iOS					Android		
	F	C	O	E	S	F	C	E
NoRevocationInfo	✓	✓	✓	✓	✓	✓	✓	✓
NoCRL	✓	✓	✓	✓	✓	✓	✓	✓
DownOCSP	✓	✓	✓	✓	✓	✓	✓	✓
OCSPStapleRevoked	✓	✓	✓	✓	✓	HF	✓	✓
OCSPMustStapleNoResponse	SF	SF	SF	SF	SF	HF	SF	SF

Table 6.9. Apache server mobile web-browser test 26-30 results. ✓: the web-browser accept the certificate; SF: Soft Fail; HF: Hard Fail; F: Mozilla Firefox; C: Google Chrome; O: Opera; E: Microsoft Edge; S: Safari.

	iOS					Android		
	F	C	O	E	S	F	C	E
SelfSignedCertificate	SF	SF	SF	SF	SF	SF	SF	SF
UnknownIssuer	SF	SF	SF	SF	SF	SF	SF	SF
NonCAIntermediate	SF	SF	SF	SF	SF	SF	SF	SF
InvalidPathLenContraint	SF	SF	SF	SF	SF	HF	SF	SF
RevokedIssuer	✓	✓	✓	✓	✓	✓	✓	✓
NotYetValidLeaf	SF	SF	SF	SF	SF	SF	SF	SF
NotYetValidIssuer	SF	SF	SF	SF	SF	SF	SF	SF
NotYetValidRoot	SF	SF	SF	SF	SF	SF	✓	✓
ChainWithRoot	✓	✓	✓	✓	✓	✓	✓	✓

Table 6.10. Apache server mobile web-browser test 31-39 results. ✓: the web-browser accept the certificate; SF: Soft Fail; HF: Hard Fail; F: Mozilla Firefox; C: Google Chrome; O: Opera; E: Microsoft Edge; S: Safari.

	iOS					Android		
	F	C	O	E	S	F	C	E
KU_KA_noEKU	✓	✓*	✓	✓*	✓	✓	✓	✓
KU_DE_noEKU	✓	✓*	✓	✓*	✓	HF	✓	✓
KU_KE_noEKU	✓	✓*	✓	✓*	✓	✓	✓	✓
KU_DS_noEKU	✓	✓	✓	✓	✓	✓	✓	✓
KU_KA_EKU_SA	✓	✓*	✓	✓*	✓	✓	✓	✓
KU_DE_EKU_SA	✓	✓*	✓	✓*	✓	HF	✓	✓
KU_KE_EKU_SA	✓	✓*	✓	✓*	✓	✓	✓	✓
KU_DS_EKU_SA	✓	✓	✓	✓	✓	✓	✓	✓
KU_DS_EKU_CA	SF	SF	SF	SF	SF	HF	HF	HF
noKU_EKU_CA	SF	SF	SF	SF	SF	HF	HF	HF

Table 6.11. Apache server mobile web-browser test 40-49 results. ✓: the web-browser accept the certificate; SF: Soft Fail; HF: Hard Fail; F: Mozilla Firefox; C: Google Chrome; O: Opera; E: Microsoft Edge; S: Safari; *: Absence of lock icon near the URL; KA: Key Agreement; KE: Key Encipherment; DE: Data Encipherment; DS: Digital Signature; SA: Server Authentication; CA: Client Authentication; KU: Key Usage; EKU: Extended Key Usage.

6.3 TLS interception products results

TLS interception products showed different certificate generation approaches from each other.

Mitmproxy generates always a valid certificate to present to the client, signed by its root certificate, with a validity period of one year. From the original certificate keeps the Subject *Common Name*, the Subject *Organization Name*, the *Subject Alternative Names* extension and the *Extended Key Usage* extension (Figure 6.3). In case of error in the validation of the certificate it just shows an error page at client-side.

Same thing is done by Squid, which keeps also the validity period of the original certificate when issues its own one. It inserts in it the *Distinguished Name* attributes of the subject too (Figure 6.4).

Kaspersky Total Security, in case of not valid certificate, could show either a soft-fail page or an hard-fail one but the certificate submitted is always valid. This antivirus also issues one-year-valid certificates (Figure 6.5).

Slightly different is the approach used by ESET Smart Security. It generates two different kind of certificates in case of error. In particular, ESET could issue a certificate with its own root certificate, but leave the validation to the client (delegated validation) or could issue a particular certificate issued by an authority named: *The original certificate provided by the server is untrusted* after its own validation (Figure 6.7). This causes a validation failure at client-side with error code *SEC_ERROR_UNKNOWN_ISSUER*. The former case is indicated with → in results, while the latter case is seen as soft-fail. In addition, ESET generates certificates containing all the information in the original one but *Authority Information Access* extension and *CRL Distribution Point* Extension (Figure 6.6).

The actual interception process of the antivirus tools is not described in their official documentation. For this reason a traffic analysis has been made using *Wireshark*. The analysis wanted to inspect how the client machine sees the outgoing connection towards the server and how the server sees the incoming connection from the client. On both of them, the traffic has been sniffed. From the client machine, using Google Chrome, a TLS handshake towards a valid server in started.

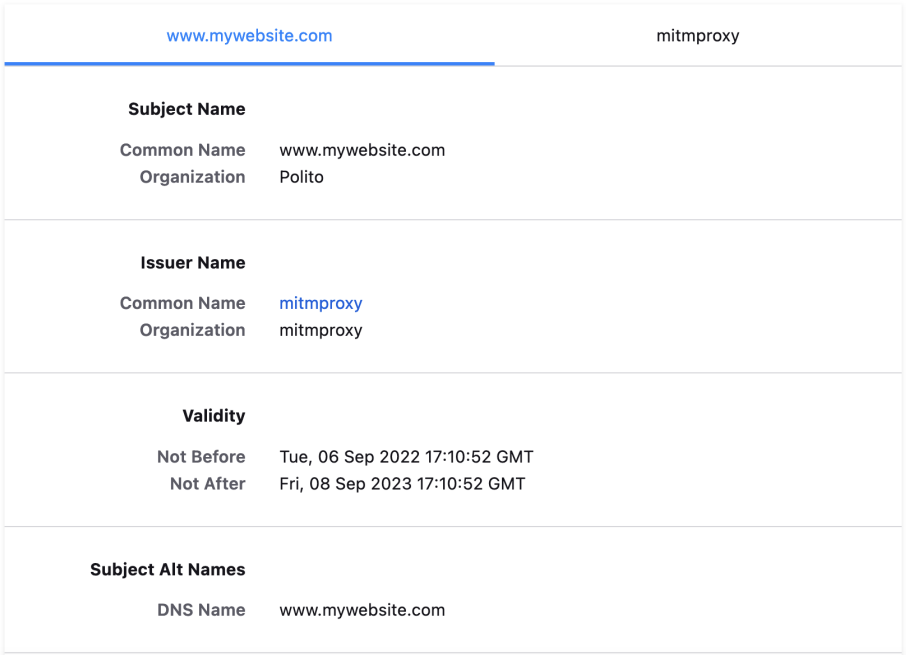

Figure 6.3. Mitmproxy certificate submitted to the client.

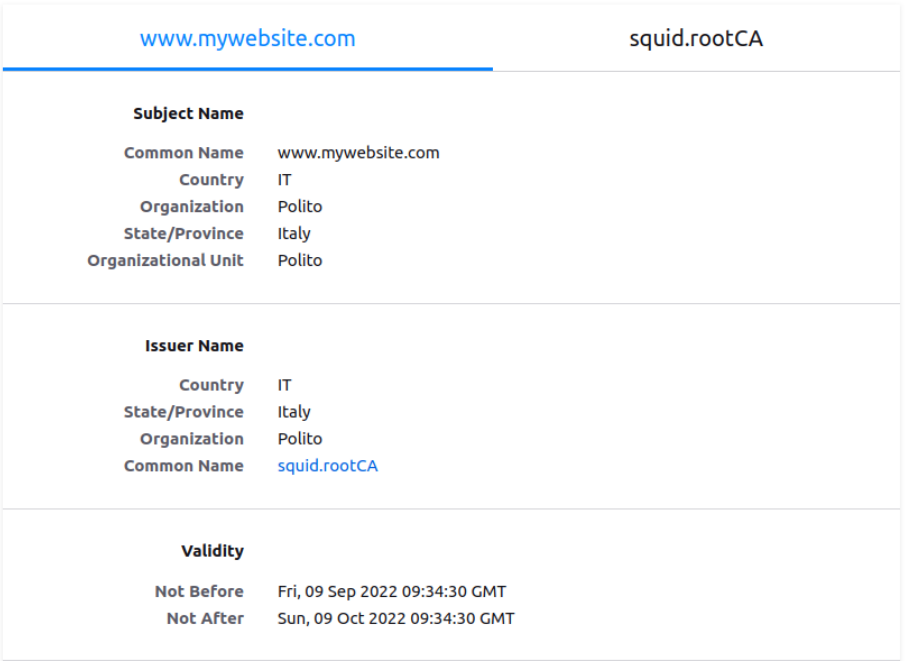

Figure 6.4. Squid proxy certificate submitted to the client.

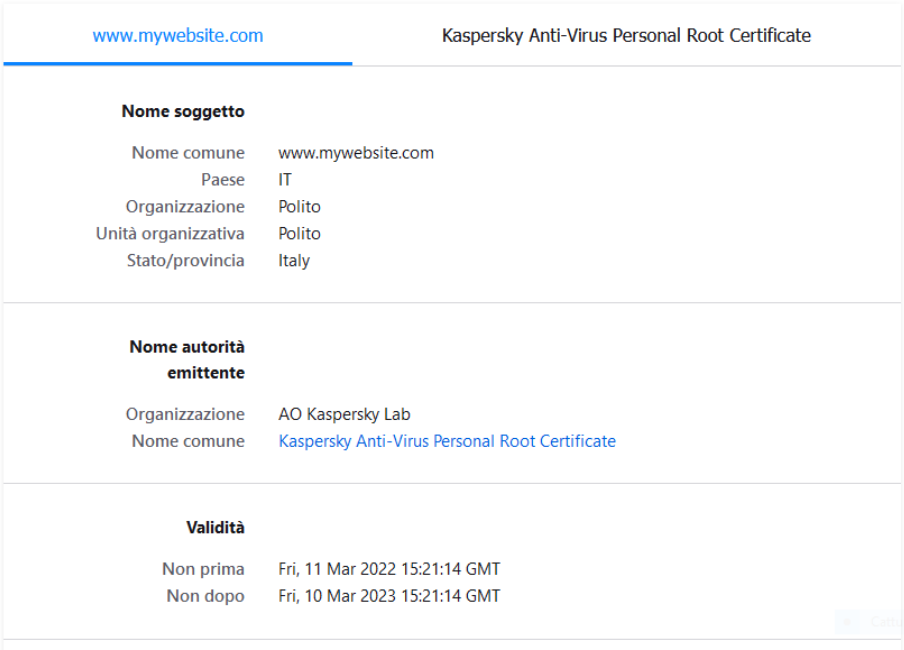

Figure 6.5. Kaspersky Total Security certificate submitted to the client.

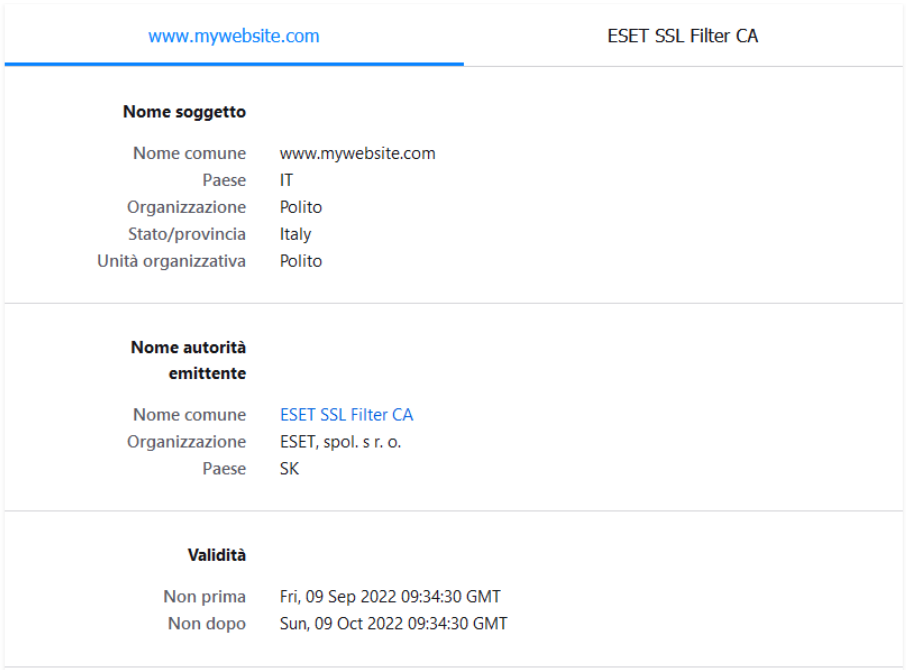

Figure 6.6. ESET Smart Security certificate submitted to the client.

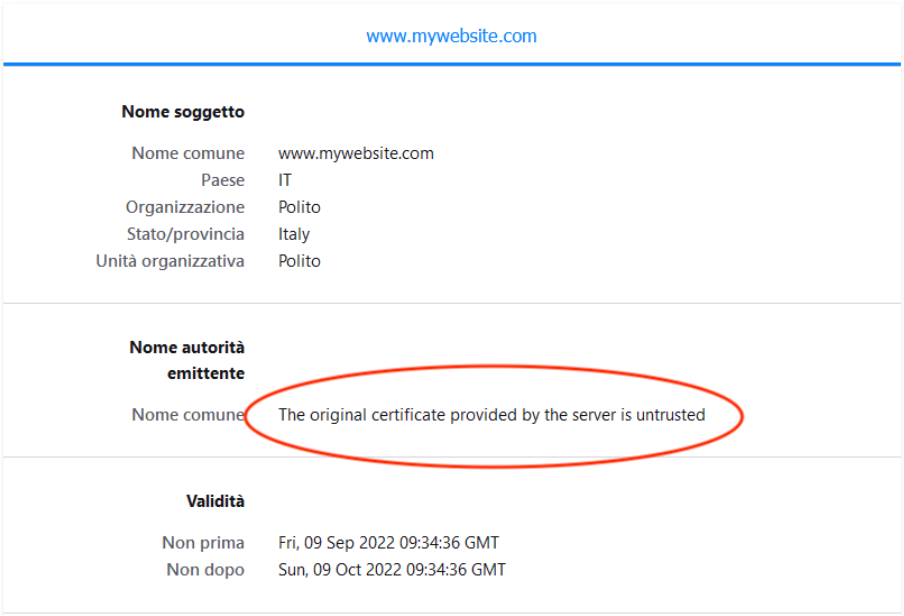

Figure 6.7. ESET Smart Security certificate after failed verification.

For what concerns ESET Smart Security, both the parties in the connection see a single TLS connection. The browser, anyway, show a certificate signed by the *ESET SSL Filter CA*. This manipulation is performed on the machine itself after receiving the original certificate from the server.

What is concerning, is the behavior of Kaspersky Total Security. First of all, from the client machine there is no connection towards the server IP address 192.168.0.106 (Figure 6.8). Among the different servers contacted by the browser, there is an occasional connection to a RIPE database [2] server, which is managed by Kaspersky (Figure 6.9). From the server machine, a direct connection with the client is seen (Figure 6.10). According to a layer 2 inspection, the source MAC address in the *ClientHello* message comes directly from the client machine. At client-side the certificate of the server is the one signed on-the-fly by Kaspersky (Figure 6.12). In conclusion, the actual process through which Kaspersky Total Security generates its certificates remains undefined.

Focusing on the first set of tests (Table 6.12), all the products let pass as valid tests *LongEnd*, *LongOU* and *LongRandomSerial*. Mitmproxy and Squid proxy correctly validated the other test certificates showing an hard-fail error page. Kaspersky opted, instead, for a soft-fail in those cases. ESET delegated the validation of the certificates in *SwapStartEnd*, *NullCN*, *FooCN*, *TabCN*, *BackspaceCN* to the client (Mozilla Firefox). The certificates generated by ESET for those tests were issued by the ESET root CA certificate.

The behavior of those products diverges more when it comes the turn of SAN extension usage tests (Table 6.13). Mitmproxy block all connections but those in tests *NullCN-SAN_DNS*,

[2]It is a public database that contains information about registered IP address space and AS Numbers, routing policies, and reverse DNS delegations in the Réseaux IP Européens Network Coordination Centre (RIPE NCC) service region

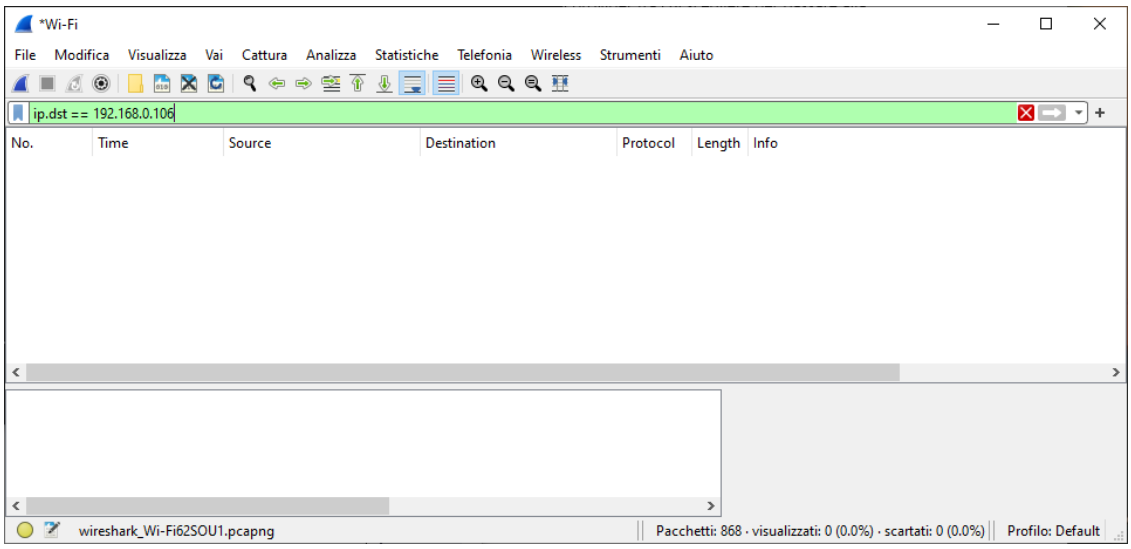

Figure 6.8. Client side packet capture. No connection towards 192.168.0.106, the IP address of the server VM.

	macOS		Windows				Ubuntu	
	M	S	M	S	K	E	M	S
WrongKey	HF	HF	HF	HF	SF	SF	HF	HF
SwapStartEnd	HF	HF	HF	HF	SF	→	HF	HF
MissingStart	-	-	-	-	-	-	-	-
MissingEnd	-	-	-	-	-	-	-	-
LongEnd	✓	✓	✓	✓	✓	✓	✓	✓
NullCN	HF	HF	HF	HF	SF	→	HF	HF
FooCN	HF	HF	HF	HF	SF	→	HF	HF
TabCN	HF	HF	HF	HF	SF	→	HF	HF
BackspaceCN	HF	HF	HF	HF	SF	→	HF	HF
LongOU	✓	✓	✓	✓	✓	✓	✓	✓
LongRandomSerial	✓	✓	✓	✓	✓	✓	✓	✓
UnknownCriticalExt	HF	HF	HF	HF	SF	SF	HF	HF

Table 6.12. TLS interception products tests 1-12 results. M: Mitmproxy; S: Squid; K: Kaspersky Total Security; A: ESET Smart Security; ✓: the web-browser accept the certificate; -: Apache server not even started due to error in certificate format; SF: Soft Fail; HF: Hard Fail. →: validation delegated to the client.

No.	Time	Source	Destination	Protocol	Length	Info
1062	19.530555	192.168.0.2	195.122.177.184	TCP	66	62931 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
1063	19.548775	195.122.177.184	192.168.0.2	TCP	60	443 → 62931 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1452
1064	19.548902	192.168.0.2	195.122.177.184	TCP	54	62931 → 443 [ACK] Seq=1 Ack=1 Win=64240 Len=0
1065	19.549533	192.168.0.2	195.122.177.184	TLSv1.2	627	Client Hello
1066	19.639478	195.122.177.184	192.168.0.2	TLSv1.2	204	Server Hello, Change Cipher Spec, Encrypted Handshake Message
1067	19.639798	192.168.0.2	195.122.177.184	TLSv1.2	105	Change Cipher Spec, Encrypted Handshake Message
1068	19.639967	192.168.0.2	195.122.177.184	TLSv1.2	107	Application Data
1069	19.640060	192.168.0.2	195.122.177.184	TLSv1.2	110	Application Data
1070	19.640105	192.168.0.2	195.122.177.184	TLSv1.2	96	Application Data
1071	19.640213	192.168.0.2	195.122.177.184	TLSv1.2	200	Application Data
1072	19.640315	192.168.0.2	195.122.177.184	TLSv1.2	153	Application Data
1073	19.659000	195.122.177.184	192.168.0.2	TCP	60	443 → 62931 [ACK] Seq=151 Ack=678 Win=64123 Len=0
1074	19.659002	195.122.177.184	192.168.0.2	TLSv1.2	98	Application Data
1075	19.659003	195.122.177.184	192.168.0.2	TLSv1.2	92	Application Data
1076	19.659003	195.122.177.184	192.168.0.2	TCP	60	443 → 62931 [ACK] Seq=233 Ack=922 Win=64067 Len=0
1077	19.659004	195.122.177.184	192.168.0.2	TLSv1.2	109	Application Data
1078	19.659005	195.122.177.184	192.168.0.2	TLSv1.2	167	Application Data
1079	19.659006	195.122.177.184	192.168.0.2	TLSv1.2	168	Application Data
1080	19.659111	192.168.0.2	195.122.177.184	TCP	54	62931 → 443 [ACK] Seq=1021 Ack=515 Win=65340 Len=0
1081	19.659215	192.168.0.2	195.122.177.184	TLSv1.2	92	Application Data
1082	19.718965	195.122.177.184	192.168.0.2	TCP	56	443 → 62931 [ACK] Seq=515 Ack=1059 Win=64067 Len=0

```
matteo@LAPTOP-E6HT09A5:~$ whois 195.122.177.184
% This is the RIPE Database query service.
% The objects are in RPSL format.
%
% The RIPE Database is subject to Terms and Conditions.
% See http://www.ripe.net/db/support/db-terms-conditions.pdf

% Note: this output has been filtered.
%       To receive output for a database update, use the "-B" flag.

% Information related to '195.122.177.128 - 195.122.177.255'

% Abuse contact for '195.122.177.128 - 195.122.177.255' is 'abuse@level3.com'

inetnum:        195.122.177.128 - 195.122.177.255
netname:        KASPERSKY-LAB-UK-LTD
descr:          KASPERSKY LAB LAN
country:        DE
admin-c:        SA7294-RIPE
tech-c:         SA7294-RIPE
status:         ASSIGNED PA
remarks:        all abuse reports to abuse@level3.com
mnt-by:         LEVEL3-MNT
mnt-lower:      LEVEL3-MNT
mnt-routes:     LEVEL3-MNT
created:        2013-09-25T11:03:32Z
last-modified:  2013-09-25T11:03:32Z
source:         RIPE # Filtered
```

Figure 6.9. Client side packet capture. On top TLS connection with an external server. On bottom whois command run against that server, showing it is managed by Kaspersky.

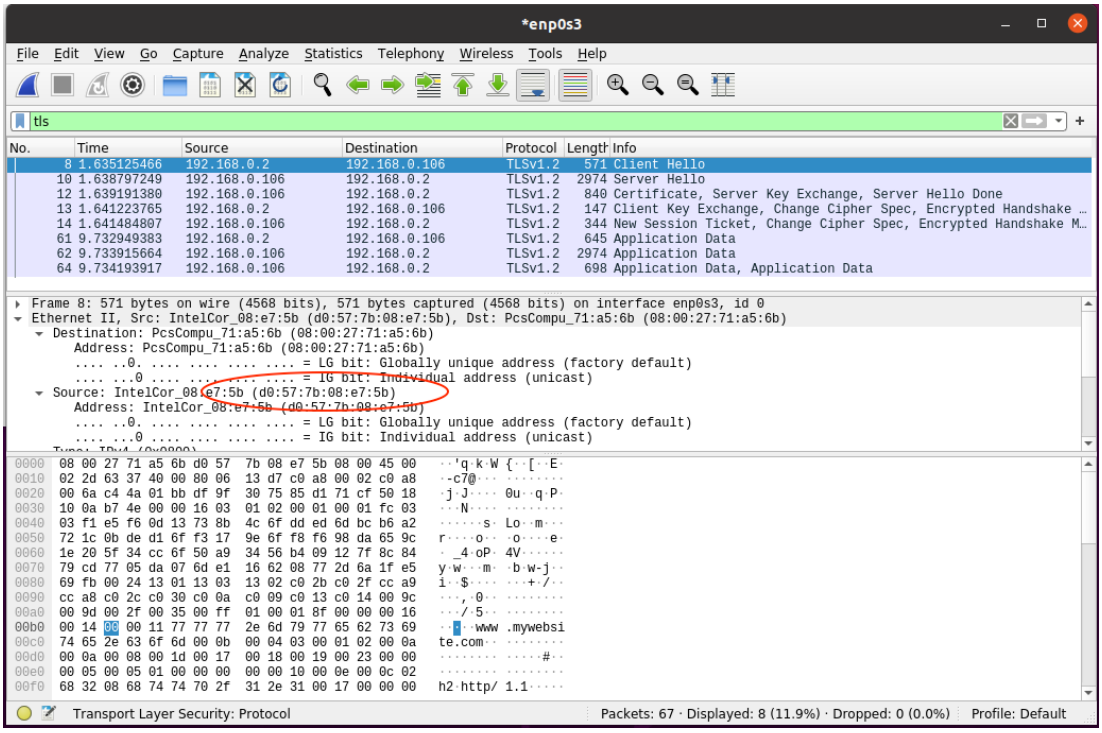

Figure 6.10. Server side packet capture. Source MAC address is the one of the machine used as client.

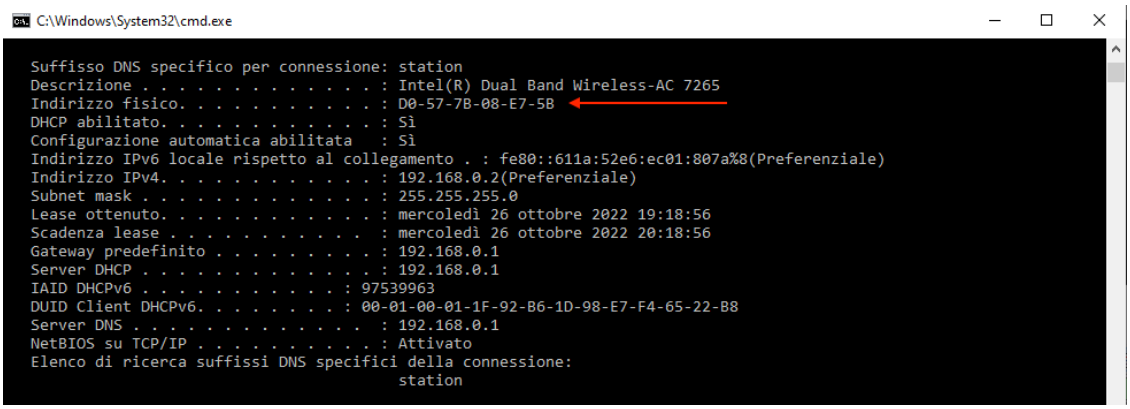

Figure 6.11. Network information taken from the Kaspersky Total Security Windows 10 machine.

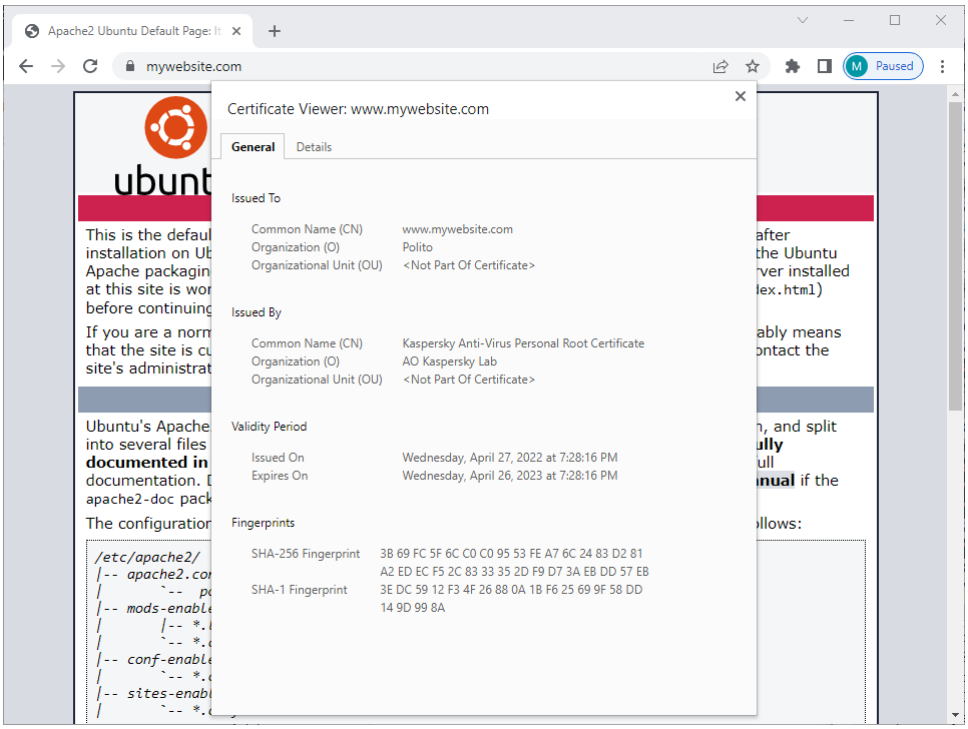

Figure 6.12. Certificate signed by Kaspersky CA.

NullCN-2SAN, *NullCN-SAN_DNS-SAN_IP* and *nullCN-SAN_DNS-and-SAN_pubIP*. For them it generates its own certificates and opts for a delegated validation, which resulted in a warning message by the client.

These last tests have been, on the contrary, recognized as valid by Squid proxy. Squid delegated the validation of the certificates generated for tests *CN-SAN-diff-value*, *CN-NoSAN*, *CN-SAN_privIP-diff-value* and *CN-SAN_pubIP*. It blocked all the others.

Kaspersky let pass the validation of all those certificates in which just one among the Subject CN and the SAN entries has a valid value. In all other cases it cause a soft-fail of the TLS connection and an hard-fail for test *nullCN-nullSAN-and-SAN_pubIP* .

ESET Smart Security considers valid only those certificates that have at least a SAN entry configured properly. In all the other cases generates a certificate with its self-signed certificate and delegates the validation to the client which showed a warning message every time.

None of the TLS interception products rejected the connection if they are not able to retrieve revocation information. All of them but ESET showed a lack of support for OCSP Stapling since they accepted a revoked certificate. Actually, Kaspersky do something unexpected. It downloads, when it can, the CRL using the *CRL Distribution Point* extension in the certificates, but this downloaded CRL is not used. Otherwise the revoked certificate should have caused the failure of the validation. ESET, on the other hand, correctly recognizes that the certificate is revoked and block the connection (Figure 6.13).

All of them do not support OCSP Must-Staple but reject the connection since it is an unrecognized X.509 critical extension.

	macOS		Windows				Ubuntu	
	M	S	M	S	K	E	M	S
CN-SAN-diff-value	HF	→	HF	→	SF	→	HF	→
NullCN-SAN_DNS	→	✓	→	✓	✓	✓	→	✓
NullCN-NullSAN_DNS	HF	HF	HF	HF	HF	→	HF	HF
CN-NoSAN	HF	→	HF	→	✓	→	HF	→
NullCN-2SAN	→	✓	→	✓	✓	✓	→	✓
NoCN-SAN_privIP	HF	HF	HF	HF	SF	→	HF	HF
NullCN-SAN_privIP	HF	HF	HF	HF	SF	→	HF	HF
CN-SAN_privIP-diff-value	HF	→	HF	→	✓	→	HF	→
NullCN-SAN_DNS-SAN_IP	→	✓	→	✓	✓	✓	→	✓
nullCN-SAN_pubIP	HF	HF	HF	HF	SF	→	HF	HF
CN-SAN_pubIP	HF	→	HF	→	✓	→	HF	→
nullCN-SAN_DNS-and-SAN_pubIP	→	✓	→	✓	✓	✓	→	✓
nullCN-nullSAN-and-SAN_pubIP	HF	HF	HF	HF	HF	→	HF	HF

Table 6.13. TLS interception products tests 13-25 results. M: Mitmproxy; S: Squid; K: Kaspersky Total Security; A: ESET Smart Security; ✓: the web-browser accept the certificate; SF: Soft Fail; HF: Hard Fail. →: validation delegated to the client.

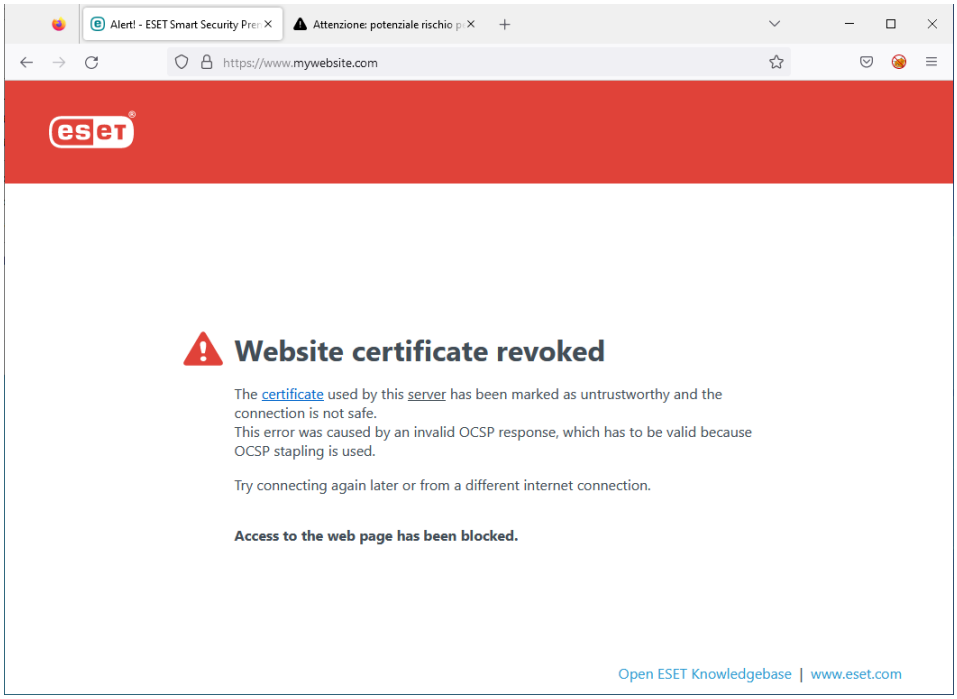

Figure 6.13. ESET Smart Security revoked certificate rejection.

Set of results from 31 to 39 (Table 6.15) demonstrated also that those interceptor are not able to recognize that a certificate has been issued by a revoked CA certificate. While the other products do not even download the CRLs or the CA certificates from the stores available respectively in the *CRL Distribution Point* and in the *Authority Information Access* extensions, Kaspersky did

	macOS		Windows				Ubuntu	
	M	S	M	S	K	E	M	S
NoRevocationInfo	✓	✓	✓	✓	✓	✓	✓	✓
NoCRL	✓	✓	✓	✓	✓	✓	✓	✓
DownOCSP	✓	✓	✓	✓	✓	✓	✓	✓
OCSPStapleRevoked	✓	✓	✓	✓	✓	HF	✓	✓
OCSPMustStapleNoResponse	HF	HF	HF	HF	SF	SF	HF	HF

Table 6.14. TLS interception products tests 26-30 results. M: Mitmproxy; S: Squid; K: Kaspersky Total Security; A: ESET Smart Security; ✓: the web-browser accept the certificate; -: Apache server not even started due to error in certificate format; SF: Soft Fail; HF: Hard Fail. →: validation delegated to the client.

it (Figure 6.14). Also in this case the CRL downloaded in test 35, *RevokedIssuer*, should have caused the rejection of the connection, but it did not happen.

```
ubuntu-server@ubuntuserver-VirtualBox:~/test_script$ sudo python3 -m http.server 8082 --directory /var/www/htr
[sudo] password for ubuntu-server:
Serving HTTP on 0.0.0.0 port 8082 (http://0.0.0.0:8082/) ...
192.168.0.2 - - [10/Oct/2022 18:15:55] "GET /download/rootCA.crl HTTP/1.1" 200 -
192.168.0.2 - - [10/Oct/2022 18:15:55] "GET /download/rootCA.crl HTTP/1.1" 200 -
192.168.0.2 - - [10/Oct/2022 18:17:13] "GET /download/rootCA.crl HTTP/1.1" 200 -
192.168.0.2 - - [10/Oct/2022 18:17:13] "GET /download/rootCA.crl HTTP/1.1" 200 -
192.168.0.2 - - [10/Oct/2022 18:17:36] "GET /download/rootCA.crl HTTP/1.1" 200 -
192.168.0.2 - - [10/Oct/2022 18:17:36] "GET /download/rootCA.crl HTTP/1.1" 200 -
192.168.0.2 - - [10/Oct/2022 18:19:22] "GET /download/rootCA.crl HTTP/1.1" 200 -
192.168.0.2 - - [10/Oct/2022 18:19:22] "GET /download/rootCA.crl HTTP/1.1" 200 -
192.168.0.2 - - [10/Oct/2022 18:24:26] "GET /download/rootCA.crl HTTP/1.1" 200 -
192.168.0.2 - - [10/Oct/2022 18:24:26] "GET /download/rootCA.crl HTTP/1.1" 200 -
192.168.0.2 - - [10/Oct/2022 18:28:09] "GET /download/rootCA_2.crt HTTP/1.1" 200 -
192.168.0.2 - - [10/Oct/2022 18:33:02] "GET /download/rootCA.crl HTTP/1.1" 200 -
192.168.0.2 - - [10/Oct/2022 18:33:02] "GET /download/rootCA.crl HTTP/1.1" 200 -
192.168.0.2 - - [10/Oct/2022 18:33:26] "GET /download/rootCA.crl HTTP/1.1" 200 -
192.168.0.2 - - [10/Oct/2022 18:33:26] "GET /download/rootCA.crl HTTP/1.1" 200 -
192.168.0.2 - - [10/Oct/2022 18:41:55] "GET /download/rootCA.crl HTTP/1.1" 200 -
192.168.0.2 - - [10/Oct/2022 18:41:55] "GET /download/rootCA.crl HTTP/1.1" 200 -
192.168.0.2 - - [10/Oct/2022 18:43:49] "GET /download/rootCA.crl HTTP/1.1" 200 -
192.168.0.2 - - [10/Oct/2022 18:43:49] "GET /download/rootCA.crl HTTP/1.1" 200 -
```

Figure 6.14. Kaspersky running on machine with IP address 192.168.0.2 downloading the CRL from the store indicated in CRLDistributionPoint extension in the certificate.

Mitmproxy and Squid blocked all the connections but the one in which the server provided the certificate chain including the root CA certificate. To be more precise, the Mitmproxy version running on macOS rejected such a connection, even though it was configured to accept certificates issued by that root CA certificate.

Kaspersky let *RevokedIssuer* and *ChainWithRoot* certificates pass the validation process. *Non-CAIntermediate* and *InavalidPathLenContraint* tests resulted in an hard-fail of the TLS connection. The remaining tests in the set generated a soft-fail.

ESET always opted for a soft-fail of the TLS connection but for test *NotYetValidLeaf*. In that case it decided for a delegated validation, which resulted in a warning message by the client. Also, ESET recognized as valid a chain containing the root CA certificate. Also in this case, ESET showed an error message blocking the connection for test *RevokedIssuer*

The cipher suites negotiated by the TLS interception products are collected in Table 6.16.

Also in this case the correct KU should be *Digital Signature* and the EKU associated *Server Authentication*. Table 6.17 contains the results of this set of test passing through the TLS interception products.

	macOS		Windows				Ubuntu	
	M	S	M	S	K	E	M	S
SelfSignedCertificate	HF	HF	HF	HF	SF	SF	HF	HF
UnknownIssuer	HF	HF	HF	HF	SF	SF	HF	HF
NonCAIntermediate	HF	HF	HF	HF	HF	SF	HF	HF
InvalidPathLenConstraint	HF	HF	HF	HF	HF	SF	HF	HF
RevokedIssuer	✓	✓	✓	✓	✓	HF	✓	✓
NotYetValidLeaf	HF	HF	HF	HF	SF	→	HF	HF
NotYetValidInter	HF	HF	HF	HF	SF	SF	HF	HF
NotYetValidRoot	HF	HF	HF	HF	SF	SF	HF	HF
ChainWithRoot	HF	✓	✓	✓	✓	✓	✓	✓

Table 6.15. TLS interception products tests 31-39 results. M: Mitmproxy; S: Squid; K: Kaspersky Total Security; A: ESET Smart Security; ✓: the web-browser accept the certificate; SF: Soft Fail; HF: Hard Fail. →: validation delegated to the client.

Mitmproxy	Squid	Kasperky	ESET
TLS_ECDHE_RSA_ WITH_AES_128_ GCM_SHA256	TLS_ECDHE_RSA_ WITH_AES_256_ GCM_SHA384	TLS_ECDHE_RSA_ WITH_AES_128_ GCM_SHA256	TLS_ECDHE_RSA_ WITH_AES_128_ GCM_SHA256

Table 6.16. TLS 1.2 cipher suites negotiated by the TLS interception products examinated.

Mitmproxy and Squid accepted as valid all those certificate with KU different from *Data Encipherment* and those in which the EKU is set to *Client Authentication*.

Kaspersky Total Security and ESET Smart Security soft-failed the TLS connection only when the EKU of the certificate is set to *Client Authentication*.

	macOS		Windows				Ubuntu	
	M	S	M	S	K	E	M	S
KU_KA_noEKU	✓	✓	✓	✓	✓	✓	✓	✓
KU_DE_noEKU	HF	HF	HF	HF	✓	✓	HF	HF
KU_KE_noEKU	✓	✓	✓	✓	✓	✓	✓	✓
KU_DS_noEKU	✓	✓	✓	✓	✓	✓	✓	✓
KU_KA_EKU_SA	✓	✓	✓	✓	✓	✓	✓	✓
KU_DE_EKU_SA	HF	HF	HF	HF	✓	✓	HF	HF
KU_KE_EKU_SA	✓	✓	✓	✓	✓	✓	✓	✓
KU_DS_EKU_SA	✓	✓	✓	✓	✓	✓	✓	✓
KU_DS_EKU_CA	HF	HF	HF	HF	SF	SF	HF	HF
noKU_EKU_CA	HF	HF	HF	HF	SF	SF	HF	HF

Table 6.17. TLS interception products tests 40-49 results. M: Mitmproxy; S: Squid; K: Kaspersky Total Security; A: ESET Smart Security; ✓: the web-browser accept the certificate; SF: Soft Fail; HF: Hard Fail. →: validation delegated to the client; KA: Key Agreement; KE: Key Encipherment; DE: Data Encipherment; DS: Digital Signature; SA: Server Authentication; CA: Client Authentication; KU: Key Usage; EKU: Extended Key Usage.

There are no differences with the results obtained from Lighttpd this time too. Nginx results,

instead, show a different behavior of ESET Smart Security when, again, OCSP Stapling in involved. In both the cases (tests 29 and 35) it let the revoked certificate or certificate issued by a revoked CA certificate pass the validation process.

6.4 Top-1-Million analysis results

The script collecting information from the Majestic Top-1-Million domains [12] run in parallel on two Ubuntu machines for more than a week. 271,817 of the domains in the Majestic ranking were unreachable. So only about the 72.82% of the domains have been used to produce the charts and results. Table 6.18 shows the OCSP Stapling and Must-Staple resulting adoption in 2022.

	OCSP Stapling	OCSP Must-Staple
2018	19%	0.04%
2019	27%	0%
2022	31%	0%

Table 6.18. OCSP Stapling and OCSP Must-Staple support in 2018(Source [2]), 2019[2] and 2022.

Only the 31.469% of the status request performed during the TLS handshake returned an OCSP response. None of the certificates retrieved from the domains contained the *TLS Feature* extension.

The subsequent pie chart produced by the script shows the TLS versions negotiated between the client and the server.

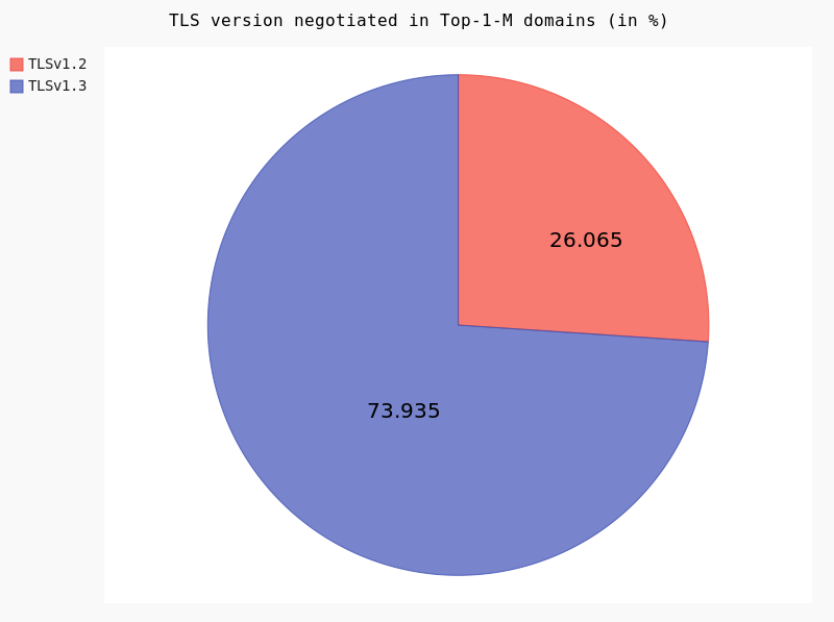

Figure 6.15. TLS versions negotiated in the Top-1-Million domains reached by the script.

According to the analysis, there is still about a quarter of the servers across the world which still uses TLS 1.2 against the 73.935% of the connections which negotiated TLS 1.3 (Figure 6.15).

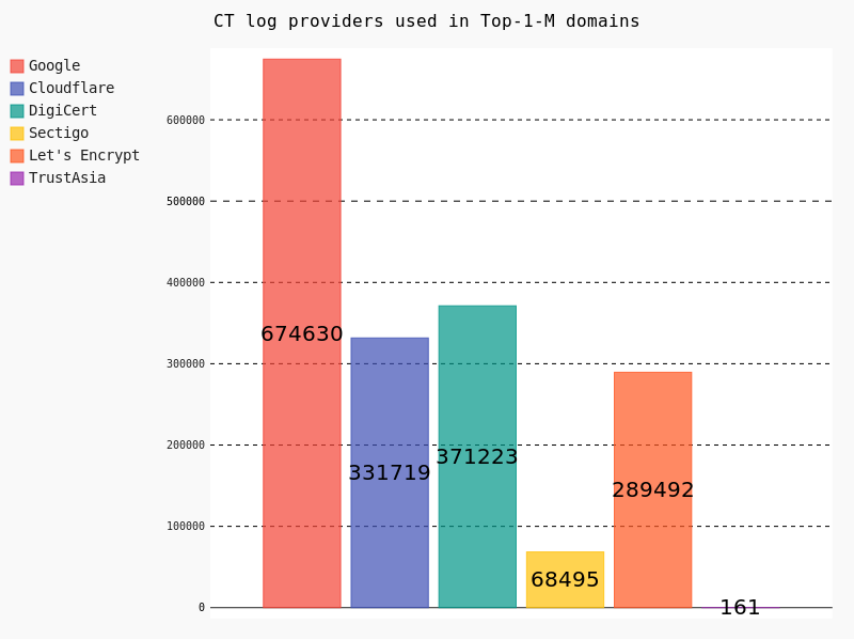

Figure 6.16. CT log server providers used in the Majestic Top-1-Million domains reached by the script.

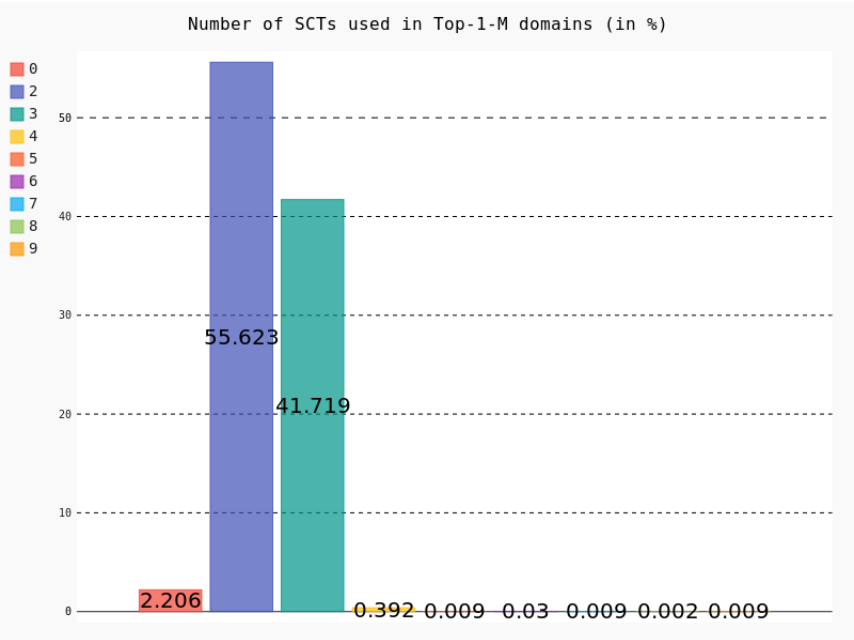

Figure 6.17. Number of SCTs in the certificates retrieved from the Majestic Top-1-Million domains reached by the script.

Figure 6.16 shows that the most used logs are provided by Google with 674,630 SCTs embedded in the certificates. The second most used provider is Digicert, with 371,223 SCTs pointing its log servers. Going on, Cloudflare, with 331,719 SCTs. Then, it is the turn of Let's Encrypt with 289,492 SCTs. The less used providers are Sectigo and TrustAsia. Sectigo has been used 68,495 times, while TrustAsia just in 161 cases.

Every certificate can contain an arbitrary number of SCTs. The more SCTs they have, the more trustworthy is the authenticity of the certificate. The second chart produced by the script (Figure 6.17) wanted to study, in percentage, how many of these different timestamps are embedded within a single certificate. The most of them uses 2 or 3 SCTs in their certificates. To be more precise, the 55.623% uses 2 SCTs and the 41.719% uses 3 of them. There is a 2.206% of the domains reached that do not even embed a single SCT in their certificate. Then, the 0.392% inserts in the certificate 4 SCTs; the 0.009% embeds 5 SCTs; the 0.030% uses 6 of them; another 0.009% goes for 7 SCTs; 8 SCTs in the 0.002% of the certificates and a last 0.009% uses the maximum number of SCTs found during this analysis, 9 SCTs.

The domains *you-fm.de*, *smokehouse.com*, *ingenia.org.uk*, *truecolorsintl.com* and *meformeren-region.fr* contain 2 SCTs in their certificate which do not point to any of the trusted log servers.

In Table 6.19 there is a more detailed usage of the SCTs in the server certificates. Each of them is a log server considered trusted by the Chromium Project[26].

Log server	No. of SCT in certificates
Google Argon2022 log	64,056
Google Argon2023 log	326,431
Google Argon2024 log	0
Google Xenon2022 log	77,134
Google Xenon2023 log	207,001
Google Xenon2024 log	0
Google Icarus log	0
Google Pilot log	2
Google Rocketeer log	4
Google Skydiver log	2
Cloudflare Nimbus2022 Log	76,465
Cloudflare Nimbus2023 Log	255,254
Cloudflare Nimbus2024 Log	0
DigiCert Log Server	188
DigiCert Log Server 2	2
DigiCert Yeti2022 Log	34
DigiCert Yeti2023 Log	204,593
DigiCert Yeti2024 Log	0
DigiCert Yeti2025 Log	0
DigiCert Nessie2022 Log	7,688
DigiCert Nessie2023 Log	158,184
DigiCert Nessie2024 Log	0
DigiCert Nessie2025 Log	0
DigiCert Yeti2022-2 Log	534
Sectigo Sabre CT log	14,507
Sectigo Mammoth CT log	53,988
Let's Encrypt Oak2022 log	52,692
Let's Encrypt Oak2023 log	236,800
Let's Encrypt Oak2024H1 log	0

Let's Encrypt Oak2024H2 log	0
Trust Asia Log2022	15
Trust Asia Log2023	146

Table 6.19: Number of SCTs every trusted CT log server.

6.5 Discussion and comparison with related work

Desktop web-browsers and mobile web-browsers follow the CA/Browser Forum suggestions in almost all the cases. Meanwhile, in 2018, a long term certificate and a NULL value in the Subject Common Name was accepted by the browsers[1], now such certificates get rejected in most of the cases. Mozilla Firefox, desktop version, is still the only one that perform classic OCSP queries. Almost all of the revocation status tests resulted in a successful connection. That probably is caused by the fact that tests' root CA certificates are installed at client-side. In fact, even if not receiving any information about where to collect revocation status data, the certificate passed the validation process since it has been issued by a trusted CA of the RP. This is an index of how much dangerous could be if an attacker is able to install a root CA certificate in the trusted store of the victim.

TLS interception products still showed a lack of support for classic revocation status check techniques[2]. Noticeable is the fact that, actually, ESET Smart Security was able to recognize a revoked certificate or a certificate issued by a revoked CA certificate, but only when the server hosting the website was Lighttpd or Apache. During Ahmad Samer Wazan et al. analysis ESET was not able to do so. On the other hand, the behavior of Mitmproxy, Squid and Kaspersky seems to have remained the same since then.

The tests performed in this work showed that OCSP Must-Staple in 2022 stayed unused, confirming the result obtained by Ahmad Samer Wazan et al. [2] in 2019. The presence of the TLS Feature in the certificates is still of the 0%. For what concerns OCSP Stapling, a slightly increase of support is noticed. It went from the 19% in 2018 to the 27% in 2019 to about the 31.5% in 2022. In the past, Ahmad Samer Wazan et al. supposed that the abandonment of OCSP support by PKIs. Results obtained by this work analysis in 2022 against the 718,283 domains of the Majestic Top-1-Million, showed, instead, that the OCSP Stapling support is still growing.

Moreover, in August 2018, from the work of Taejoong Chung et al. [11], OCSP Stapling was adopted in about the 38% of the domains in the Censys Alexa Top-1-Million.

Nowadays the most adopted TLS version is 1.3. There is still a 26.065% of the domains reached that still uses TLS 1.2. Only 5 years ago, TLS 1.3 was used rarely[5] and almost the totality of the web-servers were configured to use TLS 1.2. This thesis' results remark the always higher importance of security for the community.

From the analysis of CT usage in the Majestic Top-1-Million domains, it appears clear that there is still a concentration of trust. There is no more a log server managed by Symantec[38], which detained the major number of SCTs embedded in the certificates in 2017. In 2022, are Google, DigiCert, Cloudflare and Let'Encrypt the most used log providers. Google has the majority of timestamps in its logs, with a total of 674,630 SCTs embedded in the 718,283 certificates obtained. It is positive that none of the certificates retrieved contained a single SCT as happened in the past[5]. Almost the 97% of them uses from 2 to 3 SCTs in their certificates, which is a good news since with a cross-verification of the information obtained by different log servers the trustworthiness of the certificate increases.

Chapter 7

Conclusions

The thesis gives an insight of the actual behavior of browsers and TLS interception products receiving deliberately malformed certificates with respect to the state-of the-art within a testing environment or an enterprise network. Moreover, it measures the TLS versions negotiated, OCSP Stapling and Must-Staple adoption and Certificate Transparency (CT) usage in the actual top 1 million domains.

Tests that have been automatically generated to fulfill the first objective showed that there are still some inconsistencies in how the different TLS clients respond to wrongly configured certificates. This inconsistency is less evident in browsers which follow in most of the cases the suggestions of the CA/Browser Forum.

The most problematic step in the validation process seems to be still the revocation status check of the certificate. The clients mainly rely on OCSP Stapling to verify that the certificate is not revoked. If a server do not provide the status response, as happens for Nginx, the validation of the certificate is successful, even if the certificate is revoked. This thesis' tests evidence, actually, the fact that the absence of revocation status information does not affect the validation process in the majority of the browsers and TLS interception products. This is to be considered keeping in mind that the root CA certificates are installed on the target machines and can bias the validation process.

Surely, a step forward has been made by ESET Smart Security by adding the support for OCSP Stapling to its certificate validation process. Among all the TLS interception products tested in this work it has been the only one able to properly reject a connection towards a server whose certificate is revoked or has been issued by a revoked CA certificate.

More in general, each TLS interception product still follows its own set of rules for the certificate validation process. The same interception product's choices may be different even among different OS versions of it, as shown by Mitmproxy on macOS.

The statistical analysis of the Majestic Top-1-Million showed how the community is always more security-oriented. In 2022 almost the 74% of the major domains chooses to use TLS 1.3, more secure and fast with respect to the previous TLS 1.2, leading version in 2017.

Unexpectedly, only the 31% of them supports OCSP Stapling. The adoption of the protocol has seen a growth, but less rapid with respect to the one between 2016 and 2018 observed by Taejoong Chung et al.[11]. Such a situation is not so secure. It is true that now there are CRLSets, sub-sets of CRLs periodically pushed to the browsers, but they contain just revoked certificate due to some incident. They are not always sufficient to guarantee that the certificate is not revoked.

Moreover, only Mozilla Firefox allows the user to perform OCSP queries and that is the reason why even when Nginx has not sent the OCSP stapled response it was able to block the TLS connection anyway. It is desirable, in the future, to observe a major adoption of OCSP Stapling by web-domains as well as an increased support of it in TLS interception products.

OCSP Must-Staple is not used by the totality of the domains. The reason why that happens is maybe the fact that it is required that all the CAs issue their certificates adding an additional extension and it will require that all the servers that are going to use that certificate must support OCSP Stapling. Furthermore, since the OCSP responder may be unreachable due to a network problem, the TLS connection with the server may fail.

The usage of CT has seen an evolution with respect to the past 4 years. First of all, there are no more certificates with just one Signed Certificate Timestamp (SCT) embedded it them. Having just one log server to refer to when performing an auditor verification could introduce a single point of failure. The 55.623% and the 41.719% of the 728,183 certificates retrieved from the Majestic Top-1-Million domains contained respectively 2 and 3 SCTs. There is a 2.206% of web-servers that do not rely on SCT embedded in the certificate. These domains may rely on the provision of SCTs via TLS extension, more privacy-aware, or via OCSP Stapled response.

The analysis also showed a concentration of trust on 4 of the 6 recognized CT log server providers. The most of the certificates presented timestamps pointing to log servers maintained by Google, DigiCert[29], Cloudflare[27] and Let's Encrypt[30]. Less used are log servers provided by Sectigo[31] and just 161 of the domains reached contained a TrustAsia[28] SCT. This concentration of trust on those log providers is probably the cause of the reduction of the number of CT log providers with respect to the past. In 2020 they were 12, other than the 6 already above cited, there were also Certly[46], Izenpe[47], WhoSign, Venafi[48], CNNIC[49], and StartCom[50]. It is probable that a further reduction of the number of providers will be observed in few years.

As suggestion for future works, it is possible to extend the software realized to generates intentionally wrong certificates and automatic server configurations by including tests on certificate policies. Such policies may introduce different results in different browsers. It could be also interesting to verify how TLS interception products react to them.

It is also suggested to run the same tests against different TLS interception products or browsers. In this way, will be possible to collect more data and produce a more precise picture of the actual behavior of TLS clients.

In addition, other TLS interception techniques may be analyzed, such as Enterprise Transport Security (ETS), also known as eTLS. ETS is a variant of TLS 1.3 standardized by ETSI and combines TLS session splitting, server impersonation (sharing the private key of the server) and static DH keys on firewalls and/or enterprise servers to passively decipher the HTTPS traffic within an enterprise network. It could be interesting to compare this solution with the simple TLS session splitting techniques studied in this thesis, trying to define what are the advantages or disadvantages of one solution with respect to the other.

Finally, as already suggested by Ahmad Samer Wazan et al. [2], could be interesting to analyze how Content Delivery Netowrks (CDNs) validate certificates. Fetch the content from the cache of CDNs instead of getting data directly from the web-server is a well spread solution nowadays. CDNs act as TLS interception proxy collecting and caching the website data intercepting the TLS traffic of the users in order to provide it to other users. In this way they reduce the bandwidth costs and can provide the website content with an higher level of security and availability.

Bibliography

[1] D.McLuskie and X.Belleken, "X.509 Certificate Error Testing", ARES 2018: Preoceedings of the 13th International Conference on Availability, Reliability and Security, August 2018, pp. 1–8, DOI 10.1145/3230833.3232820

[2] A. S. Wazan, R. Laborde, D. W. Chadwick, R. Venant, A. Benzekri, E. Billoir, and O. Alfandi, "On the validation of web x.509 certificates by tls interception products", IEEE Transactions on Dependable and Secure Computing, vol. 19, no. 1, 2022, pp. 227–242, DOI 10.1109/TDSC.2020.3000595

[3] L. Waked, M. Mannan, and A. Youssef, "To intercept or not to intercept: Analyzing tls interception in network appliances", Proceedings of the 2018 on Asia Conference on Computer and Communications Security, New York, NY, USA, 2018, pp. 399–412, DOI 10.1145/3196494.3196528

[4] Y. Liu, W. Tome, L. Zhang, D. Choffnes, D. Levin, B. Maggs, A. Mislove, A. Schulman, and C. Wilson, "An end-to-end measurement of certificate revocation in the web's pki", Proceedings of the 2015 Internet Measurement Conference, New York, NY, USA, 2015, pp. 183–196, DOI 10.1145/2815675.2815685

[5] J. Amann, O. Gasser, Q. Scheitle, L. Brent, G. Carle, and R. Holz, "Mission accomplished? https security after diginotar", Proceedings of the 2017 Internet Measurement Conference, New York, NY, USA, 2017, pp. 325–340, DOI 10.1145/3131365.3131401

[6] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile." RFC-5280, May 2008, DOI 10.17487/RFC5280

[7] CA/Browser Forum, https://cabforum.org/wp-content/uploads/CA-Browser-Forum-BR-1.8.4.pdf

[8] R. Housley, T. Polk, D. W. S. Ford, and D. Solo, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile." RFC 3280, May 2002, DOI 10.17487/RFC3280

[9] Dennis Fischer, "Final Report on DigiNotar Hack Shows Total Compromise of CA Servers", October 2012, https://threatpost.com/final-report-diginotar-hack-shows-total-compromise-ca-servers-103112/77170/

[10] Certificate Transparency, https://certificate.transparency.dev/

[11] T. Chung, J. Lok, B. Chandrasekaran, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, J. Rula, N. Sullivan, and C. Wilson, "Is the web ready for ocsp must-staple?", Proceedings of the Internet Measurement Conference 2018, New York, NY, USA, 2018, pp. 105–118, DOI 10.1145/3278532.3278543

[12] Majestic Million, https://it.majestic.com/reports/majestic-million

[13] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile." RFC-5280, May 2008, DOI 10.17487/RFC5280

[14] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and D. C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP." RFC 6960, June 2013, DOI 10.17487/RFC6960
[15] The Chromium Project, https://www.chromium.org/Home/chromium-security/crlsets/
[16] Mark Goodwin, "Revoking Intermediate Certificates: Introducing OneCRL", March 2015, https://blog.mozilla.org/security/2015/03/03/revoking-intermediate-certificates-introducing-onecrl/
[17] D. E. E. 3rd, "Transport Layer Security (TLS) Extensions: Extension Definitions." RFC 6066, January 2011, DOI 10.17487/RFC6066
[18] Y. N. Pettersen, "The Transport Layer Security (TLS) Multiple Certificate Status Request Extension." RFC 6961, June 2013, DOI 10.17487/RFC6961
[19] P. Hallam-Baker, "X.509v3 Transport Layer Security (TLS) Feature Extension." RFC 7633, October 2015, DOI 10.17487/RFC7633
[20] B. Laurie, A. Langley, and E. Kasper, "Certificate Transparency." RFC 6962, June 2013, DOI 10.17487/RFC6962
[21] Alin Tomescu, "What is a Merkle Tree?", December 2020, https://decentralizedthoughts.github.io/2020-12-22-what-is-a-merkle-tree/
[22] DigiCert, https://www.digicert.com/faq/certificate-transparency/how-it-works.htm
[23] The Chromium project, https://chromium.googlesource.com/chromium/src/+/master/net/docs/certificate-transparency.md
[24] Apple, "Apple's Certificate Transparency policy", March 2021, https://support.apple.com/en-us/HT205280
[25] MDN Web Docs, November 2022, https://developer.mozilla.org/en-US/docs/Web/Security/Certificate_Transparency
[26] The Chromium project, https://googlechrome.github.io/CertificateTransparency/log_list.html
[27] Cloudflare, https://www.cloudflare.com/
[28] TrustAsia, https://www.trustasia.com/
[29] Digicert, https://www.digicert.com/
[30] Let's Encrypt, https://letsencrypt.org/
[31] Sectigo, https://sectigo.com/
[32] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3." RFC 8446, August 2018, DOI 10.17487/RFC8446
[33] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2." RFC-5246, August 2008, DOI 10.17487/RFC5246
[34] https://medium.com/@vanrijn/what-is-new-with-tls-1-3-e991df2caaac/
[35] N. Naziridis, "Tls 1.3 is here to stay", October 2018, https://www.ssl.com/article/tls-1-3-is-here-to-stay/
[36] H. Böck, J. Somorovsky, and C. Young, "Return of bleichenbacher's oracle threat (robot)", Proceedings of the 27th USENIX Conference on Security Symposium, USA, 2018, pp. 817–832
[37] Rapid7, https://www.rapid7.com/
[38] Symantec, https://securitycloud.symantec.com/cc/landing
[39] Censys, https://censys.io/
[40] X. d. C. de Carnavalet and P. C. van Oorschot, "A survey and analysis of tls interception mechanisms and motivations", 2020, DOI 10.48550/ARXIV.2010.16388
[41] SSL.com, https://www.ssl.com/article/browsers-and-certificate-validation
[42] PyOpenSSL Documentation, https://www.pyopenssl.org/en/latest/
[43] Cryptography Documentation, https://cryptography.io/en/latest/

[44] The OpenSSL project, http://www.openssl.org/
[45] OpenSSL PKI tutorial, https://pki-tutorial.readthedocs.io/en/latest/cadb.html#index-file
[46] Certly, https://www.certly.co/
[47] Izenpe, https://www.izenpe.eus/webize00-inicio/es
[48] Venafi, https://www.venafi.com/
[49] CNNIC, https://www.cnnic.com.cn/
[50] StartCom CA, https://www.startcomca.com/

Appendix A

User's Manual

A.1 TLS interceptor products installation

A.1.1 Mitmproxy

macOS

Type in terminal the following command. Note: you must install homebrew first in order to install mitmproxy on a MacBook.

```
brew install mitmproxy
```

In order to install the tool certificate at client-side to make *Mitmproxy* able to certify websites, open a terminal and run:

```
mitmproxy
```

Next, go to https://mitm.it after having set up the web-browser or the machine to use the interceptor as proxy. It is possible to do that in the web-browser settings or machine network settings. Simply insert IP address of the machine the interceptor it is installed on and the port on which it is listening (by default it is 8080).

If everything has been done correctly, the web-page can be displayed on the client machine. Now download the certificate according to the Operating System and follow the instructions reported in the web-page.

To make *Mitmproxy* able to validate the Certification Authority root certificate that has been created in test section, download the latest Mozilla bundle from https://curl.se/docs/caextract.html. Then use this command within the download folder:

```
cat -v <rootCA.pem> >> bundle.pem
```

This will create a bundle that contains both the real and the fake root certificates used for certificate validation. To launch *Mitmproxy* using the bundle type the following command:

```
mitmproxy --set ssl_verify_upstream_trusted_ca=path_to_bundle/bundle.pem
```

It is also possible to set it later using the command *O* once the interceptor is running. You need just to set the proper path to the bundle in the same option field as above.

Windows

Go to https://mitmproxy.org/ and download the installer, then install it. Open a Powershell and run the following command:

```
mitmproxy
```

Next, go to https://mitm.it after having set up the web-browser or the machine to use the interceptor as proxy. It is possible to do that in the web-browser settings or machine network settings. Simply insert IP address of the machine the interceptor it is installed on and the port on which it is listening (by default it is 8080).

If everything has been done correctly, the web-page can be displayed on the client machine. Now download the certificate according to the Operating System and follow the instructions reported in the web-page.

To make *Mitmproxy* able to validate the Certification Authority root certificate that has been created in test section, use the same certificate bundle created in macOS settings.

To launch *Mitmproxy* using the bundle type the following command into a Powershell:

```
mitmproxy --set ssl_verify_upstream_trusted_ca=path_to_bundle/bundle.pem
```

Ubuntu

Go to https://mitmproxy.org/ and downlaod the binaries. open a terminal in the folder where the binaries have been downloaded. Then type the command

```
./mitmproxy
```

Next, go to https://mitm.it after having set up the web-browser or the machine to use the interceptor as proxy. It is possible to do that in the web-browser settings or machine network settings. Simply insert IP address of the machine the interceptor it is installed on and the port on which it is listening (by default it is 8080).

If everything has been done correctly, the web-page can be displayed on the client machine. Now download the certificate according to the Operating System and follow the instructions reported in the web-page.

To make *Mitmproxy* able to validate the Certification Authority root certificate that has been created in test section, use the same certificate bundle created in macOS settings.

To launch *Mitmproxy* using the bundle type the following command within the binaries download folder:

```
./mitmproxy --set ssl_verify_upstream_trusted_ca=path_to_bundle/bundle.pem
```

A.1.2 Squid

macOS

Donwload Squid using Homebrew. The latest stable version available is 4.17.

```
brew install squid
```

Generate the *Squid* self-signed CA certificate to be installed at client-side. It can be used for all the different OSs.

```
openssl req -new -newkey rsa:2048 -sha256 -days 365 -nodes -x509 -extensions
    v3_ca -keyout squidCA.pem -out squidCA.pem
```

Edit the Squid configuration file in */opt/homebrew/etc/squid.conf* with root privileges modifying the *http_port* command as follow:

```
http_port 3128 ssl-bump cert=<path_to_certificate/squidCA.pem>
    generate-host-certificates=on dynamic_cert_mem_cache_size=0MB
```

Insert the certificate in a folder with these permissions:

```
chmod 700 <cert_folder>
```

In this way we say to squid to intercept SSL communications and use that certificate to sign the one to be sent to the client. Moreover we decided to generate dynamically those certificates and to not use cache. This last option for test mutters. Then add the following lines:

```
sslcrtd_program /opt/homebrew/opt/squid/libexec/security_file_certgen -s
    /opt/homebrew/var/cache/squid/ssl-db -M 4MB

sslcrtd_children 5

ssl_bump server-first all

sslproxy_cert_error deny all

always_direct allow all

tls_outgoing_options cafile=path_to_cert_upstream_check/bundle.pem
```

Those lines are needed to generate dynamically the certificates with *sslcrtd_program* and *sslcrtd_children* (maximum number of concurrent processes for certificate generation) options. *server-first* option says to negotiate first the connection between the server and the interceptor. *sslproxy_cert_error* is used to proxy to the client eventually the errors occurred during connection as a warning message. In this case it has been deactivated. *always_direct* is used to make Squid contact directly the server. *tls_outgoing_options* specifies which bundle has to be used when performing up-stream certificate verification. Insert here the same bundle generated for Mitmproxy. Before to start squid run the following command to initialize the SSL-DB and cache as root. To avoid permission problems change them the owner of both with:

```
sudo chmod -R 700 <path_to_folder>
sudo chown -R nobody <path_to_folder>
```

Then:

```
/opt/homebrew/opt/squid/libexec/security_file_certgen -c -s
    /opt/homebrew/var/cache/squid/ssl-db -M 4MB
squid -z
```

To run Squid:

```
sudo squid -d 10
```

To stop it:

```
sudo kill -9 <PID>
```

The PID is shown in one of the first lines of the output once the interceptor is running.

Windows

Go to https://squid.diladele.com and download *Squid for Windows* msi file. Then follow the installation guide to install it. Use the same certificate generated in macOS sub-section.

Edit the Squid configuration file in *path_to_installation_folder/Squid/etc/squid/squid.conf* modifying the *http_port* command as follow:

```
http_port 3128 ssl-bump cert=<path_to_certificate/squidCA.pem>
    generate-host-certificates=on dynamic_cert_mem_cache_size=0MB
```

Check the macOS sub-section to see what those options do. Then add the following lines:

```
sslcrtd_program /cygdrive/c/squid/lib/squid/security_file_certgen -s
    /cygdrive/c/squid/var/cache/ssl-db -M 4MB

sslcrtd_children 5

ssl_bump server-first all

sslproxy_cert_error deny all

always_direct allow all

tls_outgoing_options cafile=path_to_cert_upstream_check/bundle.pem
```

Check the macOS sub-section to see what those options do.

The bundle must contain the real trusted CA certificates bundle plus those generated by the tests.

To run *Squid*, open *cmd* as Administrator and type:

```
net start squidsrv
```

To stop it:

```
net stop squidsrv
```

Ubuntu

Download the latest stable version (5.5) from the official website and build it with the *with-default-user-proxy*, *with-openssl*, *enable-ssl-crtd* options:

```
mkdir squid
cd squid
wget http://www.squid-cache.org/Versions/v5/squid-5.5.tar.gz
tar -zxfv squid-5.5.tar.gz
```

```
cd squid-5.5
sudo ./configure --with-default-user-proxy --with-openssl --enable-ssl-crtd
sudo make
sudo make install
```

Use the same certificate created in macOS sub-section.

Edit the Squid configuration file in */usr/local/squid/etc/squid.conf* with root privileges modifying the *http_port* command as follow:

```
http_port 3128 ssl-bump cert=<path_to_certificate/squidCA.pem>
    generate-host-certificates=on dynamic_cert_mem_cache_size=0MB
```

Insert the certificate in a folder with these permissions:

```
chmod 700 <cert_folder>
```

In this way we say to squid to intercept SSL communications and use that certificate to sign the one to be sent to the client. Moreover we decided to generate dynamically those certificates and to not use cache. This last option for test mutters. Then add the following lines:

```
sslcrtd_program /usr/local/squid/libexec/security_file_certgen -s
    /usr/local/squid/var/cache/squid/ssl-db -M 4MB

sslcrtd_children 5

ssl_bump server-first all

sslproxy_cert_error deny all

always_direct allow all

tls_outgoing_options cafile=path_to_cert_upstream_check/bundle.pem
```

Check the macOs sub-section to see what those options do.

Before to start squid run the following command to initialize the SSL-DB and cache as root. To avoid permission problems change them the owner of both with:

```
sudo chmod -R 700 <path_to_folder>
sudo chown -R nobody <path_to_folder>
```

Then:

```
/usr/local/squid/libexec/security_file_certgen -c -s
    /usr/local/squid/var/cache/squid/ssl-db -M 4MB

/usr/local/squid/sbin/squid -z
```

To run Squid:

```
sudo /usr/local/squid/sbin/squid -d 10
```

To stop it:

```
sudo kill <PID>
```

The PID is shown in one of the first lines of the output once the interceptor is running.

A.1.3 Kaspersky Total Security

Go to https://www.kaspersky.it/downloads/total-security-free-trial and click on the red *Try for free* button. Double click on the setup file and follow the instructions to install it.

A.1.4 ESET Smart Security

Go to https://www.eset.com/int/home/free-trial/ and click on *Try for free* button under ESET Smart Security. Double click on the setup file and follow the instructions to install it.

A.2 Server installation

A.2.1 Apache2

To install *Apache 2.4* and configure it to use TLS, run the following commands in order:

```
sudo apt-get update
sudo apt-get install apache2
sudo a2ensite default-ssl.conf
sudo a2enmod ssl
sudo service apache2 restart
```

To start the server with a test-specific configuration use the following commands:

```
sudo apachectl -f <absolute_path_to_config/httpd.conf>
```

To stop it run this or an equivalent command:

```
sudo killall apache2
```

A.2.2 Nginx

Run the following commands to install *Nginx*:

```
sudo apt update
sudo apt install nginx
```

In order to allow *Nginx* traffic through the firewall on both port 80 and 443, this command have to be run:

```
sudo ufw allow 'Nginx␣Full'
```

To start the server with a test-specific configuration use the following commands:

```
sudo nginx -c <absolute_path_to_config/nginx.conf>
```

To stop it run this or an equivalent command:

```
sudo killall nginx
```

A.2.3 Lighttpd

Prerequisites:

- autoconf
- automake
- libtool
- m4
- pcre-devel / libpcre3-dev
- pkg-config
- OpenSSL
- libssl-dev

Install all the dependencies listed before.

Download source files from: https://www.lighttpd.net/2022/8/7/1.4.66/.

Then:

```
cd lighttpd1.4.66
sudo ./autogen.sh
sudo ./configure -C --prefix=/usr/local --with-openssl # ./configure --help
    for additional options
sudo make -j 4
sudo make check
sudo make install
```

To start the server with a test-specific configuration use the following commands:

```
sudo lighttpd -f <absolute_path_to_config/lighttpd.conf>
```

To stop it run this or an equivalent command:

```
sudo killall lighttpd
```

To generate the OCSP stapled response:

```
openssl ocsp -issuer [CHAIN_PEM] -cert [CERT_PEM] -respout [OCSP_RESP]
    -noverify -no_nonce -url [OCSP_URI]
```

Appendix B

Developer's Reference Guide

B.1 Test generation script

B.1.1 Dependencies

In order to make the script work properly it is need to install the following modules in python:

```
python3 -m pip install pyopenssl
python3 -m pip install cryptography
```

The script will automatically generate the configuration files for Nginx, Apache and Lighttpd. In order to launch them it is needed to install the servers as shown in A.2.

B.1.2 Manual

To start generate tests and their Apache, Nginx, Lighttpd configuration run with superuser privileges the file `run.sh`.

This file will clear the already existing test generated, if any. It will create the proper destination folders in /tmp and /var/www/html if not presents. After that will run the python script test_generation.py. Listing B.1 shows the content of this file.

```
1  #! /bin/bash
2
3  echo 'Clearing␣environment...'
4
5  mkdir -p /var/www/html/download
6
7  if [[ -d certs ]]
8  then
9          rm -R certs
10         rm -R /var/www/html/download
11 fi
12
13 mkdir -p /var/www/html/download
```

```
14 mkdir -p /tmp/lighttpd
15
16 if [[ -d tests ]]
17 then
18         rm -R tests
19 fi
20
21 echo 'Starting␣test␣generation...'
22 sudo /bin/python3 $PWD/test_generation.py
```

Listing B.1. Bash script to run the test generation process (Source: run.sh).

`test_generation.py` is made up a loop iterating over the configuration read by the file `configuration.yml` present in the `config/` folder (Listing B.2).

```
28     with open('./config/configuration.yml', 'rt') as f:
29         config = yaml.full_load(f)
30
31     if not os.path.exists(BASE_PATH):
32         os.mkdir(BASE_PATH)
33
34     for item, doc in config.items():
35         print(item, ':', doc, end='␣-->␣')
36         path = os.path.join(BASE_PATH, item)
37         if not os.path.exists(path):
38             os.mkdir(path)
39             os.mkdir(os.path.join(path, 'certificates'))
40             os.mkdir(os.path.join(path, ROOT_CA_PATH))
41             os.mkdir(os.path.join(path, INTER_CA_PATH))
42             os.mkdir(os.path.join(path, LEAF_CRT_PATH))
43             os.mkdir(os.path.join(path, APACHE_PATH))
44             os.mkdir(os.path.join(path, NGINX_PATH))
45             os.mkdir(os.path.join(path, LIGHTTPD_PATH))
46
47         profile = profileGeneration(doc.get('profile'))
```

Listing B.2. Code section with reading of configuration and forlders creation per profile (Source: test_generation.py).

The profile is retrieved thanks to a function in a second script named `profile_generation.py`: `profileGeneration()`. This second script imports all the test classes definitions contained in the `profiles.py`. Depending on the profile name read by the configuration, the function, calls the proper constructor and return the newly created profile class instance.

It is possible to extend the profiles' class definition properly adding new definitions in files `profiles.py` and `profile_generation.py`. The place where it is possible to insert new code is signaled with comments as Listing B.3 and Listing B.4.

Lastly, it is possible to customize the servers' configurations modifying properly the functions in `server_config.py` file.

```
2614                         x509.AccessDescription(
2615                         AuthorityInformationAccessOID.OCSP,
2616                         x509.UniformResourceIdentifier(u'http://ocsp.interCA
    :8081')
```

```
                    )
                        ]), False)
            ] )
        print('SUCCESS!', end='')

# Insert here the definition of your test classes
# Documentation of PyOpenSSL: https://www.pyopenssl.org/en/latest/
# Documentation of cryptography: https://cryptography.io/en/latest/
```

Listing B.3. Code section where to insert new profiles' class definitions (Source: profiles.py).

```
    if (profile == 'CN-SAN_pubIP'): ret = CN_SAN_pubIP()
    if (profile == 'NullCN-SAN_DNS_and_pubIP'): ret = NullCN_SAN_DNS_and_pubIP
    ()
    if (profile == 'NullCN-SAN_nullDNS_and_pubIP'): ret =
    NullCN_SAN_nullDNS_and_pubIP()

    # Insert here your new profiles classes
    #
    # if (profile == [your configuration profile name]): ret =
    Your_Profile_Class()


    return ret
```

Listing B.4. Code section where to insert new profiles' class creations (Source: profile_generation.py).

The zip folder of this script contains also the files that goes automatically in the `/var/www/html/download` folder within the `crls_crts` folder.

All the certificates generated use 192.168.0.106 as private IP address, if the machine on which the test are generated is not configured to use this IP address, modify it in the `profiles.py` file. It is possible to change the public DNS name and IP address as well.

Before to run the tests, some operation are required. First of all it is needed to install at client-side all root certificates generated by the script but `rootCA_2` one, which is used for *UnknownIssuer* test. It is needed also to create a CA bundle to be used for interceptor proxies as specified in Appendix A.1. In Ubuntu the CA certificates must be installed directly in the browsers stores, while in the other systems, in the majority of the cases, it is sufficient to install them in the system store. Mozilla Firefox requires to install the certificates in its store in any case on desktop devices. Mozilla Firefox Android version requires also to activate the secret settings (by tapping several times on the logo in the application information page of the browser settings) to allow it to use local certificates.

It is needed to insert the following entries in the clients and interceptor machines /etc/hosts file:

```
[IP address] www.mywebsite.com
[IP address] ocsp.interCA
[IP address] ocsp.rootCA
[IP address] ca.interCA
[IP address] ca.rootCA
```

In order to run the test server configuration see Appendix A.2.

To launch an OCSP server:

```
openssl -index certs/[CA]/demoCA/index.txt -rsigner
    certs/[CA]/OCSP/ocsp.crt -rkey certs/[CA]/OCSP/ocspSigning.key -CA
    certs/[CA]/demoCA/cacert.pem -port [port no.] -text &
```

To launch an HTTP server for the CA:

```
python3 -m http.server [port] --directory /var/www/html
```

The ports to be used for the OCSP servers and the CA stores are written within the test profile class definition.

To use stapling with Lighttpd it is required to cache manually the status of the certificate e place the output file called "stapling-cache" in `/tmp/lighttpd/`. The command to do so can be found in Lighttpd section of Appendix A.2.

To run the tests using the TLS interception proxies it is needed to configure the client to use a proxy. It is possible to do so from the browser settings or from the system network settings. The proxy must be configured with the IP address of the machine running it and the port specified in the proxy configuration.

To run the tests against iOS or Android it is possible to perform a DNS spoofing attack using Ettercap or an equivalent tool since there is no direct access to the hosts file as happen for desktop systems.

B.2 Top-1-Million analysis script

B.2.1 Dependencies

In order to make the script work properly it is need to install the following modules in python:

```
python3 -m pip install pygal
python3 -m pip install lxml
python3 -m pip install cairosvg
python3 -m pip install tinycss
python3 -m pip install cssselect
```

The Python version used to develop the script is Python 3.10, so in order to avoid compatibility problems install the same version.

B.2.2 Manual

The default behavior is to analyze a whole list of domain names from an input file, produce a CSV output file and automatically analyze the results and produce the charts.

The data collection process operates two different TLS connections. One through a `os.subprocess` command and a second using a secure socket. The first connection is performed with the OpenSSL `s_client` module, requiring the revocation status of the certificate. The second one is required to retrieve the TLS version negotiated. Listing B.5 contains the section of the function `analyze_domains()` in `tls_client.py` file responsible of these connections.

```
13      def analyze_domains(domains: list, outfile, start, end, plot=False):
14      num = start+1
15      dataOut = open(outfile+'.csv', "a")
16      for domain in domains[start:end]:
17          out = open("server_info.txt", "w")
18
19          res = subprocess.run('echo -n | timeout 3 openssl s_client -connect '+
        domain+':443 -status', shell=True, stdout=out)
20          # Extract PEM encoded server certificate
21          # if return code is 124 it means the command failed due to timeout
22          if res.returncode == 0:
23              commandRes = open("server_info.txt", "r")
24              data = commandRes.read()
25              subStart = '-----BEGIN CERTIFICATE-----'
26              subEnd = '-----END CERTIFICATE-----'
27
28              start = data.find(subStart)
29              end = data.find(subEnd)+len(subEnd)
30
31              with open('cert.pem', 'w') as cert:
32                  cert.write(data[start:end])
33
34              tls_v = ""
35              context = ssl.create_default_context()
36              try:
37                  with socket.create_connection((domain, 443)) as sock:
38                      sock.settimeout(3.0)
39                      with context.wrap_socket(sock, server_hostname=domain) as
        ssock:
40                          tls_v = ssock.version()
```

Listing B.5. Code section of analyze_domains() function (Source: tls_client.py).

The chart and the analysis of the results is performed by the function `plot_results()` which (Listing B.6):

- prints the max number of SCTs embedded within a single certificate, the number of domains unreached, the number of domains which used ivalid SCTs, the SCTs' counts per log server, the OCSP Stapling and OCSP Must-Staple percentage support;
- produces a bar chart for SCTs used per log providers, a bar chart for number of SCTs emebedded within a single certificate and a pie chart of the percentage of negotiation of different TLS versions.

```
197     # Printing max number of SCTs, number of domains unreached, the domains
        which used ivalid SCTs, SCT counts per log server, OCSP Stapling
        percentage support, OCSP Must-Staple percentage support
198     print('Max number of SCTs: ' + str(max))
199     print(str(not_reached_count)+' domains not reached over '+str(len(results)
        ))
200     print(str(invalid_sct_count)+' domains used invalid SCT in their
        certificate')
```

```
201     for i in range(0,len(logs),1):
202         print(str(list(logs.keys())[i]) + ": " + str(list(logs.values())[i]))
        # Printing single logs counts
203
204     per = (ocsp_stapling_count/(len(results)-not_reached_count))*100
205     f = float(f'{per:.3f}')
206     print(str(f)+'% of domains support OCSP Stapling')
207
208     per = (ocsp_must_staple_count/(len(results)-not_reached_count))*100
209     f = float(f'{per:.3f}')
210     print(str(f)+'% of domains support OCSP Must-Staple')
211
212     # Charts generation
213     providers_chart = pygal.Bar(human_readable=True, print_values=True, style=
        DefaultStyle(
214                 value_font_family='googlefont:Raleway',
215                 value_font_size=20,
216                 value_colors=('black',)))
217     providers_chart.title = 'CT log providers used in Top-1-M domains'
218     for i in range(0, len(providers), 1):
219         providers_chart.add(list(providers.keys())[i], list(providers.values()
        )[i])
220     providers_chart.render_to_file('log_providers.svg')
221     providers_chart.render_to_png(filename='log_providers.png')
222
223     sct_chart = pygal.Bar(print_values=True, print_zeroes=False,
        human_readable=True, style=DefaultStyle(
224                 value_font_family='googlefont:Raleway',
225                 value_font_size=20,
226                 value_colors=('black',)))
227     sct_chart.title = 'Number of SCTs used in Top-1-M domains (in %)'
228     for i in range(0, len(scts_counts), 1):
229         n = (list(scts_counts.values())[i])
230         if (n>0):
231             per = (n/(len(results)-not_reached_count))*100
232             f = float(f'{per:.3f}')
233             sct_chart.add(list(scts_counts.keys())[i], f)
234     sct_chart.render_to_file('log_scts.svg')
235     sct_chart.render_to_png(filename='log_scts.png')
236
237     tls_chart = pygal.Pie(print_values=True, print_zeroes=False,
        human_readable=True, style=DefaultStyle(
238                 value_font_family='googlefont:Raleway',
239                 value_font_size=20,
240                 value_colors=('black',)))
241     tls_chart.title = 'TLS version negotiated in Top-1-M domains (in %)'
242     for i in range(0, len(tls_versions), 1):
243         n = (list(tls_versions.values())[i])
244         if (n>0):
245             per = (n/(len(results)-not_reached_count))*100
```

```
246             f = float(f'{per:.3f}')
247             tls_chart.add(list(tls_versions.keys())[i], f)
248     tls_chart.render_to_file('tls_versions.svg')
249     tls_chart.render_to_png(filename='tls_versions.png')
```

Listing B.6. Code section of plot_results() function (Source: tls_client.py).

It allows also to analyze just a fraction of the domains, to do not analyze and plot the results or only analyze and plot results previously obtained. To run the script:

```
python3 tls_client.py [options]
```

The options are:

- **-in input_file**. It is used to specify the source file where to pick the domains name to contact. Optional. If present require the presence of '-out' option.
- **-out output_file**. It is used to specify the name of the output file. Mandatory if '-in' option is specified.
- **-start index**. It is used to specify where to start collecting the data in the input source file. Optional. If not present the domains are analyzed from the beginning.
- **-end index**. It is used to specify where to stop collecting the data in the input source file. Optional. If not present the domains are analyzed until the end.
- **-no_plot**. It is used to specify that the results obtained must not be analyzed and plotted. Optional.
- **-plot_only output_file**. Optional. If present must be used alone. It is used to specify a previous collected results file to be analyzed and plotted.