POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

SSI-aware TLS handshake in OpenSSL

Supervisor
prof. Antonio Lioy

Candidate
Leonardo Perugini

Internship Tutor
PhD. Andrea Vesco
PhD. Alberto Carelli

Academic Year 2022-23

A mia nonna Amelia

† A mio nonno Gian Mario

Summary

Self-Sovereign Identity (SSI) is a new digital identity paradigm that allows users to create and control their own identity, without relying on any centralised authority. A user can generate its own identity (Decentralized Identifiers) stored in a Distributed Ledger (DL) and associates authentic and demonstrable attributes to it (Verifiable Credentials). The Transport Layer Security (TLS) protocol version 1.3 is a client/server Internet protocol that allows two endpoints to communicate securely by providing authentication of the parties, confidentiality and integrity of the messages. In TLS identities are issued in the form of X.509 certificates by Certification Authorities (CAs), which are centralized entities who have full control on the certificates they emit. In this project I have designed an SSI-aware version of TLS providing an authentication mechanism that substitutes X.509 certificates with Decentralized Identifiers. Furthermore, to facilitate the transition to this new identity system I have also developed a hybrid TLS model to allow an endpoint identify itself with a Decentralized Identifier and still allow the other to make use of an X.509 certificate and vice-versa. I have implemented and tested the solution in OpenSSL, an open-source, widespread and multiplatform cryptographic library that supports TLS 1.3.

Acknowledgements

In the first place I would like to thank the professor Antonio Lioy for giving me the opportunity to work on this project. Then I would like to show my gratitude to Ing. Andrea Vesco, Ing. Alberto Carelli and Ing. Davide Margaria for the patience and genuineness they demonstrated throughout this journey. In the end I will forever be grateful for my family for their abundance in everything.

Contents

Chapter 1

Introduction

Digital identities are used over the Internet to prove who a user is to the websites, services, and apps in order to establish trusted relationships to access or to protect private information. The way digital identities are managed can be grouped under three different models, namely: centralized, federated, decentralized [1].

1.1 Centralized identity model

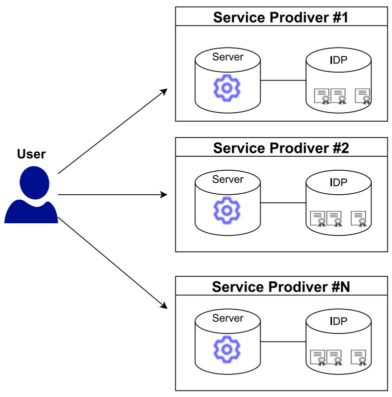


Figure 1.1: Centralized Identity Model.

The Centralized Identity model (Figure 1.1) is the first model to be proposed and the one that, most of the times is still employed today. In practice, a user accesses a website, a service or an application that manages the identity system. The user must create its own identity by registering an *account* (that usually consists of a tuple, composed of a username and a password) that will be stored in the databases of the service provider [2]. However, this approach presents several drawbacks:

- accounts are not portable, so users have to create one identity per service provider, with the responsibility of handling and remembering several different names and passwords [1];
- compromise user's privacy [1], for example by tracing user's online activities;
- the centralized databases that store all the users' credentials have been subject to some of the biggest data breaches in history [1].
- depending on the security and privacy policies of the service provider the quality of data associated to the user's account can be more or less reliable: heavily regulated sectors such as governments and financial services have accurate identity proofing process, instead of some social media platforms where creating multiple and fake identities can be easily accomplished [2].

1.2 Federated identity model

This model tries to overcome some of the problems brought up by the centralized approach, by inserting a service provider named *identity provider* (IDP) in between the user and the website, the service or the application is trying to access (Figure 1.2). This approach offers a degree of portability to the centralised identity model: the user now can register an account with a certain IDP that will provide some basic identity data to any site, service, or app that uses that IDP to log the user in. The collection of all the sites that exploit the same IDP is called a *federation*. Unfortunately, even this model presents some flaws [1]:

- there is not a single IDP compatible with all websites, services, and apps;
- even though a federation proposes some kind of portability, the identity provider still has full control over the user's data;
- as with centralized systems, the trust level may highly vary depending on the system owners and the identity verification degree that they perform [2];
- IDP can track user's login activity across multiple services;

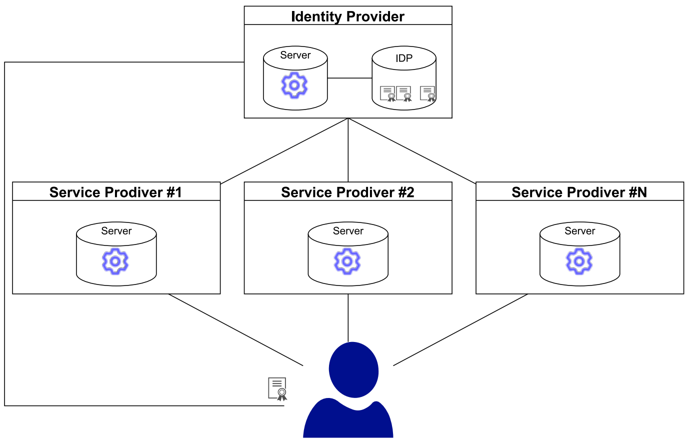


Figure 1.2: Federated Identity Model.

1.3 Decentralized identity model

In the two approaches addressed above the identity is provided by a third party. In the decentralized identity model the user has full control of their identity and decides how and under what circumstances their data should be shared with others. This technology has a completely different architecture compared to the other two models because it leverages the features of a peer-to-peer network, such as the **Distributed Ledger Technology** (DLT). The decentralized identification system consists of many nodes that can act as users, organizations, issuers, and validators. The **Self-sovereign identity** (SSI) model is an instance of this system [3].

In the SSI paradigm the user creates its own digital identity(ies), by creating one or several unique identifiers, named **Decentralized IDentifiers** (DIDs), and associating authentic and provable attributes to them. After this step, the user can request and obtain credentials from trusted parties and show them when it is necessary. For instance, a student can collect credentials from their university, and then can present the appropriate credential when applying for a student loan. This technology exploits *public key cryptography*, such as digital signatures, to prove the authenticity of a credential (for instance to inspect if it has been issued by the named authority and has not been manipulated or if the person who presents it is the same person being referred to). These digital credentials take the name of **Verifiable Credentials** (VCs) and they can be recognized by any service providers and website. However, the SSI will not completely get rid of intermediate parties, but it will still rely on them to sign data that will turn into verifiable credentials. Like physical credentials, the authority that issues a VC still has full control over it (e.g. the state can issue a driving license, and can also revoke it). For many use cases, relying on trusted third parties to issue VCs that can be associated with user-generated identifier would be desirable [2].

Self-Sovereign Identity is an emerging technology and the lack of standard implementations keeps it from being adopted, so the goal of my thesis is to provide a way to establsih a secure communication among two parties over the Internet employing one of the fundamental elements of this new paradigm: the Decentralized Identifiers.

Chapter 2

Self-sovereign identity model

2.1 Self-Sovereign Identity

Self-Sovereign Identity is an emerging digital identity management system where users should be able to create and control their own identity, without relying on any centralised authority. At the basis of this new paradigm there are three fundamental technologies: the *Distributed Ledger Technology* (DLT), the *Decentralized Identifier* (DID), and the *Verifiable Credential*(VC).

2.2 Distributed Ledger Technology

A Distributed Ledger Technology (DLT) or Distributed Ledger is a decentralized system that records events and allows all participants to interact without needing a trusted third party, even if they do not trust each other. It is based on three main components: *(i) public key cryptography*, which provides a secure digital identity for each participant, consisting of a pair of keys (one public, one private); *(ii) a distributed peer-to-peer network* that ensures all participants have a copy of the ledger and that transactions are propagated to everyone in the network, avoiding a single point of failure and preventing a single or small group of players to take over the network; and *(iii) consensus mechanisms* that ensure all participants agree on the validity and order of transactions. With their digital identity, participants can claim ownership over objects in the ledger [4].

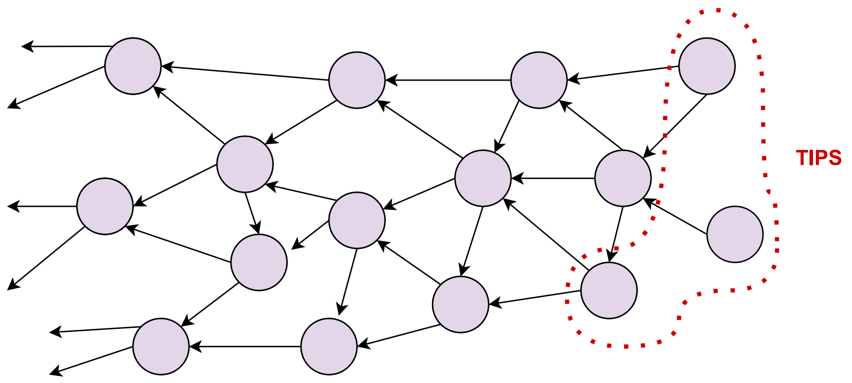


Figure 2.1: IOTA Tangle structure.

The IOTA Tangle is a type of DLT which has been created by the IOTA Foundation, a non-profit foundation. The Tangle consists of a DAG (*Directed Acyclic Graph*), i.e. a graph where nodes are directionally connected to each other but they never form a cycle. In the Tangle each node corresponds to a record and must be pointed by at least two other nodes (i.e. at least two incoming edges), but without ever forming a cycle. At the end of the Tangle there are nodes that do not have incoming edges which means that these records are unconfirmed. These nodes are called tips. When a new record must be inserted in the Tangle it randomly selects two tips, up to eight, and attaches itself to them becoming a new tip of the Tangle. In short, whenever a new record is added to the Tangle two records are being confirmed. The IOTA Tangle structure is illustrated in Figure 2.1

2.3 Decentralized Identifiers

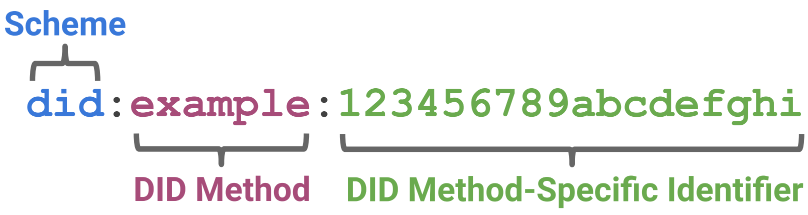

Scheme
did:example:123456789abcdefghi
DID Method
DID Method-Specific Identifier

Figure 2.2: Decentralized Identifier example (source: Decentralized Identifiers - W3C).

Decentralized identifiers (DIDs) are a new type of identifier that enables verifiable, decentralized digital identity. A DID is a URI (Figure 2.2) that consists of three parts [5]:

- the did URI scheme identifier;
- the identifier for the DID method;
- the DID method-specific identifier.

A DID is associated to a **DID document** (a JSON object stored in the DLT specified in the DID method of the DID) that contains information related to the DID, such as ways to cryptographically authenticate a DID controller or services to interact reliably with the DID subject [5].

The **DID subject** is the entity identified by the DID (e.g., a person, organization, thing, data model, abstract entity, etc..). The **DID controller** is the entity (person, organization, autonomous software) allowed to make changes to a DID document and can select which subject the DID refers to. Furthermore, a DID could have more than one controller and the DID subject can be the DID controller or can correspond to one of them. Differently from typical identifiers, DIDs no longer depend on centralised registries and identity providers enabling the controller of a DID to prove control over it without relying on any other party [5]. Figure 2.3 offers an overview of the relations between this components.

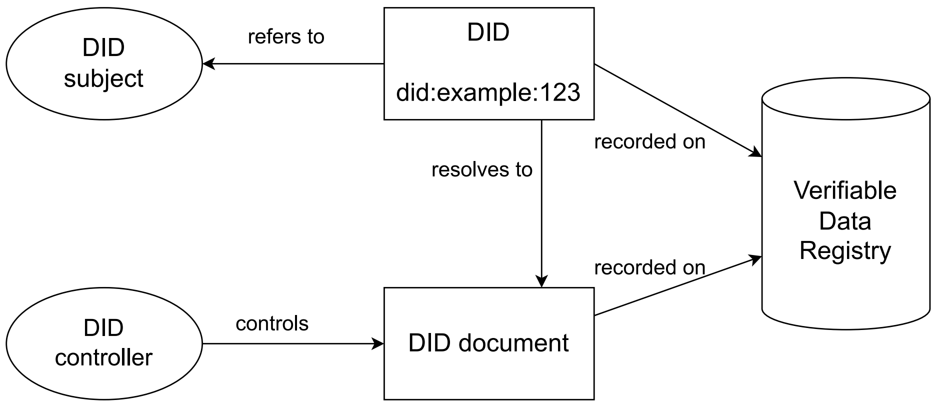


Figure 2.3: DID architecture

Figure 2.4 illustrates a sample of a DID document where the *context* field states which JSON representations the DID document follows, the *id* property expresses the DID subject (in this case corresponds to the DID controller) and the *authentication method* (to authenticate the controller) has various fields such as the name (id), the type, the controller and the value of the public key.

```
{
   "@context":[
      "https://www.w3.org/ns/did/v1",
      "https://w3id.org/security/suites/ed25519-2018/v1"
   ],
   "id": "did:example:123456789abcdefghi",
   "authenticationMethod": [
      {
         "id": "did.example:123456789abcdefghi#key-0",
         "type": "Ed25519VerificationKey2018",
         "controller": "did:example:123456789abcdefghi",
         "publicKeyBase58": "3M5RCDjPTWPkKSN3sxUmmMqHbmRPegYP1tjcKyrDbt9J"
      }
   ]
}
```

Figure 2.4: DID document example.

2.4 Verifiable Credentials

A credential in the physical world expresses information, such as the identification of the subject (photo, name, identification number, etc.), the issuer of the credential (government, national agency, etc.), the type of the credential (passport, driving license, etc.) and some other content. A **verifiable credential** (VC) is a machine-readable credential and can represent the same information a physical credential does, but it can also depict things that have no physical equivalent, such as the ownership of a bank account.

These new type of credentials are more tamper-evident and trustworthy than the classical ones, since with the introduction of public key cryptography, they are digitally signed by the issuer and can be verified cryptographically. Whenever the holder of a verifiable credential is requested to authenticate to access a service can always generates verifiable presentations and then share them with verifiers to verify that matches certain characteristics. This new credential model provides the additional feature of establishing trust at a distance in a very efficient and easy way as the transmission of verifiable credentials and verifiable presentations can be done rapidly, employing the network. [6]

The three main actors in the Verifiable Credentials ecosystem are (Figure 2.5):

- the **holder**: the entity who owns one or more verifiable credentials and then generate verifiable presentations starting from them. Students, employees, and customers could belong to this category;

- the **issuer**: its task is to make claims about one or more subjects (a claim is an assertion about an entity), creating a verifiable credential from these claims and transmitting the verifiable credential to the holder which will store it in a credential repository. Most of the times the holder of a VC is the subject, but there are cases in which this statement is not true, like a parent might hold the verifiable credentials of a child. This issuer role could be interpreted by corporations, non-profit organizations, trade associations, governments, and individuals;

- the **verifier**: processes the verifiable credentials received, optionally inside a verifiable presentation, from the holder. Example of verifiers include employers, security personnel, and websites. Among the verifier and the issuer there is implicit trust.

Technically a verifiable credential is a JSON object and consists of three main parts (Figure 2.6):

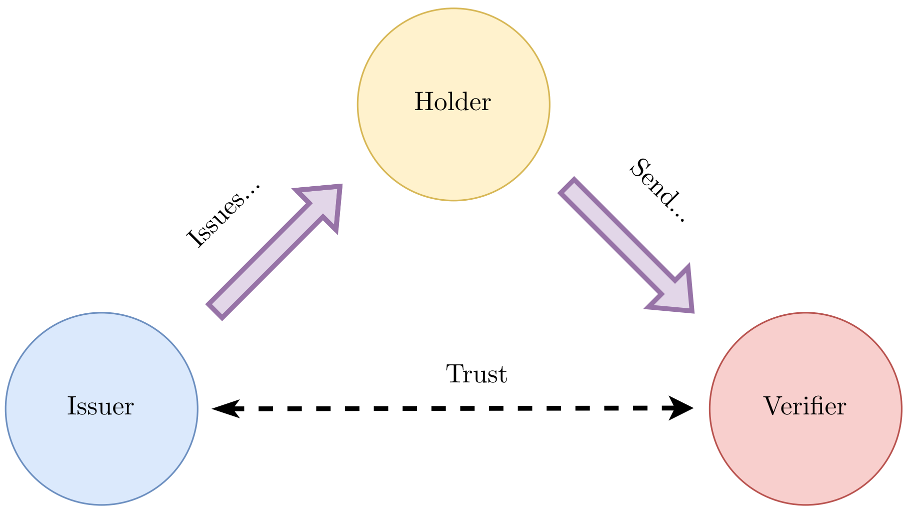


Figure 2.5: VC schema (source: Verifiable Credentials - W3C).

- some **metadata** to express properties of the credential, such as the issuer, the expiry date and time, a representative image, a public key to use for verification purposes, the revocation mechanism and so on;
- a set of one or more **claims** made by the same entity;
- **digital signature** (digital proof) of metadata and claims to cryptographically prove who issued them.

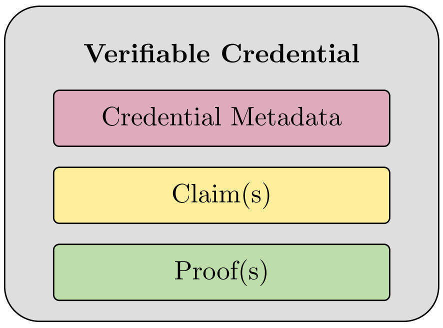


Figure 2.6: Verifiable Credential structure (source: Verifiable Credentials - W3C).

There are cases where a holder wants to protect its privacy and express only the strictly necessary information for a specific situation and this is when **Verifiable Presentations** (VP) come into play to represent a subset of one's persona. A verifiable presentation collects data from one or more verifiable credentials, and it is structured in such a way that the authorship of the data is verifiable. The data in a presentation is often about the same subject, but might have been issued by multiple issuers. The aggregation of this information typically expresses an aspect of a person, organization, or entity [6]. Similarly to verifiable credentials, three fundamental elements give shape to a verifiable presentation (Figure 2.7):

- the **presentation metadata**;
- one or more **verifiable credentials**;
- one or more **digital proofs**.

The fact that the attribute "verifiable" precedes the terms credential and presentation does not guarantee the veracity of the claims that have been made, but it is linked to the characteristic

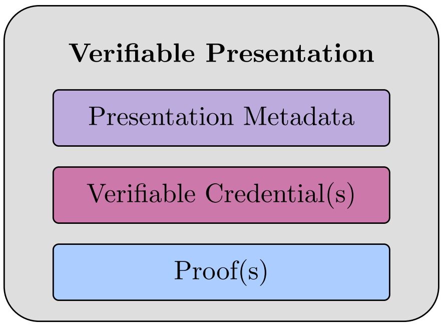


Figure 2.7: Verifiable Presentation structure (source: Verifiable Credentials - W3C).

of a credential or presentation as being able to be verified by a verifier; however, the issuer can include values in the evidence property to help the verifier apply their business logic to determine whether the claims have sufficient truthfulness for their needs.

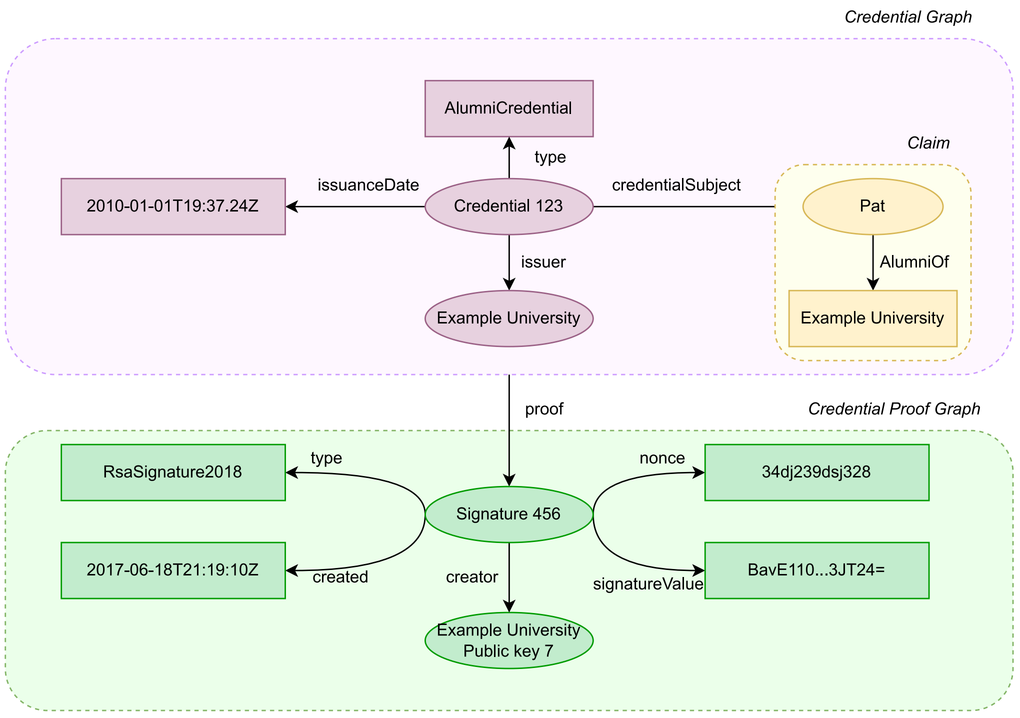


Figure 2.8: Pat's credential graph (source: Verifiable Credentials - W3C).

Let's consider an example to illustrate a typical use case scenario [6], were both credentials and presentations are employed, such as redeeming an alumni discount from a university. For instance Pat receives an alumni verifiable credential from a university, signed with the private key of the University authentication method, and stores it in a digital wallet (Figure 2.8 shows Pat's credential graph). Later, Pat will try to buy a ticket to access some sporting event. The ticket sales system declares that any alumni of "Example University" receives a discount for that specific game. Pat then starts the procedure of purchasing the ticket and will be requested to demonstrate the ownership of an alumni verifiable credential. Pat's digital wallet opens up and asks him if he would like to provide a previously generated verifiable credential. Pat chooses the alumni verifiable credential, which is then configured into a verifiable presentation that will be sent to the verifier to be verified. The proof in Figure 2.9 is generated employing the private key corresponidng to Pat's DID document auhentication method.

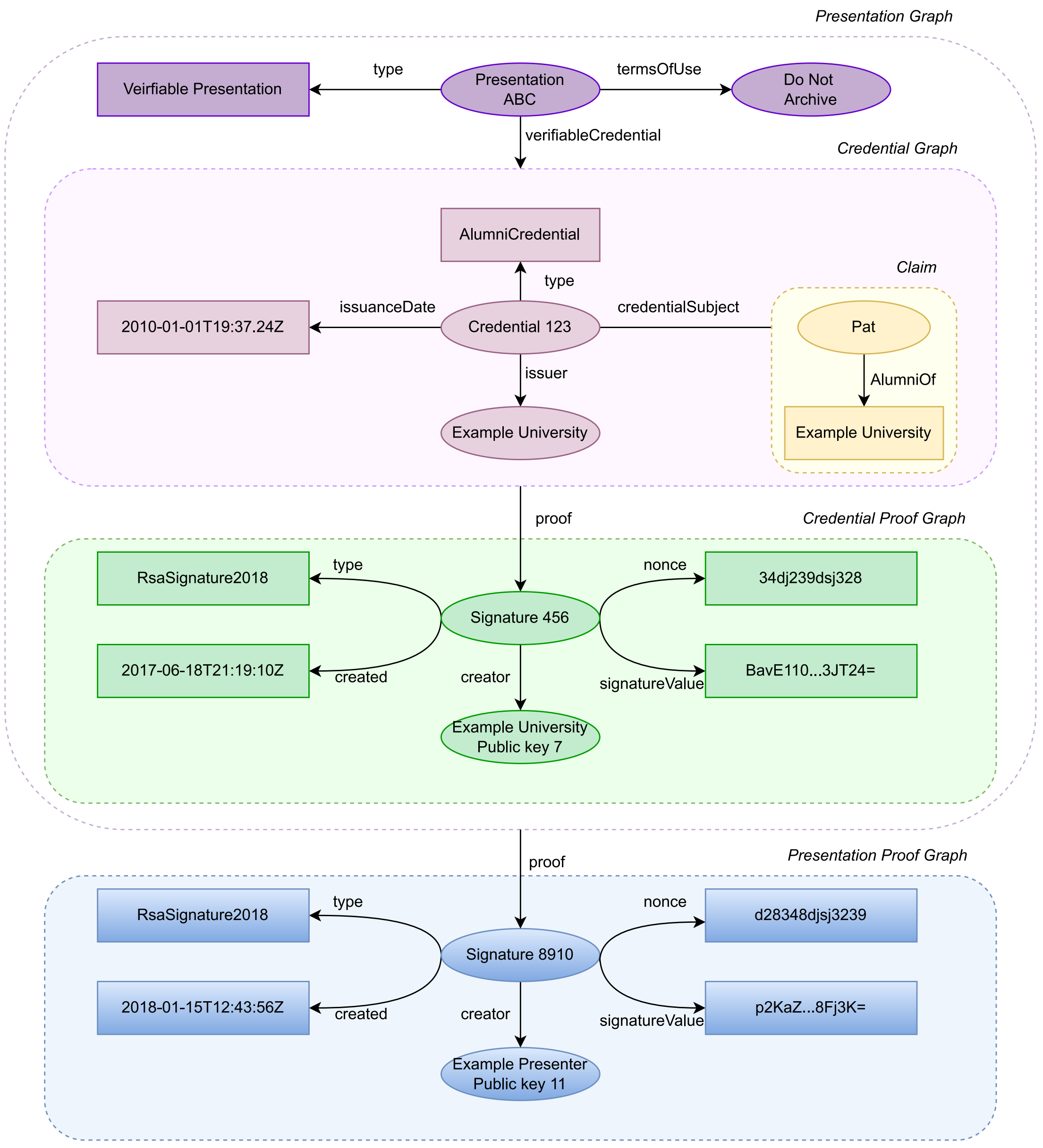
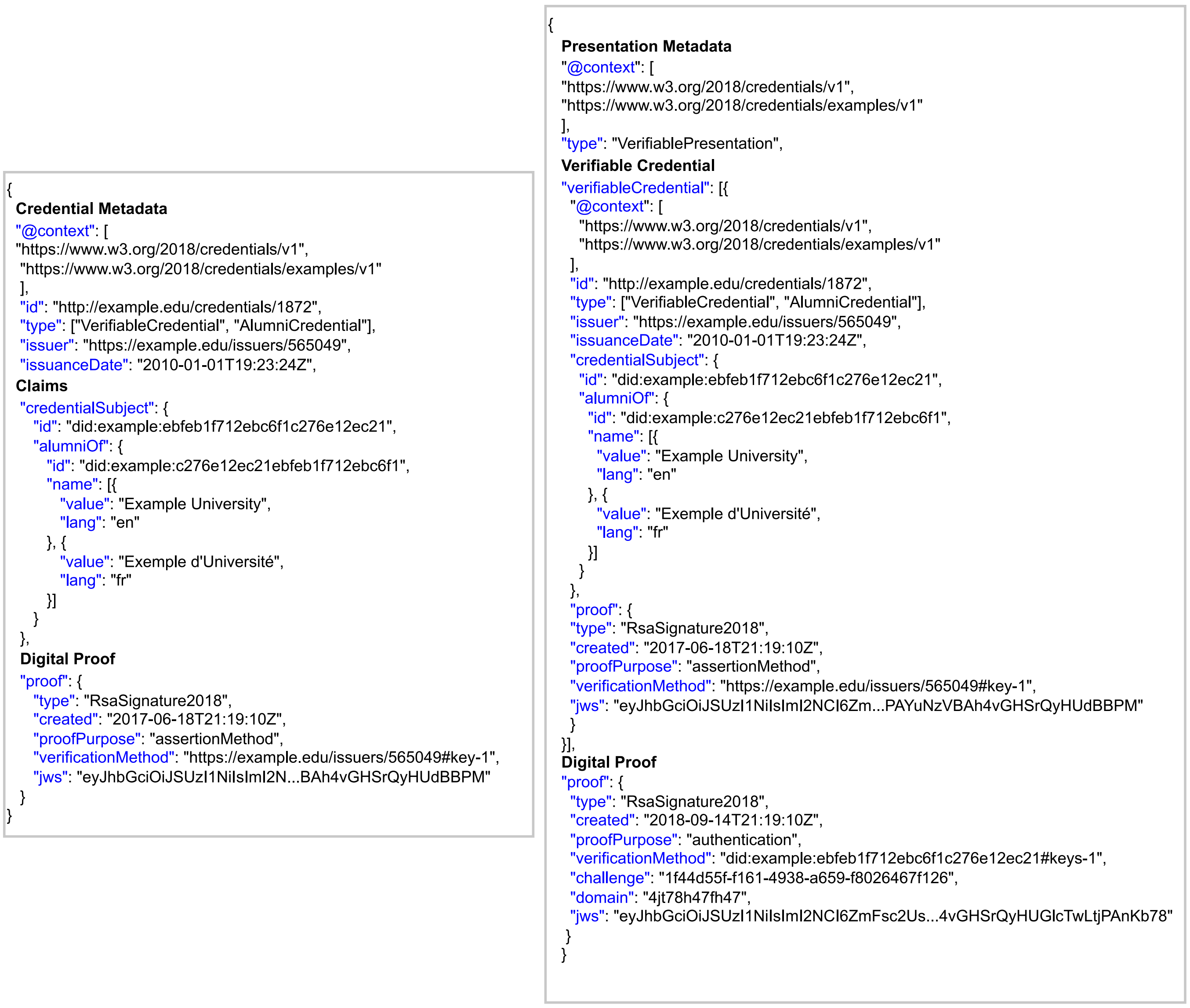


Figure 2.9: Pat's presentation graph (source: Verifiable Credentials - W3C).

Figure 2.10 shows the JSON representation of the Verifiable Credential and Verifiable Presentation discussed above.



```
{
 Credential Metadata
 "@context": [
 "https://www.w3.org/2018/credentials/v1",
  "https://www.w3.org/2018/credentials/examples/v1"
  ],
  "id": "http://example.edu/credentials/1872",
  "type": ["VerifiableCredential", "AlumniCredential"],
  "issuer": "https://example.edu/issuers/565049",
  "issuanceDate": "2010-01-01T19:23:24Z",
 Claims
  "credentialSubject": {
    "id": "did:example:ebfeb1f712ebc6f1c276e12ec21",
    "alumniOf": {
      "id": "did:example:c276e12ec21ebfeb1f712ebc6f1",
      "name": [{
        "value": "Example University",
        "lang": "en"
      }, {
        "value": "Exemple d'Université",
        "lang": "fr"
      }]
    }
  },
  Digital Proof
  "proof": {
    "type": "RsaSignature2018",
    "created": "2017-06-18T21:19:10Z",
    "proofPurpose": "assertionMethod",
    "verificationMethod": "https://example.edu/issuers/565049#key-1",
    "jws": "eyJhbGciOiJSUzI1NiIsImI2N...BAh4vGHSrQyHUdBBPM"
  }
}
```

```
{
 Presentation Metadata
 "@context": [
 "https://www.w3.org/2018/credentials/v1",
 "https://www.w3.org/2018/credentials/examples/v1"
 ],
 "type": "VerifiablePresentation",
 Verifiable Credential
 "verifiableCredential": [{
  "@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://www.w3.org/2018/credentials/examples/v1"
  ],
  "id": "http://example.edu/credentials/1872",
  "type": ["VerifiableCredential", "AlumniCredential"],
  "issuer": "https://example.edu/issuers/565049",
  "issuanceDate": "2010-01-01T19:23:24Z",
  "credentialSubject": {
    "id": "did:example:ebfeb1f712ebc6f1c276e12ec21",
    "alumniOf": {
      "id": "did:example:c276e12ec21ebfeb1f712ebc6f1",
      "name": [{
        "value": "Example University",
        "lang": "en"
      }, {
        "value": "Exemple d'Université",
        "lang": "fr"
      }]
    }
  },
  "proof": {
  "type": "RsaSignature2018",
  "created": "2017-06-18T21:19:10Z",
  "proofPurpose": "assertionMethod",
  "verificationMethod": "https://example.edu/issuers/565049#key-1",
  "jws": "eyJhbGciOiJSUzI1NiIsImI2NCI6Zm...PAYuNzVBAh4vGHSrQyHUdBBPM"
  }
 }],
 Digital Proof
 "proof": {
  "type": "RsaSignature2018",
  "created": "2018-09-14T21:19:10Z",
  "proofPurpose": "authentication",
  "verificationMethod": "did:example:ebfeb1f712ebc6f1c276e12ec21#keys-1",
  "challenge": "1f44d55f-f161-4938-a659-f8026467f126",
  "domain": "4jt78h47fh47",
  "jws": "eyJhbGciOiJSUzI1NiIsImI2NCI6ZmFsc2Us...4vGHSrQyHUGlcTwLtjPAnKb78"
 }
}
```

Figure 2.10: Verifiable Credential **(left)** and Verifiable Presentation **(right)** example.

2.5 SSI framework

After discussing the main features of the single elements at the core of the SSI paradigm, it is time to explore how these components connect with each other. The SSI framework presents itself as a stack that expands into four layers (Figure 2.11). The first layer consists of all the different DLTs where the DID documents are stored and manipulated. Each DLT implements a *DID method* interface which allows to perform CRUD (Create, Resolve, Upadate, and Delete) operations on DID documents. Each DLT must have its own *DID method* interface.

The second layer (*DID communication*) is responsible of establishing a secure and private connection among two parties. In order to be secure each actor should prove its identity to the other. DIDs are employed to ensure this: one party sends its DID to the other, the receiving party resolves the DID into the corresponding DID document through the DID method of the related DLT and proposes an authentication challenge by using the public key stored in the DID document of the sender to strongly verify the identity of the latter and vice versa.

The verifiable credential trust triangle appears in the *Authorization* layer where issuers, holders and verifiers exchange credentials and proofs using credential exchange protocols that run on top of the *DID Communication* layer. The fourth layer, the *Application Ecosystem* includes all the applications that exploit the lower layers with which humans can interact with [7].

The work of my thesis aims at implementing the secure channel proposed in the second layer of the stack by employing an already existing security protocol, the **Transport Layer Security** protocol, and make it compliant with the Decentralized Identifiers.

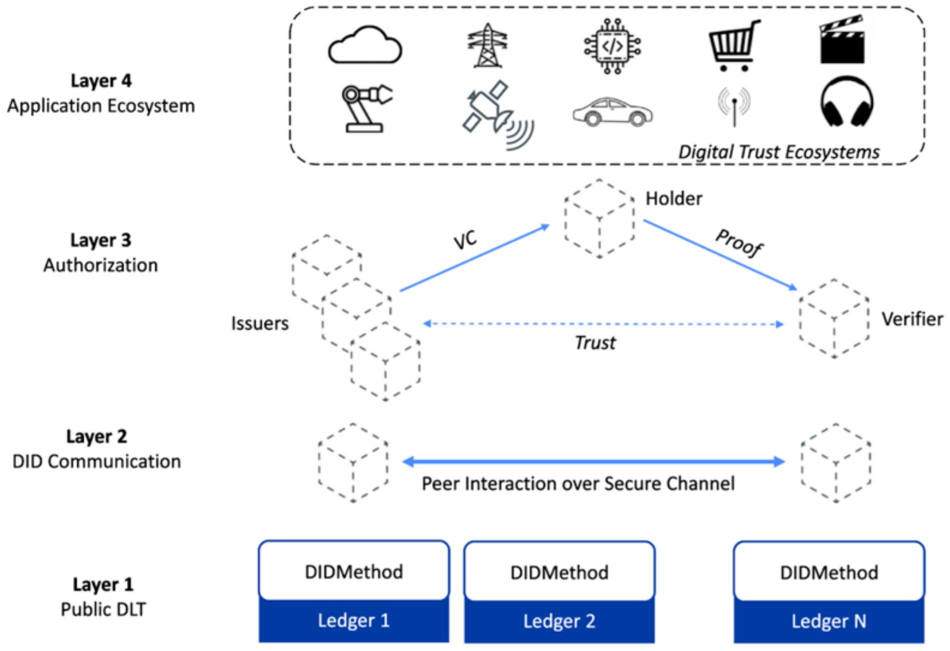


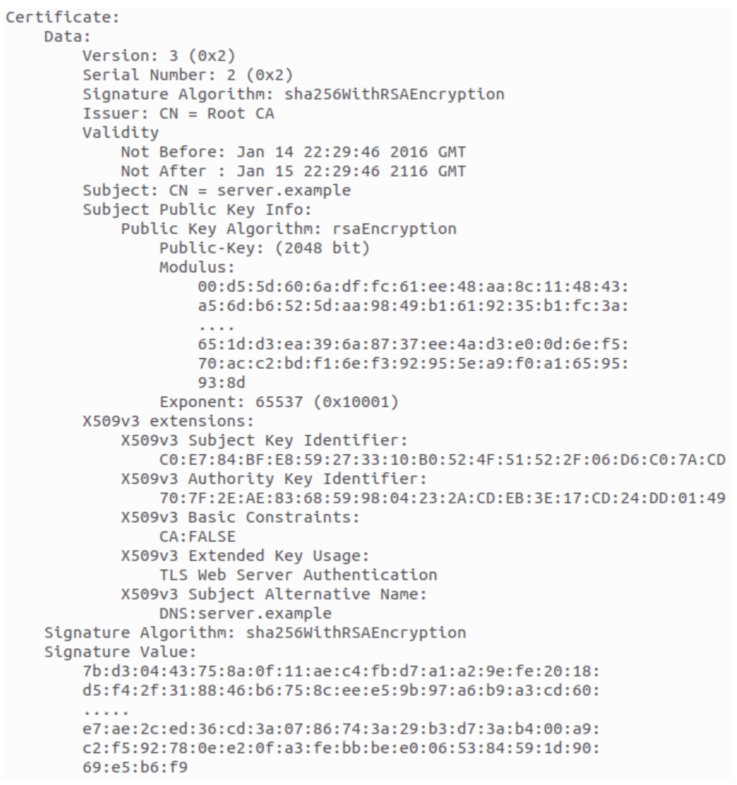
Figure 2.11: SSI stack

Chapter 3

SSI-aware TLS 1.3 handshake in OpenSSL

3.1 X.509

X.509 (Figure 3.1) is a standard format for public key certificates, which are data structures that securely tie a public key of an asymmetric keypair to some attributes like information about the key itself, information about the identity of its owner (website, individual, organization) and the digital signature of the issuer, named **Certification Authority** (CA). There are several fields in an X.509 certificate, specified in [8]. The most common ones are:



```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 2 (0x2)
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN = Root CA
        Validity
            Not Before: Jan 14 22:29:46 2016 GMT
            Not After : Jan 15 22:29:46 2116 GMT
        Subject: CN = server.example
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:d5:5d:60:6a:df:fc:61:ee:48:aa:8c:11:48:43:
                    a5:6d:b6:52:5d:aa:98:49:b1:61:92:35:b1:fc:3a:
                    ....
                    65:1d:d3:ea:39:6a:87:37:ee:4a:d3:e0:0d:6e:f5:
                    70:ac:c2:bd:f1:6e:f3:92:95:5e:a9:f0:a1:65:95:
                    93:8d
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                C0:E7:84:BF:E8:59:27:33:10:B0:52:4F:51:52:2F:06:D6:C0:7A:CD
            X509v3 Authority Key Identifier:
                70:7F:2E:AE:83:68:59:98:04:23:2A:CD:EB:3E:17:CD:24:DD:01:49
            X509v3 Basic Constraints:
                CA:FALSE
            X509v3 Extended Key Usage:
                TLS Web Server Authentication
            X509v3 Subject Alternative Name:
                DNS:server.example
    Signature Algorithm: sha256WithRSAEncryption
    Signature Value:
        7b:d3:04:43:75:8a:0f:11:ae:c4:fb:d7:a1:a2:9e:fe:20:18:
        d5:f4:2f:31:88:46:b6:75:8c:ee:e5:9b:97:a6:b9:a3:cd:60:
        .....
        e7:ae:2c:ed:36:cd:3a:07:86:74:3a:29:b3:d7:3a:b4:00:a9:
        c2:f5:92:78:0e:e2:0f:a3:fe:bb:be:e0:06:53:84:59:1d:90:
        69:e5:b6:f9
```

Figure 3.1: X.509 Certificate example.

- **version number**: version of the X.509 standard, the latest one is v3;
- **serial number**: uniquely identifies a certificate;

- **signature algorithm ID**: the algorithm the CA has employed to sign the certificate contents;
- **issuer name**: the name of the creator of the certificate which is specified by following the distinguished name (DN) syntax that consists of three fields:
  - C: the country;
  - O: the organization;
  - CN: the common name;
- **validity**: defines the period of time in which the certificate is considered valid;
- **subject**: the entity (machine, individual, organization) that controls the private key corresponding to the public key being certified;
- **subject public key info**: expresses the algorithm of the subject's public key and the public key itself.
- **extensions**: offers ways to bind more attributes to users or public keys and to handle relationships among certificate authorities.
- **digital signature**: the digital signature of the issuing certification authority computed over the certificate.

A **public key infrastructure** (PKI) is a system responsible for the creation, distribution and revocation of public key certificates. These certificates can be revoked before their expiration date on request by both the subject (if, for instance, they lose control over their private key) and the issuer (if, for instance, they realize a certificate has been released to someone claiming a fake identity).

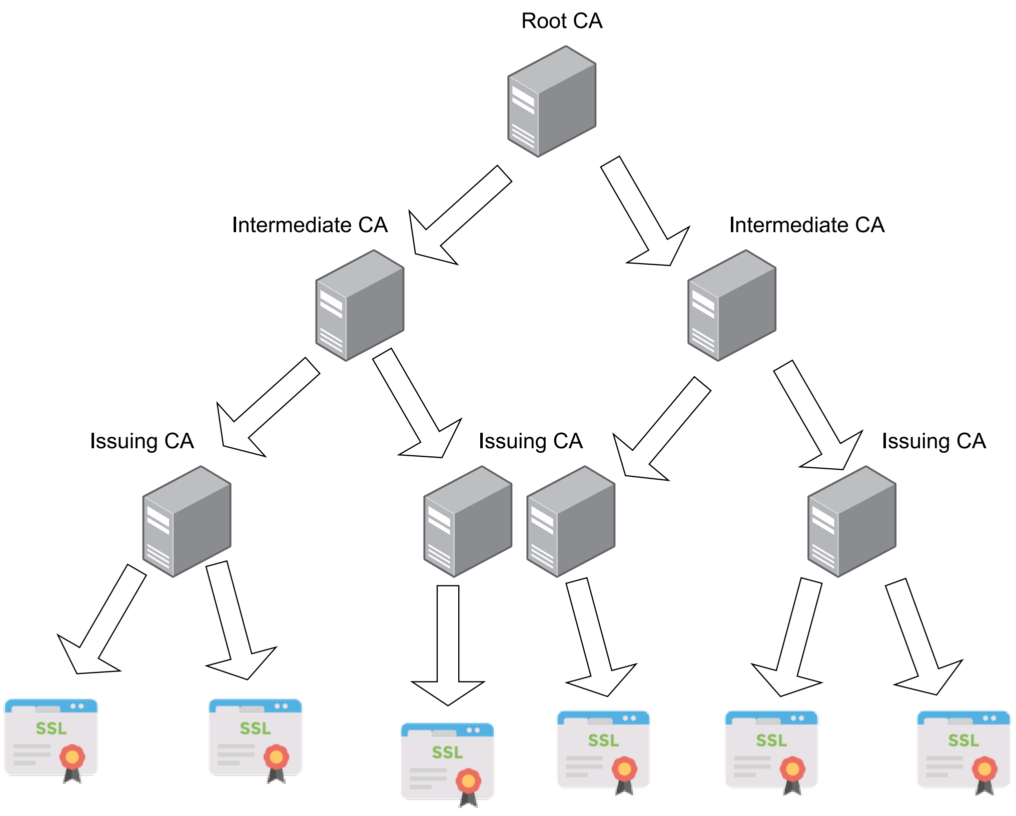


Figure 3.2: Certification Authority hierarchy.

Furthermore, CAs are organized hierarchically (Figure 3.2). The **root CA** stands at the top, self-signs its own certificate and issues certificates for intermediate CAs that in turn will release certificates for CAs one level below and so on down to CAs that operate directly with end entities [9]. There is not a single root CA in the world, but there are several to avoid that a single entity control all the certificates.

X.509 certificates are employed to verify identities, so when an entity receives a signature from a peer it must check its validity by making sure that at the moment of signature the certificate of the sender and the certificates of all the CAs that build the chain-of-trust are not expired or issued by an untrusted CA, otherwise the signature is compromised. Two are the possible ways to assess the validity of a certificate:

- through the CRL (**Certificate Revocation List**): the CA creates a list of revoked certificates and digitally signs it to keep a malicious entity from removing or adding certificates to the list;
- employing the OCSP (**Online Certificate Status Protocol**): this mechanism is preferred for real-time checks as the only concern is to verify that the certificate is valid now. The response is signed by the server so it is not possible for someone to provide a fake response.

3.2 Transport Layer Security Protocol

The **Transport Layer Security** (TLS) protocol allows client/server applications to communicate securely over the Internet and runs on top of a reliable transport protocol (e.g. TCP) in the OSI model (Figure 3.3). Specifically it provides a secure channel between two communicating parties by enabling [10]:

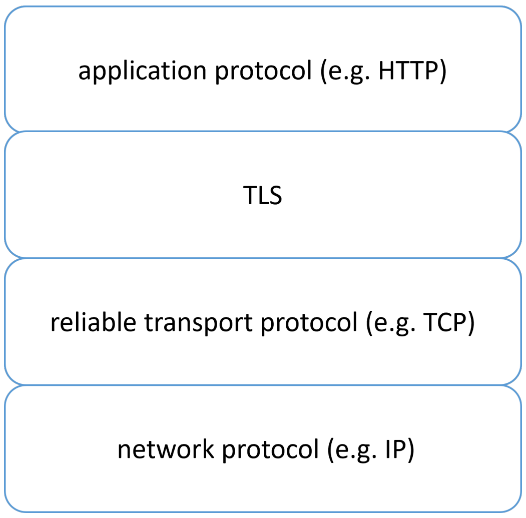

application protocol (e.g. HTTP)
TLS
reliable transport protocol (e.g. TCP)
network protocol (e.g. IP)

Figure 3.3: TCP/IP stack with TLS.

- **authentication**: the server side of the communication is always authenticated, the client side is optionally authenticated;
- **confidentiality**: once the channel is established, messages can be read only by the two endpoints;
- **integrity**: if an attacker manipulates the data sent over the channel, the endpoints can detect the changes.

Version **1.3** of the protocol, which is also the latest at the time of writing, consists of three primary components (Figure 3.4):

- a **handshake protocol**: authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material.

- an **alert protocol**: contains closure and error messages. These messages are encrypted according to the current connection state.
- a **record protocol**: splits the data to be transmitted into a series of *records*, each of which is independently protected using the cryptographic parameters negotiated during the handshake phase. This protocol is exploited by the two above and by the application one that runs on top of TLS to exchange data. For instance, HTTPS stands for HTTP protocol on top of TLS.

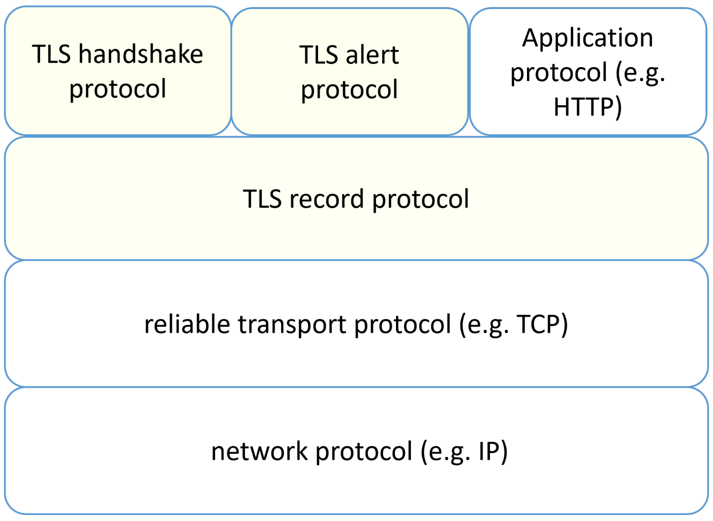

TLS handshake protocol
TLS alert protocol
Application protocol (e.g. HTTP)
TLS record protocol
reliable transport protocol (e.g. TCP)
network protocol (e.g. IP)

Figure 3.4: TLS architecture.

TLS is application protocol independent, however, it does not specify how protocols add security with TLS. The designers and developers of the protocols placed on top of TLS must establish how to initiate TLS handshaking and how to interpret the authentication certificates exchanged.

In addition, TLS introduces the concepts of **session** and **connection** (Figure 3.5): the first one is a logical association among client and server produced by the handshake protocol, establishes a set of security parameters and is shared by one or more TLS connections. The second one is a transient channel between client and server associated to one specific TLS session.

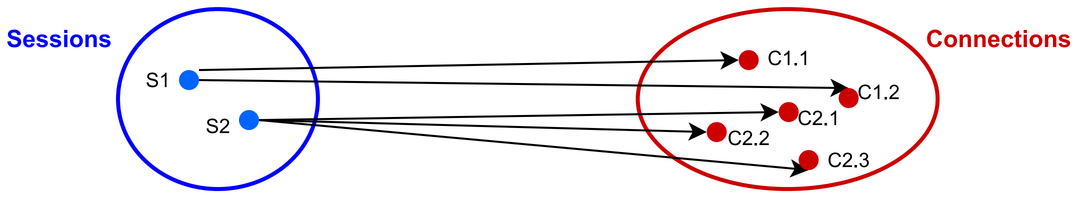


Figure 3.5: session-connection relationship in TLS.

The handshake protocol is the first used when server and client start communicating with each other. Once the handshake is complete the peers use the established keys to protect application-layer traffic. If there is a failure in the handshake the connection is terminated right away, optionally preceded by an alert message. TLS 1.3 provides three key exchange modes:

- **EC(DHE)**: Diffie-Hellman over either finite fields or elliptic curves;
- **PSK-only**: pre-shared key only;
- **PSK with (EC)DHE**.

Figure 3.6 shows the messages exchanged during a full handshake, and below I will break down the content of each handshake message proposed in the figure.

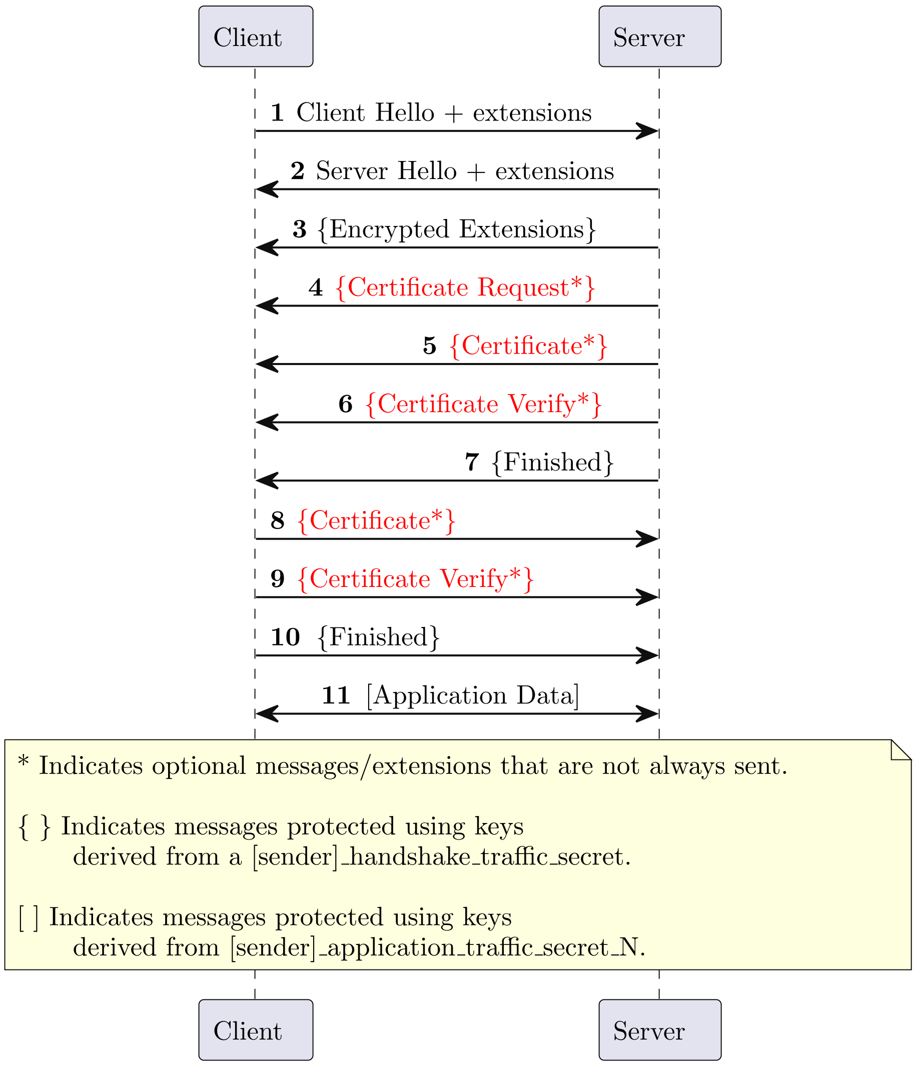

Client
Server
1 Client Hello + extensions
2 Server Hello + extensions
3 {Encrypted Extensions}
4 {Certificate Request*}
5 {Certificate*}
6 {Certificate Verify*}
7 {Finished}
8 {Certificate*}
9 {Certificate Verify*}
10 {Finished}
11 [Application Data]
* Indicates optional messages/extensions that are not always sent.
{ } Indicates messages protected using keys derived from a [sender]_handshake_traffic_secret.
[ ] Indicates messages protected using keys derived from [sender]_application_traffic_secret_N.
Client
Server

Figure 3.6: Original TLS full handshake protocol.

3.2.1 Extensions

Some TLS messages contain extensions encoded following the tag-length-value syntax. Figure 3.7 shows the extension structure and the list of all the available extensions in TLS 1.3: the *extension type* field associates a constant value to the specific extension type, the *extension data* expresses information related to a specific extension type and the *opaque* data type is used for uninterpreted data.

Usually an extension request is followed by an extension response, however some extensions do not require any response. The client sends its extension requests in the Client Hello message and server responds in the Server Hello, Encrypted Extensions, and Certificate messages. The server, instead, can produce extension requests in the Certificate Request message which a client may reply to with the Certificate message.

```
struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;

enum {
   server_name(0),
   max_fragment_length(1),
   status_request(5),
   supported_groups(10),
   signature_algorithms(13),
   use_srtp(14),
   heartbeat(15),
   application_layer_protocol_negotiation(16),
   signed_certificate_timestamp(18),
   client_certificate_type(19),
   server_certificate_type(20),
   padding(21),
   pre_shared_key(41),
   early_data(42),
   supported_versions(43),
   cookie(44),
   psk_key_exchange_modes(45),
   certificate_authorities(47),
   oid_filters(48),
   post_handshake_auth(49),
   signature_algorithms_cert(50),
   key_share(51),
   (65535)
} ExtensionType;
```

Figure 3.7: Extension structure and list of all the available extension types in TLS (source: TLS 1.3 (RFC8446)).

3.2.2 Client Hello

Client Hello is required to be the first message sent by the client when connecting to a server. It has the following structure:

- **protocol version**: in older versions of TLS represents the highest version of the protocol the client supports for version negotiation. In the case of TLS 1.3 it is set to 1.2 and the client will express its preferences in the *supported versions* extension .
- **random**: 32 bytes generated by a secure random number generator.
- **session id**: in prior versions utilized for session resumption which is now implemented with pre-shared keys, so it is kept for backward compatibility and should be a single byte set to zero.
- **cipher suites**: TLS 1.3 supports only AEAD (Authenticated Encryption with Associated Data) symmetric ciphers, so this field contains the list of *symmetric ciphers* (and encryption mode) to protect TLS records, paired with a *hash algorithm* to be used with HKDF (HMAC Key derivation function) to calculate handshake and application keys, in descending order of client preferences. The server can ignore some of the elements in this list if it does not support or wish to use them. For instance TLS_AES_128_GCM_SHA256 uses AES-128 with GCM (Galois Counter Mode) plus SHA-256 hash.

- **compression methods**: compression has been disabled in TLS 1.3 so this field contains exactly one byte set to zero which expresses the *null* compression method.
- **extensions**: they are used to request additional features to the server. In order to preserve backward compatibility with previous versions of the protocol new functionalities are expressed here and for this reason some extensions are mandatory in TLS 1.3 like the *supported versions* one to indicate version 1.3. A client that wants to exchange keys in (EC)DHE mode produces one or more keypairs, based on the (EC)DHE groups it supports, and will send the public keys in descending order of its preference in the *key share* extension.

3.2.3 Server Hello

The server processes the Client Hello and sends in response the Server Hello message. It has the same structure of the Client Hello:

- **legacy version**: contains the version number corresponding to TLS 1.2.
- **random**: 32 bytes generated by a secure random number generator.
- **session id**: echoes the session id value sent by the client.
- **cipher suite**: the selected cipher suite among the ones received by the client.
- **compression method**: a single byte with value zero.
- **extensions**: contains responses only to the extensions sent by the client and that are needed to negotiate the cryptographic context. The *supported versions* extension must be present in each Server Hello for TLS 1.3 negotiation and again if a server received a *key share* extension from the client it will have to select one entry in the list of (EC)DHE, then generate a keypair for that group and in the end send back the public key in the same extension.

3.2.4 Encrypted Extensions

After the Server Hello all the handshake messages will be encrypted, in fact both the server and the client now have all the material to derive handshake keys (the server possesses the *client public key* from Client Hello and the *server private key* from key generation, the client has the *client private key* from key generation and the *server public key* from Server Hello) to cipher future messages.

This message contains all the extensions that are not needed to establish cryptographic parameters and are not related with certificates, like the *application layer protocol negotiation* (ALPN) extension that selects the application protocol to be used on top of TLS.

3.2.5 Certificate Request

A server that requests client authentication must send this message, otherwise it should not be sent. It consists of two fields:

- a **certificate request context**: identifies the certificate request and the client will echo it in its Certificate message. In order to avoid replay of client Certificate Verify messages this value must be unique within the scope of this connection. Thus, it must be zero-valued single-byted unless used for *post-handshake authentication* (a server can request client authentication at any time after the handshake has completed by sending a Certificate Request): in this case it should be generated randomly, so an attacker that has temporary access to the client's private key will not be capable of pre-computing valid Certificate Verify messages.
- **extensions**: this field is used by the server to request features the client's certificate should have. The *signature algorithms* extension must always be provided and contains a set of signature algorithms the server is willing to accept on Certificate Verify message sent by the client, the other extensions are optional.

3.2.6 Certificate

This message contains the endpoint's X.509 certificate signed by the CA plus the whole chain of certificates up to the root CA excluded. The client sends a Certificate message solely upon receiving the Certificate Request message from the server, the latter instead, must always send this message unless PSK key exchange mode is adopted. It consists of:

- a **certificate request context**: if replying to a Certificate Request it carries the value of the certificate request context in that message, otherwise it must be zero length;
- a **certificate list**: represents the certificate chain, where each element of the chain contains an *X.509 certificate* and a set of *extensions* related to that certificate. The server must reflect the extensions in the Client Hello message and that are related to this message, instead the client should send only the ones contained in the Certificate Request message from the server. The first certificate also holds the extensions concerning to the whole chain.

3.2.7 Certificate Verify

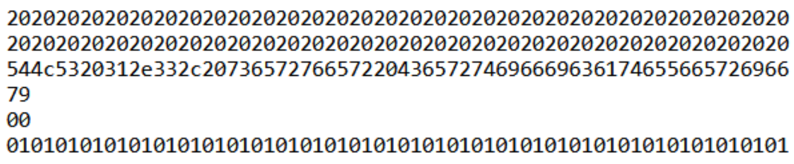
This message proves that an endpoint is the owner of the private key bound to the public key present in its certificate by signing the hash of all the handshake messages up to this point, thus providing integrity of the handshake as well. The client sends this message only when is requested to authenticate. It contains:

- the **signature algorithm**;
- the **signature**: the content of the digital signature with the above algorithm.

The signature is computed over the concatenation of four elements:

- a string composed of octet 32 (0x20) repeated 64 times;
- a context string that is "TLS 1.3, server CertificateVerify" on server side and "TLS 1.3, client CertificateVerify" on client side;
- a single 0 byte that acts as the separator;
- a hash of all the messages exchanged so far.

For instance if the hash of the messages were 32 bytes of 01 (which could be the output of SHA-256 hash algorithm) the digital signature will be applied on the content proposed in Figure 3.8.



```
2020202020202020202020202020202020202020202020202020202020202020
2020202020202020202020202020202020202020202020202020202020202020
544c5320312e332c207365727665722043657274696669636174655665726966
79
00
0101010101010101010101010101010101010101010101010101010101010101
```

Figure 3.8: Certificate Verify content example.

3.2.8 Finished

It is the final message and consists of a MAC(Message Authentication Code) over the entire handshake, so both parties can assert that the handshake has not been tampered with and tie their identities to the exchanged keys. After sending this message an endpoint derives the application keys from its handshake keys and can start sending and receiving application data.

3.3 SSI-aware TLS 1.3 handshake model

Comparing X.509 certificates and Decentralized Identifiers we can immediately notice that root CAs are represented by centralised entities who have full control on all the certificates they issue and so they can revoke them as they wish, while Decentralized Identifiers can be emitted by anyone and they benefit from the data immutability feature of the DLT.

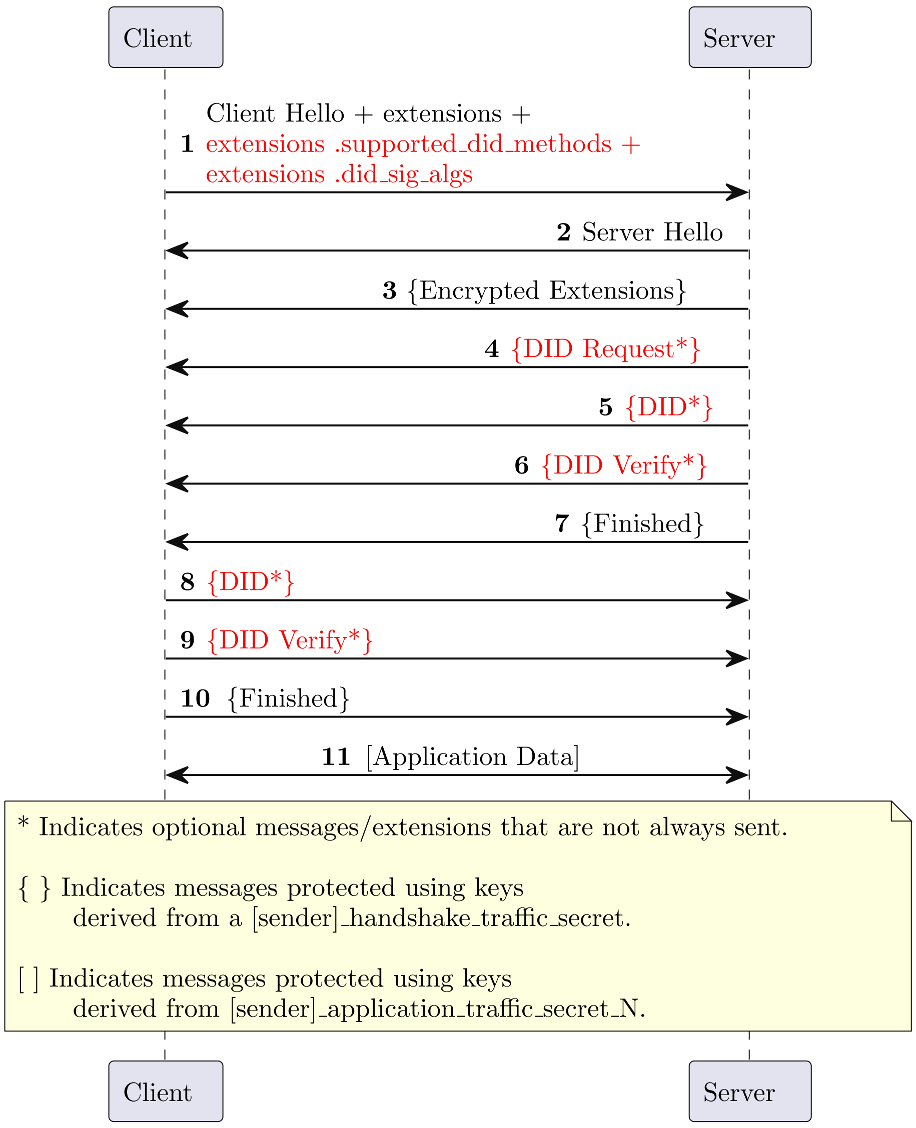

Client
Server
Client Hello + extensions +
1 extensions .supported_did_methods +
extensions .did_sig_algs
2 Server Hello
3 {Encrypted Extensions}
4 {DID Request*}
5 {DID*}
6 {DID Verify*}
7 {Finished}
8 {DID*}
9 {DID Verify*}
10 {Finished}
11 [Application Data]
* Indicates optional messages/extensions that are not always sent.
{ } Indicates messages protected using keys derived from a [sender]_handshake_traffic_secret.
[ ] Indicates messages protected using keys derived from [sender]_application_traffic_secret_N.
Client
Server

Figure 3.9: SSI-aware handshake protocol.

I have designed a new handshake model that provides a new authentication mechanism (without disrupting the default one) that leverages the potential of DIDs. Figure 3.9 proposes the structure of the new handshake protocol enabling the SSI paradigm. In the next section is detailed how each TLS message is adapted to support DIDs mechanisms.

3.3.1 Client Hello

Client Hello structure stays the same, but now the message is equipped with two completely new extensions:

- a **supported did methods** extension: expresses that the client is willing to authenticate the server through the use of Decentralized Identifiers and contains the list of *DID methods* for which a client can resolve a DID into a DID document. If the server does not have a DID or it does, but the DID method of its DID is not present in this list, it must terminate the handshake with an alert message. This extension has the same structure of all the other extensions (Figure 3.10): the *extension type* field contains a new value associated to the **supported_did_methods** enum, and the *extension data* field is of type **SupportedDid-Methods**, which in turn is a list of single-byte values utilized to depict the possible DID methods. Figure 3.11 shows how to properly build this extension.

```
struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;

enum {
   server_name(0),
   max_fragment_length(1),
   ...,
   key_share(51),
   supported_did_methods(60),
   (65535)
} ExtensionType;

enum { ott(0), btcr(1), sov(2), .., (255)
 } DidMethod;

struct {
   DidMethod supported_did_methods<1..2^8-1>
} SupportedDidMethods;
```

Figure 3.10: Supported DID Methods extension structure.

```
value = 00 3C 00 04 03 00 01 02 (in hex)

- 00 3C - 0x3C (60) assigned value for extension SupportedDidMethods
- 00 04 - 0x04 (4) bytes of SupportedDidMethods extension data follows
- 03 - 0x03 (3) bytes of data are in the following list of did methods
- 00 - 0x00 (0) assigned value for ott
- 01 - 0x01 (1) assigned value for btcr
- 02 - 0x02 (2) assigned value for sov
```

Figure 3.11: Supported DID Methods extension example.

- a **did signature algorithms** extension: must be present only if the above extension is produced and contains the list of signature algorithms (authentication methods), allowed in DID documents, the client is willing to verify on a DID Verify message later sent by the server. The server's DID document authentication method (its public key) must be compatible with one of these algorithms, otherwise the server must abort the handshake with an alert message. A new *extension type* named **did_signature_algorithms** has been inserted (Figure 3.12) and the **DidSignatureAlgorithms**, which consists of a sequence of authentication methods expressed as single-byte values, fills the *extension data* field;

```
struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;

enum {
   server_name(0),
   max_fragment_length(1),
    ...,
    key_share(51),
    supported_did_methods(60),
    did_signature_algorithms(61)
    (65535)
} ExtensionType;

enum { ecdsa_secp256k1_verification_key2019(0),
    ed25519_verification_key2018(1), rsa_verification_Key2018(2), .., (255)
 } DidSignatureScheme;

struct {
   DidSignatureScheme did_signature_algorithms<1..2^8-1>
} DidSignatureAlgorithms;
```

Figure 3.12: DID Signature Algorithms extension structure.

```
value = 00 3D 00 04 03 00 01 02 (in hex)

- 00 3D - 0x3D (61) assigned value for extension DidSignatureAlgorithms
- 00 04 - 0x04 (4) bytes of DidSignatureAlgorithms extension data follows
- 03 - 0x03 (3) bytes of data are in the following list of did signature
    algorithms
- 00 - 0x00 (0) assigned value for ecdsa_secp256k1_verification_key2019
- 01 - 0x01 (1) assigned value for ed25519_verification_key2018
- 02 - 0x02 (2) assigned value for rsa_verification_key2018
```

Figure 3.13: DID Signature Algorithms extension example.

3.3.2 DID Request

The server could request client authentication by sending a **DID Request** message to have the client authenticate through a challenge-response authentication mechanism employing the client's DID document *authentication method*. Figure 3.14 presents the addition of this new message in the handshake struct, and below I discuss the fields of this new message:

- **did request context**: must hold the value zero unless used for post-handshake authentication. In fact when the server requests post-handshake authentication to the client the context should be made unpredictable so that an attacker who is in possession of the client's private key will not be able to pre-compute valid DID Verify messages.
- **extensions**: it contains two extensions that must always be present (if a client receives this message and at least one of the two extensions is missing it must terminate the handshake with an appropriate alert message):
  - **supported did methods**: it is a set of the DID methods the two endpoints have

in common. If the client does not have a DID or it does but the DID method is not present in this list, it must abort the handshake;

  - **did signature algorithms**: list of signature algorithms allowed in DID documents the server is willing to verify on DID Verify message sent by the client.

```
enum {
     client_hello(1), server_hello(2), new_session_ticket(4),
     end_of_early_data(5), encrypted_extensions(8), certificate(11),
    certificate_request(13), certificate_verify(15), finished(20),
   key_update(24), did_request(31), message_hash(254), (255)
} HandshakeType;

struct {
   HandshakeType msg_type; /* handshake type */ 1 Byte
   uint24 length; /* remaining bytes in message */ 3 Bytes
   select (Handshake.msg_type) {
      case client_hello: ClientHello;
      case server_hello: ServerHello;
      case end_of_early_data: EndOfEarlyData;
      case encrypted_extensions: EncryptedExtensions;
      case certificate_request: CertificateRequest;
      ......
      case key_update: KeyUpdate;
      case did_request: DidRequest;
    };
} Handshake;

struct {
   opaque did_request_context<0..2^8-1>;
   Extension extensions<2..2^16-1>;
} DidRequest;
```

Figure 3.14: DID Request message structure.

Figure 3.15 demonstrates how to interpret the sequence of bytes that composes a DID Request message .

```
value = 1F 00 00 12 00 00 0F 00 3D 00 03 02 00 01 00 3C 00 04 03 00 01 02

1F - 0x1F (31) handshake message type DID Request
00 00 12 - 0x12 (18) bytes of DID Request payload
00 - 0 bytes of request context follows
00 0F - 0x0F (15) bytes of extensions will follow
00 3D - 0x3D(61) assigned value for extension DID Signature Algorithms
00 03 - 0x03 (3) bytes of DID Signature Algorithms extension data follows
02 - 0x02 (2) bytes of data are in the following list of algorithms
00 - assigned value for ecdsa_secp256k1_verification_key2019
01 - assigned value for ed25519_verification_key2018
00 3C - 0x3C (60) assigned value for extension supported did methods
00 04 - 0x04 (4) bytes of Supported Did Methods extension data follows
03 - 0x03 (3) bytes of data are in the following list of did methods
00 - 0x00 (0) assigned value for ott
01 - 0x01 (1) assigned value for btcr
02 - 0x02 (2) assigned value for sov
```

Figure 3.15: DID Request message example.

3.3.3 DID

This message is employed when one of the endpoints needs to provide its own identity: the server must send a DID message only if PSK mode has not been selected and it has previously received the *supported did methods* and *did signature algorithms* extensions from the client in Client Hello message, instead the client will supply its own identity only upon a DID request.

The DID message carries the endpoint's DID and when the peer receives it, it will perform a resolve operation employing the *DID method* library (as previously said it provides CRUD operations) offered by that specific DLT that will return the DID document associated to the initial DID. At this point the peer will extract the public key from the DID document authentication method that later on will use to verify the signature computed by the sender contained in the DID Verify message. The DID message is populated by the following fields (Figure 3.16):

- a **did request context**: on server side must assume the value zero, on client side it must hold the same value of the *did request context* present in the DID Request message;
- a **did method**: contains on a single byte the DID method of the sender's DID;
- a **did value**: the actual value of the sender's DID.

An endpoint willing to send a DID message should follow the syntax proposed by Figure 3.17.

```
enum {
    client_hello(1), server_hello(2), new_session_ticket(4),
    end_of_early_data(5), encrypted_extensions(8), certificate(11),
    certificate_request(13), certificate_verify(15),
    finished(20), key_update(24), did(30),
    did_request(31), message_hash(254), (255)
} HandshakeType;

struct {
     HandshakeType msg_type; /* handshake type */ 1 Byte
     uint24 length; /* remaining bytes in message */ 3 Bytes
     select (Handshake.msg_type) {
     case client_hello: ClientHello;
     case server_hello: ServerHello;
     case end_of_early_data: EndOfEarlyData;
     case encrypted_extensions: EncryptedExtensions;
     ...
     case key_update: KeyUpdate;
     case did: Did;
     case did_request: DidRequest;
    };
} Handshake;

struct {
    opaque did_request_context<0..2^8-1>;
    DidMethod did_method;
    opaque did_value <0..2^16-1>;
} Did;
```

Figure 3.16: DID message structure.

```
value = 1E hl hl hl 00 00 dl dl xx xx xx .. .. (in hex)

- 1E 0x1E (30) Did message type
- hl hl hl bytes of Did payload
- 00 0x00 (0) bytes of request context
- 00 0x00 (0) assigned value for "ott" did method
- dl dl did_value length
- xx xx xx .. .. did_value
```

Figure 3.17: Server's DID message example.

3.3.4 DID Verify

After sending a DID message an endpoint must send a DID Verify message to prove ownership (and so its identity) of the private key associated to its DID document authentication method. Essentially it is a signature of all the handshake messages exchanged so far, so it also supplies integrity of the handshake up to this point. The structure (Figure 3.18) remains the same as the Certificate Verify message, with two subtle differences: the signature algorithm belongs to the DID Signature Scheme and the context string changes to “TLS 1.3, server DidVerify” on server side and “TLS 1.3, client DidVerify” on client side.

Figure 3.19 offers a simple example of the proper construction of a DID Verify message.

```
enum {
   client_hello(1), server_hello(2), new_session_ticket(4),
   end_of_early_data(5), encrypted_extensions(8), certificate(11),
   certificate_request(13), certificate_verify(15),
   finished(20), key_update(24), did(30),
   did_request(31), did_verify(32), message_hash(254), (255)
} HandshakeType;

struct {
    HandshakeType msg_type; /* handshake type */ 1 Byte
    uint24 length; /* remaining bytes in message */ 3 Bytes
    select (Handshake.msg_type) {
    case client_hello: ClientHello;
    case server_hello: ServerHello;
    ....
    case did: Did;
    case did_request: DidRequest;
    case did_verify  DidVerify;
   };
} Handshake;

struct {
   DidSignatureScheme algorithm;
   opaque signature<0..2^16-1>;
} DidVerify;

The signature is computed over the concatenation of:
- a string that consists of octet 32 (0x20) repeated 64 times;
- the context string = "TLS 1.3, server DidVerify" or "TLS 1.3, client
   DidVerify";
- a single 0 byte which serves as the separator;
- a hash of all the messages exchanged so far.
```

Figure 3.18: DID Verify message structure.

```
value = 20 hl hl hl 01 dl dl xx xx xx .. .. (in hex)

- 20 0x20 (32) Did Verify message type
- hl hl hl bytes of Did Verify payload
- 01 0x01 (1) assigned value for ed25519_verification_key2018
- dl dl signature length
- xx xx xx .. .. signature
```

Figure 3.19: DID Verify message example.

3.4 Server Authentication

If during the handshake the server does not want the client to prove its identity, then the handshake message flow will be subject to the modifications proposed in Figure 3.20, where the Did Request (on server side), Did and Did Verify (on client side) messages are omitted.

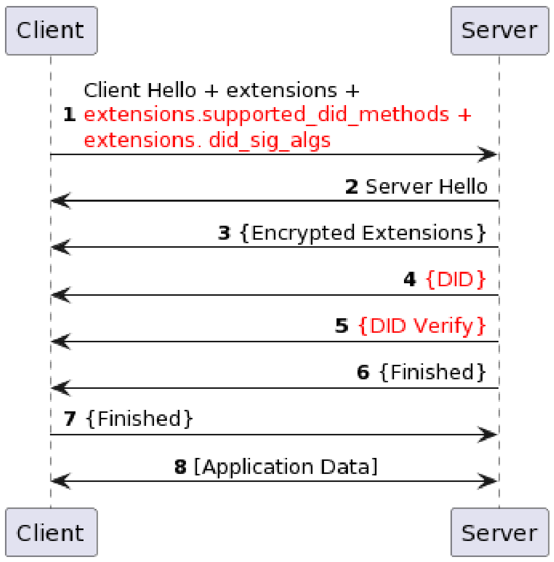


Figure 3.20: SSI-aware TLS handshake with Server Authentication.

3.5 Cross Authentication

There could also be circumstances of mutual authentication in which one of the two endpoints still wants the peer to authenticate with X.509 certificates: the client could have the server authenticate with certificates by not sending neither the *supported did methods* nor the *did signature algorithms* extensions, the server in turn should send the Certificate Request message instead of the DID Request one.

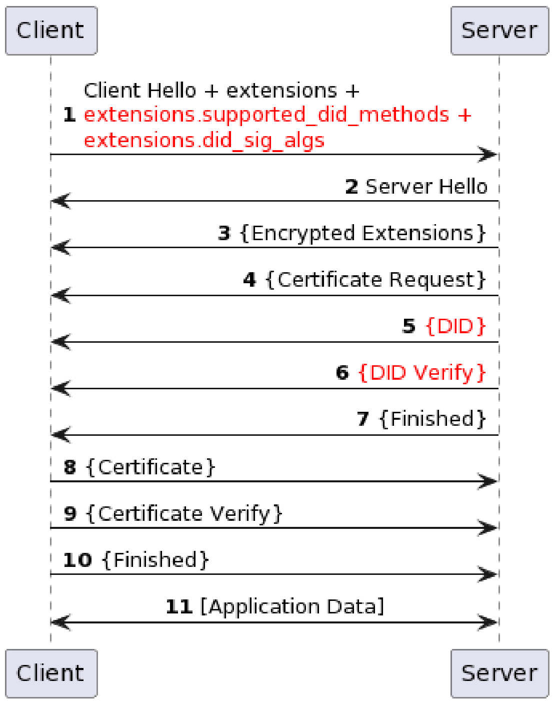


Figure 3.21: SSI-aware TLS handshake with Cross authentication. The client authenticates with an **X.509** certificate, the server with a **DID**.

In Figure 3.21 the client sends the two DID extensions so the server authenticates itself with

DID and DID Verify messages and requests client authentication with a Certificate Request message to which the client will reply with Certificate and Certificate Verify messages. In the case the server neither finds values in common for the DID extensions nor has a valid DID to send it must abort the handshake after processing the Client Hello by sending an appropriate alert message.

In Figure 3.22, instead, the client does not send the two DID extensions as the server authenticates through Certificate and Certificate Verify messages and will ask for client identity through a DID Request (the *supported did methods* extension include all the methods the server supports, since the client did not send this extension in Client Hello) message to which the client will respond with DID and DID Verify messages. In the case the client neither has values in common with the extensions present in the DID Request message nor possesses a valid DID to send back it must abort the handshake after receiving the DID Request message with an alert message.

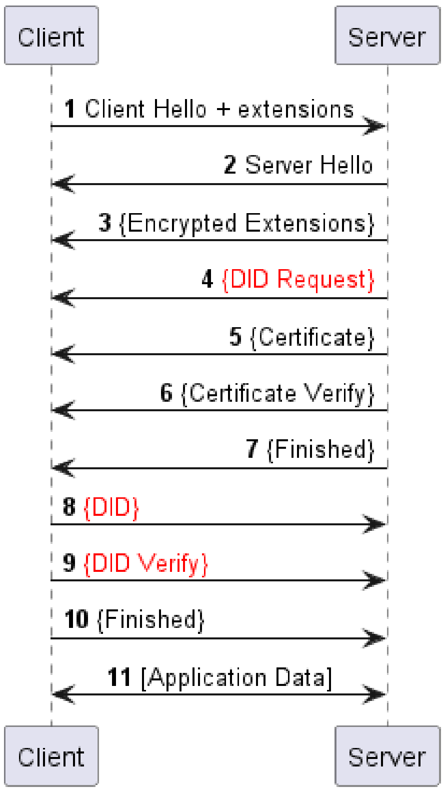
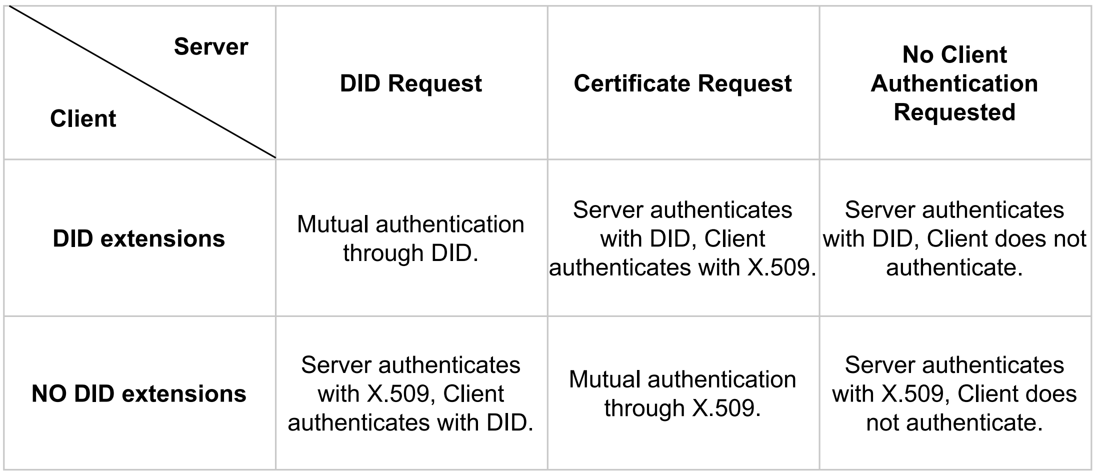


Figure 3.22: SSI-aware TLS handshake with Cross authentication. The client authenticates with a **DID**, the server with an **X.509** certificate.

The table in Figure 3.23 summarizes all the possible authentication scenarios considered in this work.



Server / Client	DID Request	Certificate Request	No Client Authentication Requested
DID extensions	Mutual authentication through DID.	Server authenticates with DID, Client authenticates with X.509.	Server authenticates with DID, Client does not authenticate.
NO DID extensions	Server authenticates with X.509, Client authenticates with DID.	Mutual authentication through X.509.	Server authenticates with X.509, Client does not authenticate.

Figure 3.23: Possible authentication scenarios in SSI-aware TLS handshake protcol.

3.6 OpenSSL

OpenSSL is an open-source cryptographic library that provides an implementation of the SSL (a previous version of TLS) and TLS protocols [11]. It is mainly written in the C programming language and offers a command line interface (CLI) to interact with it. I have selected **OpenSSL version 3.0**, which is the latest stable version, for the development of my solution since:

- it is open source;
- it is available for the most widely-spread operating systems such as Microsoft Windows, macOS and Linux;
- it supports TLS 1.3, the most recent version of the protocol.

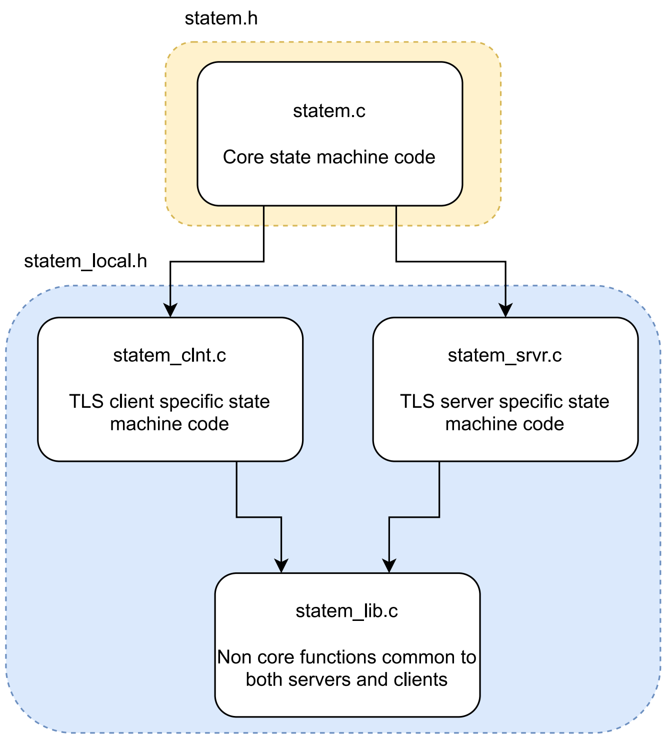

statem.h
statem.c
Core state machine code
statem_local.h
statem_clnt.c
TLS client specific state machine code
statem_srvr.c
TLS server specific state machine code
statem_lib.c
Non core functions common to both servers and clients

Figure 3.24: File hierarchy in `statem` directory in original OpenSSL.

In detail OpenSSL 3.0 consists of two primary components: the *crypto* library that implements basic cryptographic functions, and the *ssl* library that implements the SSL and TLS protocols employing the cryptographic utilities supplied by the first element. For the objective of my work I mainly focused on the second component. The ssl library consists of:

- a `record` directory that contains several files needed to implement the record protocol;
- a `statem` directory that contains several files needed to implement the handshake protocol;
- files that contain data structures and functions needed by the above packages.

Within the design of my solution the record protocol was not affected by any modifications, so the `record` directory stays unchanged. The `statem` directory is organized hierarchically like in Figure 3.24: the file `statem.c` implements the TLS handshake through two state machines: the first one is the message flow state machine that controls the reading and sending of messages including handling of non-blocking I/O events, handling of unexpected messages, etc. It is itself broken into two separate sub-state machines in charge of reading and writing messages respectively. The second one is the handshake state machine that keeps track of the current TLS handshake state that will recall client-specific handshake state machine code (`statem_clnt.c`) and server-specific handshake state machine code (`statem_srvr.c`). The functions that the two just mentioned files have in common are stored separately into an additional file named `statem_lib.c`.

Instead of making changes to the original OpenSSL files, wherever it was possible I decided to create new files from scratch to implement the handshake additional features and then include them in the appropriate section, to keep them separate from the original ones. In the statem directory I added one header file (`statem_local_did.h`) and one source file (`statem_local_did.c`). The first one declares the client and server methods to construct and process the additional DID handshake messages, while the source file implements them. Figure 3.25 proposes file hierarchy that results into the TLS handshake state machine.

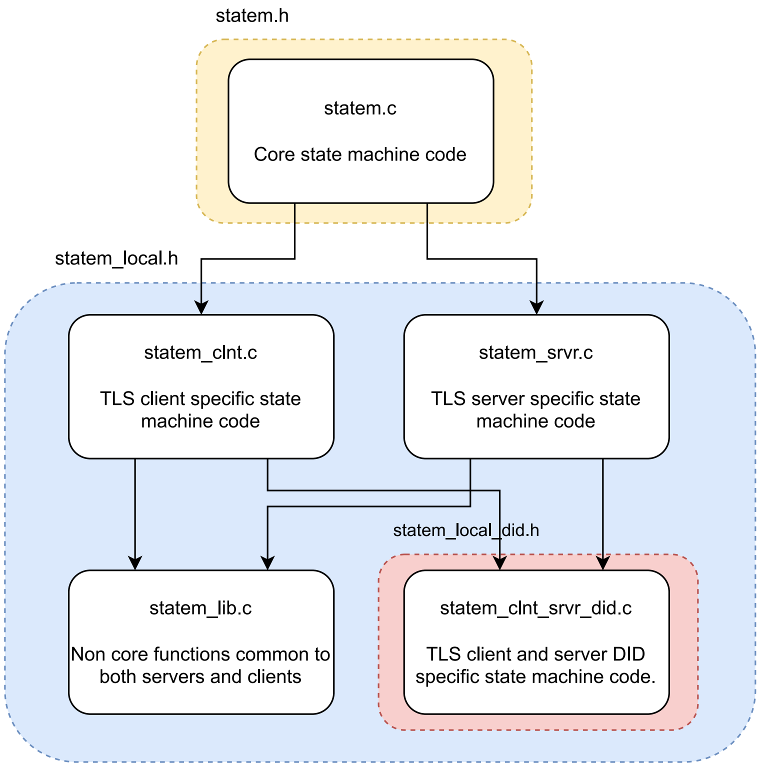

statem.h
statem.c
Core state machine code
statem_local.h
statem_clnt.c
TLS client specific state machine code
statem_srvr.c
TLS server specific state machine code
statem_local_did.h
statem_lib.c
Non core functions common to both servers and clients
statem_clnt_srvr_did.c
TLS client and server DID specific state machine code.

Figure 3.25: File hierarchy in `statem` directory in SSI-aware OpenSSL.

Among the general files contained in the `ssl` library there is the `ssl_local.h` file that supplies the essential structures and functions needed to initialize and hold the current status of the handshake context. In order to expand the TLS handshake context to make it SSI-compliant I decided to attach DID structures to this file and then provide two additional files: one header (`ssl_local_did.h`) and one source (`did.c`) that offer functions to manage the new DID structures that complete the handshake context.

3.7 WAM

Links Foundation have set up a node for the IOTA Chrysalis Tangle and developed the WAM (**Wrapped Authenticated Message**) protocol, to assist constrained IoT devices in structuring and navigating data on the IOTA Tangle, as the IOTA Foundation provides protocols only for fully-featured devices. The WAM protocol utilizes the Chrysalis message payload named the *indexation payload* to anchor data to the Tangle. The indexation payload consists of an index along with additional data (i.e. application data). All WAM protocol messages are therefore encapsulated within the indexation payload (Figure 3.26).

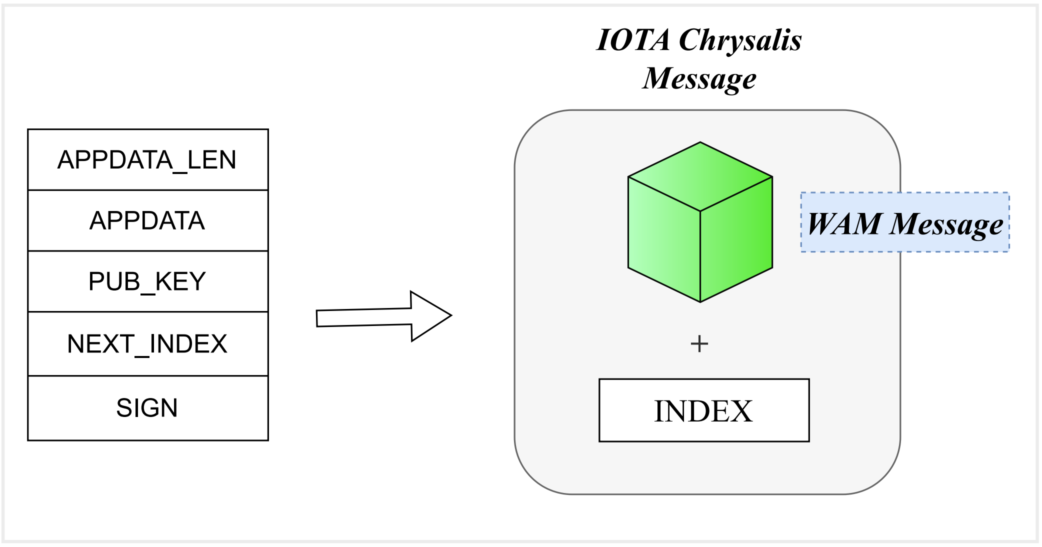


Figure 3.26: fields of a WAM message (**left**) and IOTA Chrysalis message (**right**).

WAM organizes data streams as a single-chain on the Tangle, with each piece of data in the stream linked to the next one through an index. This allows any subscriber of the data stream to reconstruct it by starting at any index on the Tangle and following the chain of linked data. Each data message includes the current index and the index of the next message. Additionally, WAM adds a security layer by using Authenticated Encryption with Associated Data (AEAD) to protect the data, which is anchored to the Tangle in plain-text. Higher-level protocols or applications can include data in the APPDATA field and its length in APPDATA_LEN field. For data that exceeds the maximum data length of a single WAM message or for continuous data transmission, multiple data messages must be linked. This is done through the NEXT_IDX field, which contains the index of the next message in the stream to look for on the Tangle. Figure 3.27 illustrates chaining mechanism.

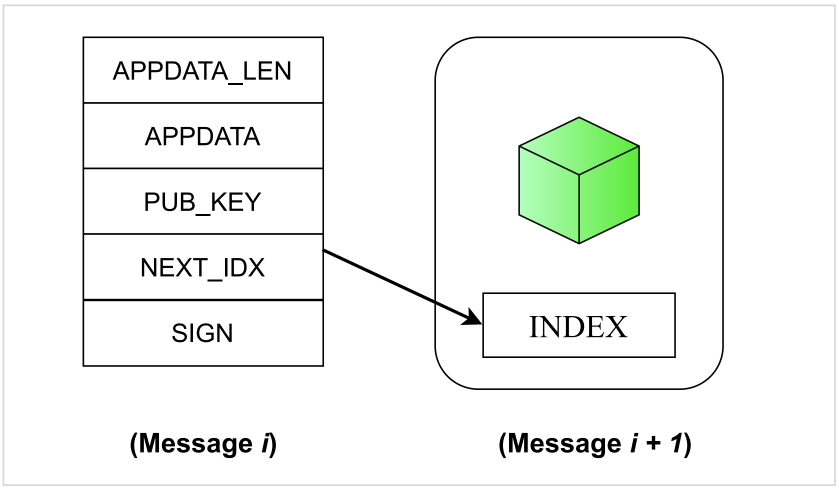


Figure 3.27: Chain of WAM messages.

By linking messages in a single chain, the WAM protocol allows subscribers to read data streams in only one direction. This is intentional as it prevents the retrieval of previous information from the same data stream. Figure 3.28 illustrates the process for generating the INDEX and NEXT_IDX. The WAM protocol generates a secret key and the corresponding public key deterministically from a random seed, using the edwards25519 curve. The index of a WAM message is then calculated by applying a hash function to the public key. Additionally, each WAM message includes a link to the next index (NEXT_IDX) to facilitate the continuous flow of data. Also the NEXT_IDX is generated in the same manner, using a different key-pair.

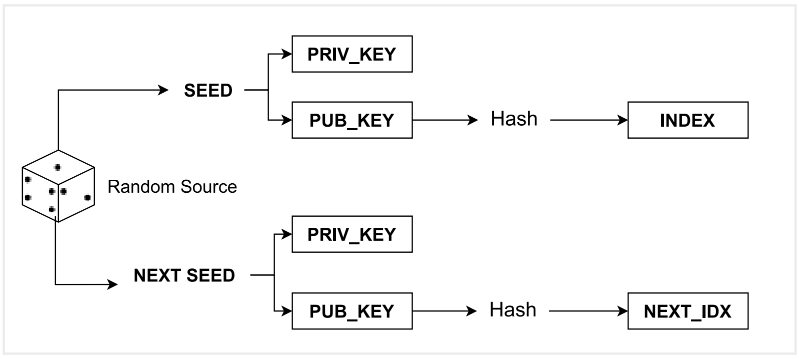


Figure 3.28: Generation of INDEX and NEXT_IDX fields.

Each message includes a field called SIGN, which is calculated by using the private key (PRIV_KEY) of the key-pair to sign a digest h, as shown in the following equations.

$$h = H(APPDATA\_LEN + APPDATA + PUB\_KEY + NEXT\_IDX)$$

$$SIGN = signature(h|PRIV\_KEY)$$

It's important to note that an attacker who wants to redirect the next message to a different, malicious stream would need to discover the public key used to derive the next index and include it in their message. On the receiver side, the signature and public key are used for verification (Figure 3.29), ensuring the recipient cannot use the discovered NEXT_IDX to append their own chain of messages because they do not have the key-pair used to derive the NEXT_IDX. This design allows subscribers to confirm that data is coming from the same source. The WAM protocol employs EdDSA (Edwards-curve Digital Signature Algorithm) over the edwards25519 elliptic curve and uses the BLAKE2b hashing algorithm for integrity signature.

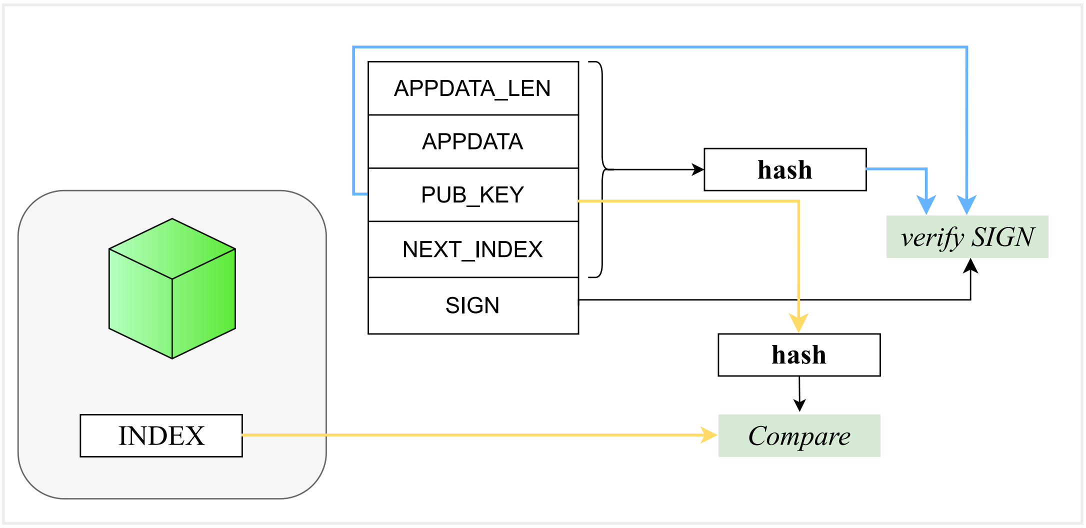


Figure 3.29: Verification of a WAM message.

However WAM manages raw data, writes/reads bytes to/from the Tangle without assigning them a particular meaning. The *DID method* library, that lies on top of WAM, has been developed to structure this data and thus be able to perform CRUD operations on DID documents over the IOTA ledger. The DID method library provides the following four methods: *Create*, *Resolve*, *Update*, *Delete*.

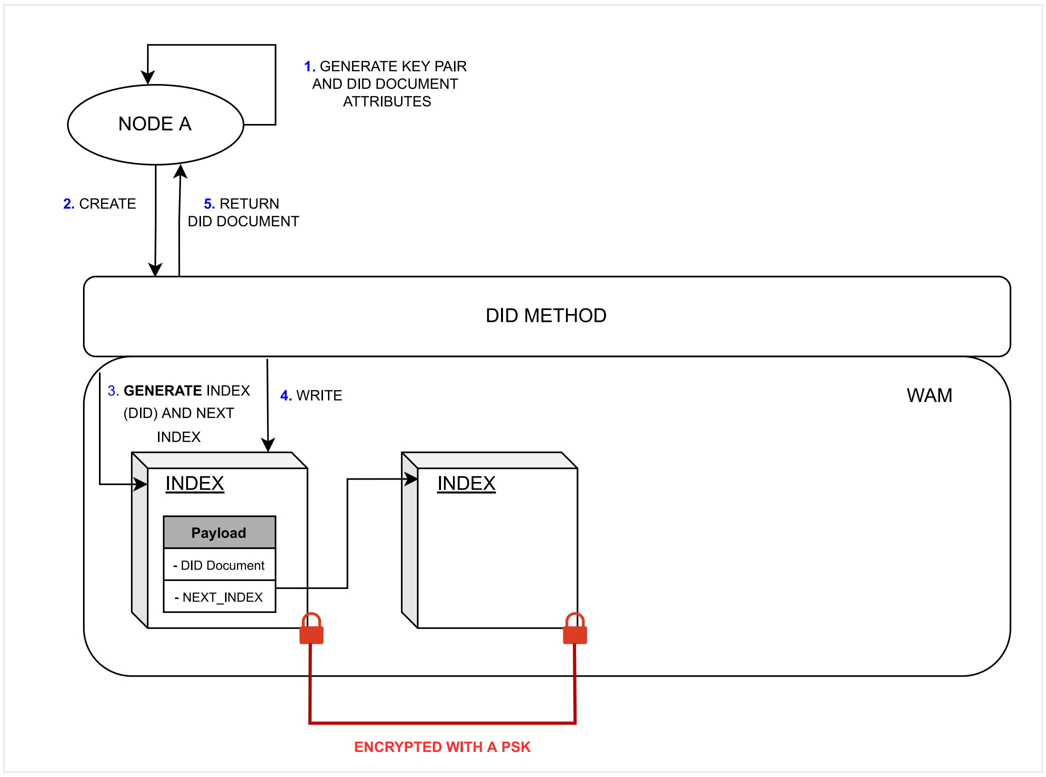


Figure 3.30: DID Method - Create method.

Figure 3.30 illustrates the process of creating a DID document. Initially a node must generate an asymmetric keypair plus all the attributes it wants to include in the document, then invokes the *Create* method of the DID method interface which will internally call WAM functions to anchor the DID document on the ledger. In this way a new data stream must be generated, so INDEX and NEXT_IDX are produced to identify the first two blocks of the chain and the INDEX field corresponds to the DID of the DID document that will be written at the INDEX block. After writing all the data to the ledger the DID document will be returned to the node (Figure 3.30).

Whenever a node wants to apply changes to a DID document the DID method library makes available the *Update* method that requests in input the DID and the changes to be made. The DID is needed to find the index of the block where the DID document is stored, then the current DID document is extracted, a new one is generated, containing the desired changes, and saved at the NEXT_IDX block. Logically, the updated DID document will also have a different DID that corresponds to the NEXT_IDX field. Furthermore, any time a new block is populated a new NEXT_IDX must be generated. Figure 3.31 illustrates all these steps.

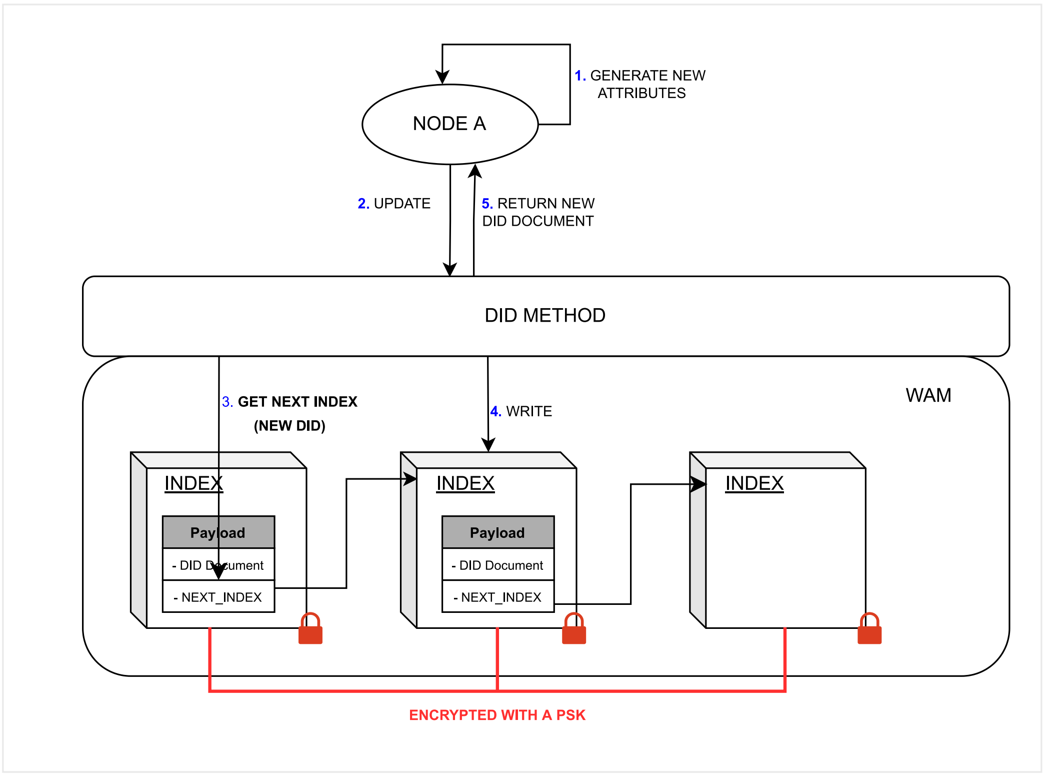


Figure 3.31: DID Method - Update method.

In the situation where node B receives a DID from node A and tries to resolve it, invokes the *Resolve* method of the DID method library. The DID must correspond to the most recent version of the associated DID document, i.e. should be the index of the last non-empty block in the chain, otherwise the resolve operation will fail. The WAM library will internally perform this check and, if it succeeds, reads the payload of the block and hands it back to the above DID method library which will organize this data into a DID document and returns it to node B, otherwise an error code will be returned. Everything is shown in Figure 3.32.

The *Revoke* operation (Figure 3.33) supported by the DID method library provides the functionality to dismiss a DID document by accepting as input parameter its DID. One of the main features of the IOTA Tangle, but more in general of the DLT, is the data immutability property which does not allow the removal of data from the ledger. In order to manage this, WAM will first individuate the block of the chain that contains the DID document to revoke and sets NEXT_IDX to the value zero. So the next time a node tries to resolve the DID will realize that is no more valid by checking that the last block of the chain has the IDX value set to zero.

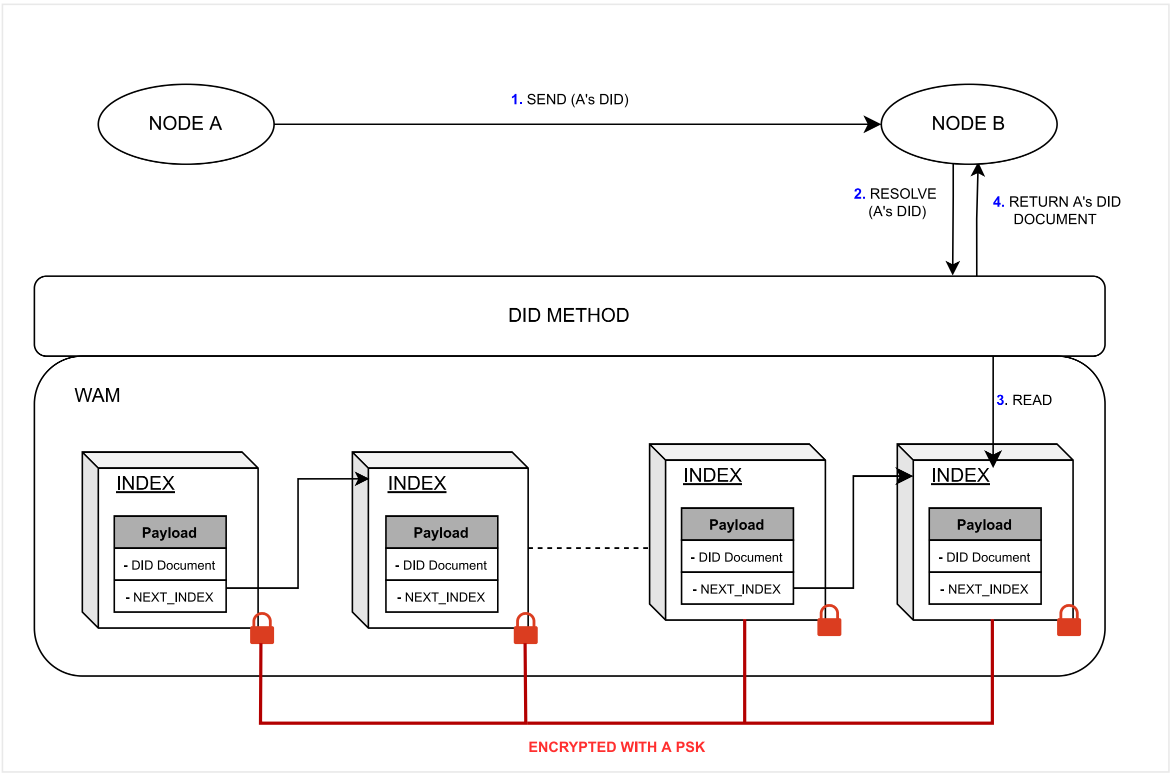


Figure 3.32: DID Method - Read method.

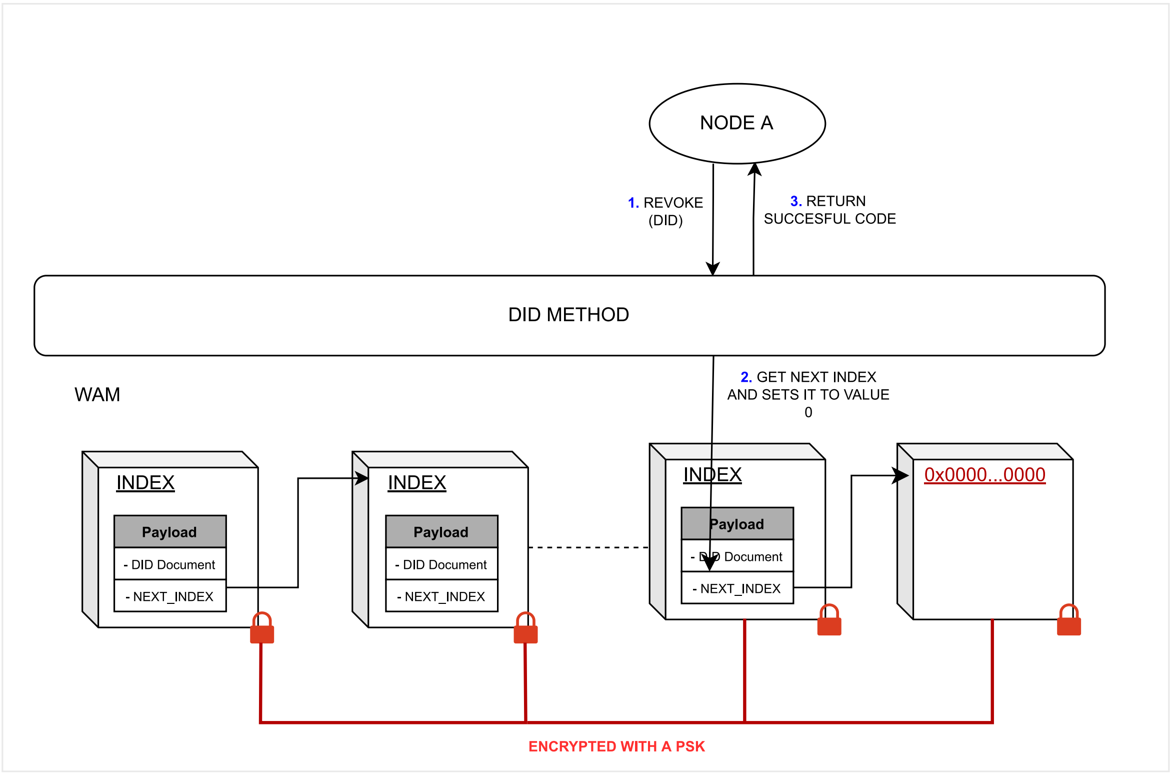


Figure 3.33: DID Method - Revoke method.

3.8 Security Assessment

The modifications the TLS handshake protocol has undergone for enabling the SSI paradigm must not invalidate the level of security and robustness the original one had. Thus, I have designed two theoretical MITM (man-in-the-middle) attacks to prove its reliability starting from a couple of assumptions:

- the attacker does not have a valid DID;
- if they do, the handshake endpoints will still be protected by the DID Method PSK;

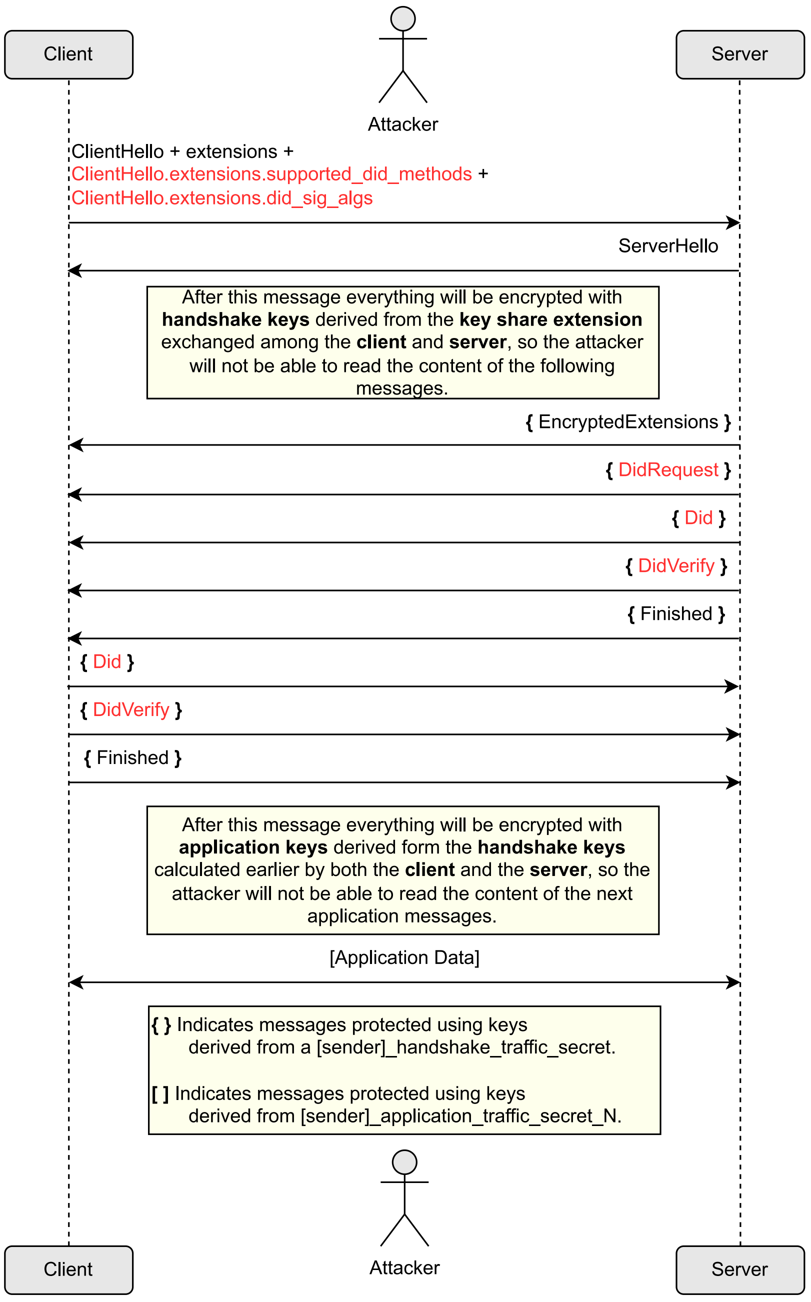
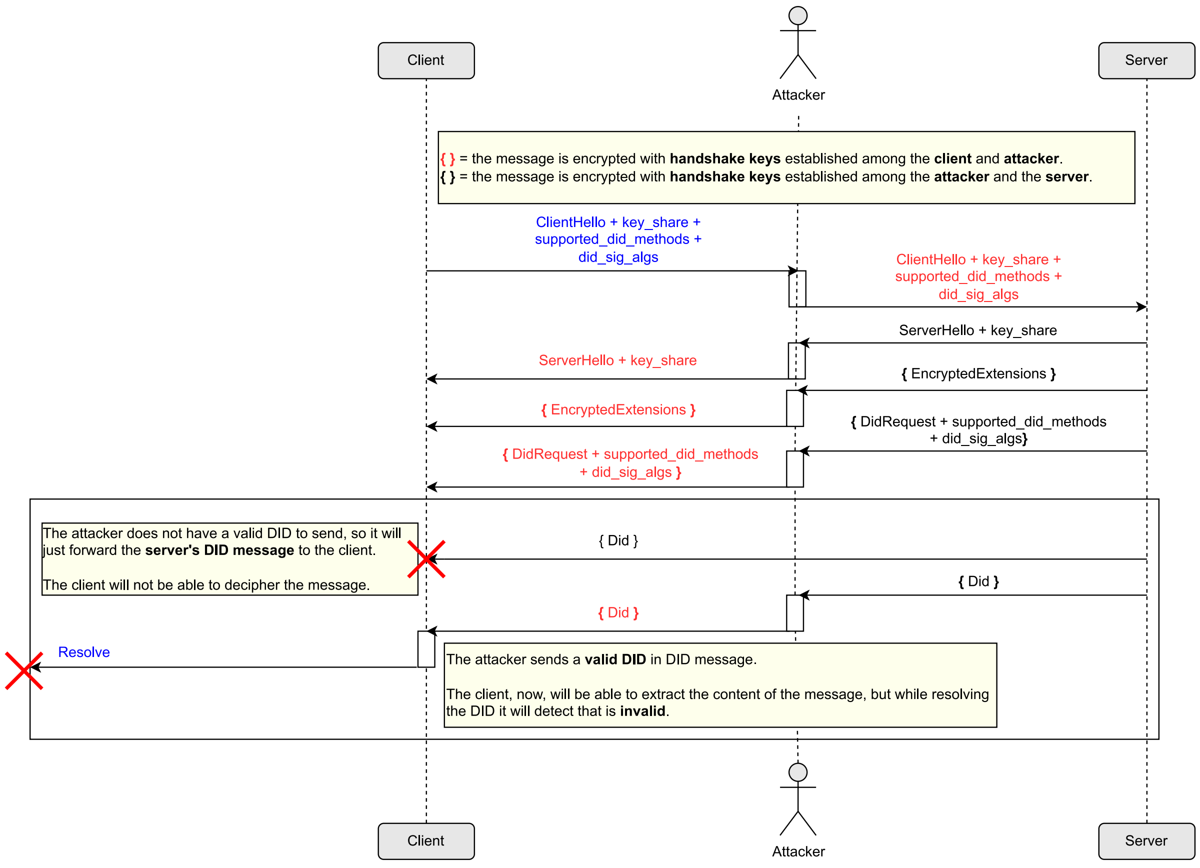

Client
Attacker
Server
ClientHello + extensions +
ClientHello.extensions.supported_did_methods +
ClientHello.extensions.did_sig_algs
ServerHello
After this message everything will be encrypted with **handshake keys** derived from the **key share extension** exchanged among the **client** and **server**, so the attacker will not be able to read the content of the following messages.
{ EncryptedExtensions }
{ DidRequest }
{ Did }
{ DidVerify }
{ Finished }
{ Did }
{ DidVerify }
{ Finished }
After this message everything will be encrypted with **application keys** derived form the **handshake keys** calculated earlier by both the **client** and the **server**, so the attacker will not be able to read the content of the next application messages.
[Application Data]
{ } Indicates messages protected using keys derived from a [sender]_handshake_traffic_secret.
[ ] Indicates messages protected using keys derived from [sender]_application_traffic_secret_N.
Client
Attacker
Server

Figure 3.34: Transparent Proxy attack in SSI-aware TLS handshake.

The first attack, denominated **MITM - Transparent Proxy** supposes the attacker is passive, that is it can only sniff messages the two endpoints exchange, but they can't forge new ones. As Figure 3.34 shows, after the Server Hello the subsequent messages exchanged are encrypted with handshake keys established among client and server in Client Hello and Server Hello by employing the *key exchange* extension. In fact in the previous two messages both parties produced an asymmetric keypair and shared with the other only the public part. Handshake keys are derived by feeding to some internal algorithm the own private key and the peer's public key. In this way the attacker will never be able to access the private keys of the two endpoints as they will never be sent over the network. Furthermore, after the client's Finished message the application data will be ciphered with application keys derived from handshake keys to which the attacker did not have access previously, so the communication can be treated as secure.

In the **MITM - Active Proxy** attack, the attacker intercepts the messages exchanged by the client and the server, manipulates them and sends them. In Figure 3.35 the blue messages are the ones sent by the client, the red ones are sent by the attacker and the black ones are sent by the server. The Client Hello gets intercepted by the attacker which discards it and sends its own Client Hello to the server. The same happens for the Server Hello. Now the attacker and the server have established their handshake keys (we reference them with black curly braces) which are different from the ones established by the client and the attacker (represented by the red curly braces). Whenever the Encrypted Extensions and Did Request messages are sent they will be blocked by the attacker which will send its version of these messages encrypted with handshake keys that it previously established with the client. Consequently, at the time the server sends the Did message two different scenarios unfolds: in the first one the attacker does not possess a valid DID to send and will simply forward the DID message from the server to the client, but the latter will not be able to decrypt it as it is encrypted with handshake keys it does not possess. In the second case the attacker discards the server's Did message and forges a new one to send to the client. The latter now will be able to extract the content of the message but when it will try to resolve it will detect that the DID does not belong to the channel that its PSK can access.


Client
Attacker
Server
ClientHello + key_share + supported_did_methods + did_sig_algs
ClientHello + key_share + supported_did_methods + did_sig_algs
ServerHello + key_share
ServerHello + key_share
{ EncryptedExtensions }
{ EncryptedExtensions }
{ DidRequest + supported_did_methods + did_sig_algs}
{ DidRequest + supported_did_methods + did_sig_algs }
{ Did }
{ Did }
{ Did }
Resolve
Client
Attacker
Server

{ } = the message is encrypted with **handshake keys** established among the **client** and **attacker**.
{ } = the message is encrypted with **handshake keys** established among the **attacker** and the **server**.

The attacker does not have a valid DID to send, so it will just forward the **server's DID message** to the client.

The client will not be able to decipher the message.

The attacker sends a **valid DID** in DID message.

The client, now, will be able to extract the content of the message, but while resolving the DID it will detect that is **invalid**.

Figure 3.35: MITM - Active Proxy attack in SSI-aware TLS handshake.

Chapter 4

Results

4.1 Functional tests

After designing and implementing the new SSI-aware TLS handshake model, I performed some functional tests on a local machine to validate the solution. Below I will list all the scenarios where the TLS handshake protocol should succeed in OpenSSL. In all the following cases both server and mutual authentication are considered.

- a client employing the original OpenSSL and a server employing the SSI-aware OpenSSL must be able to complete a handshake when using X.509 certificates.
- a client employing the SSI-aware OpenSSL and a server employing the original OpenSSL must be able to complete a handshake when using X.509 certificates.
- a client and a server both employing the SSI-aware OpenSSL must be able to complete a handshake when using X.509 certificates.
- a client and a server both employing the SSI-aware OpenSSL must be able to complete a handshake when using DIDs.
- a client and a server both employing the SSI-aware OpenSSL must be able to complete a handshake when the client authenticates through a DID and the server through an X.509 certificate.
- a client and a server both employing the SSI-aware OpenSSL must be able to complete a handshake when the client authenticates through an X.509 certificate and the server through a DID.

4.2 Performance tests

Afterwards I ran some performance tests, employing the SSI-aware OpenSSL, to measure the handshake time over different authentication scenarios. I installed the SSI-aware OpenSSL on two Raspberry Pi's model 4 in the laboratory and connected them through an Ethernet switch.

The times were taken both on client and server side. The client starts measuring time before sending the Client Hello message, thus before initiating a TLS connection, and stops the clock after sending the Finished message because that is when can start sending application data. On the other hand, a server starts the clock at the receiving of a Client Hello message, avoiding the first network trip, but it will have to process all the client's messages before stopping the time. In fact the server completes the handshake only after processing the Finished message received from the client. Anyway, for each scenario I will show a diagram of the interval of time measured by both the client and the server.

In the results I will discuss below, it is worth noting that the Round Trip Time among the two RPI's was, on average, **0,238 ms**. For each authentication scenario I ran 100 tests and considered the average time. All the DID documents and X.509 certificates I have utilized used an RSA-2048 keypair and an RSA-PSS-SHA256 algorithm for signatures.

4.2.1 Certificate Authentication

In this first scenario both parties are only allowed to authenticate with X.509 certificates. Times were taken both in a mutual authentication handshake and a server authentication handshake. Figure 4.1 shows the operations involved in the interval of times that have been measured in both situations.

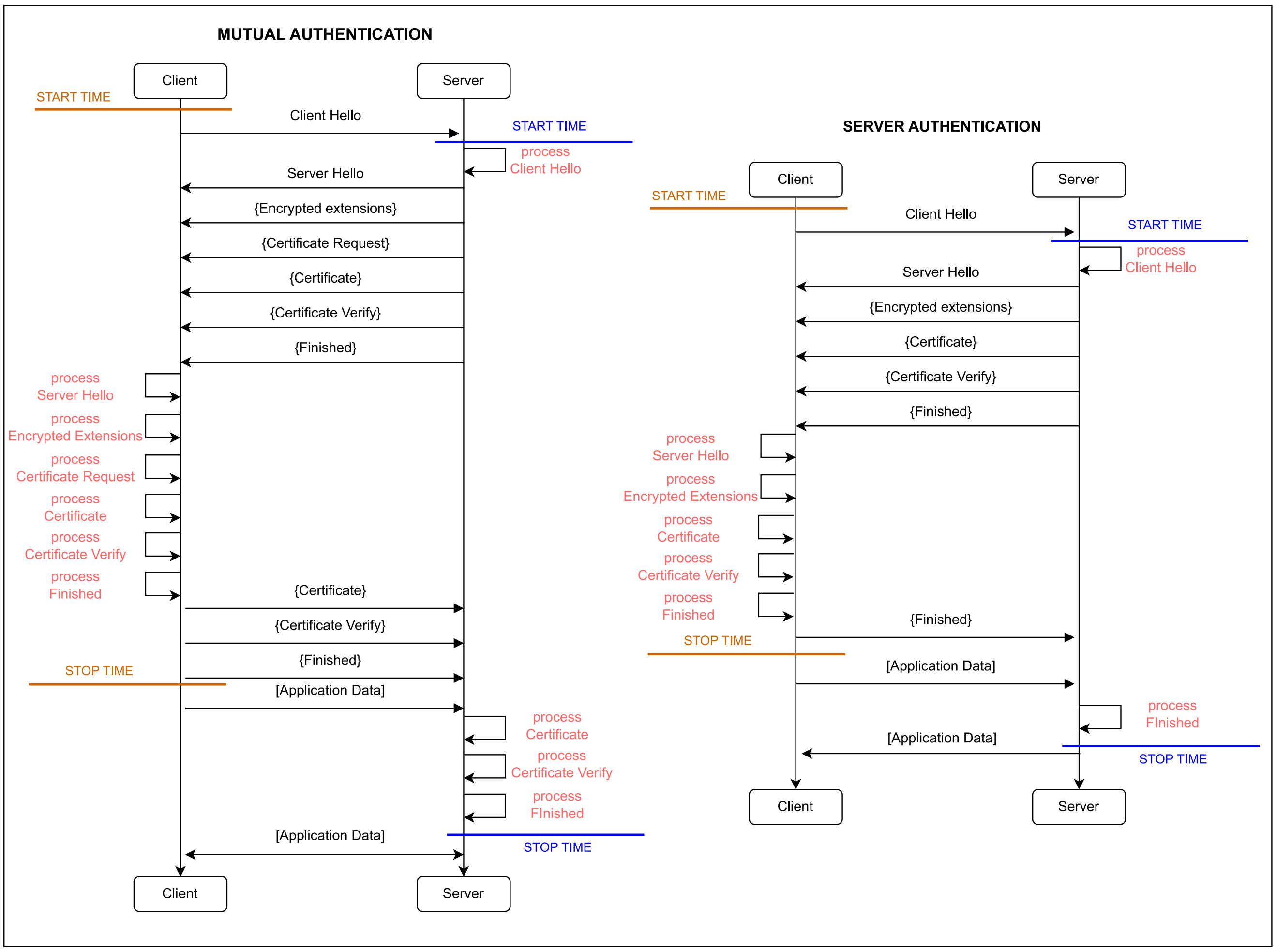
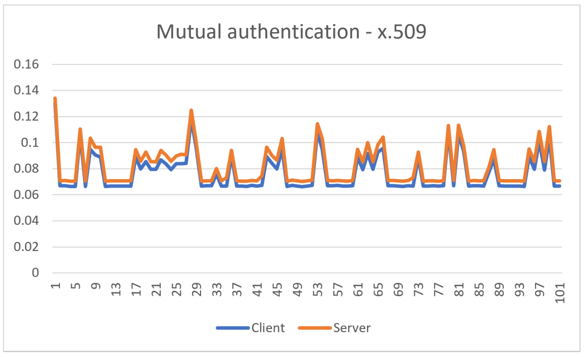


Figure 4.1: Operations included in the interval of time measured by the Client and the Server in a mutual authentication (**left**) and server authentication (**right**) scenario in the TLS handshake protocol with X.509 certificates.

In the server authentication chart (Figure 4.2) the two graphs almost overlaps as the average time of the handshake on client side is **57,74 ms** while on server side is **57,93ms**. On the mutual authentication chart (Figure 4.2) the client's average handshake time takes **76,94 ms**, while the server's is **81,93 ms**, which is slightly greater due to the processing of Client Certificate and Certificate Verify messages.



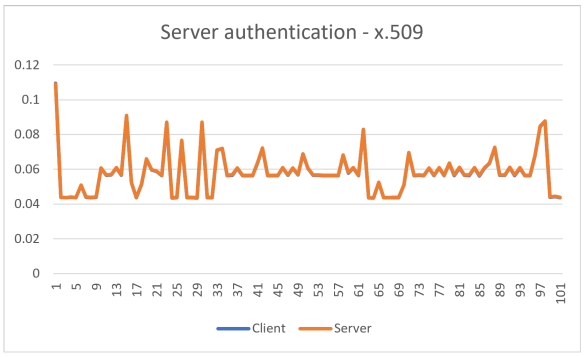


Figure 4.2: Client and Server time graph of TLS handshake with X.509 certificates. Mutual authentication (**left**) and Server authentication (**right**).

4.2.2 DID Authentication

In the second scenario authentication is performed through DIDs. I have created two DID documents, employing the DID method library, one for the server and one for the client on the IOTA Devnet Tangle (a public ledger). In all the tests I have run all the connections to the Devnet ledger are TLS-free, since data are already encrypted by the WAM protocol.

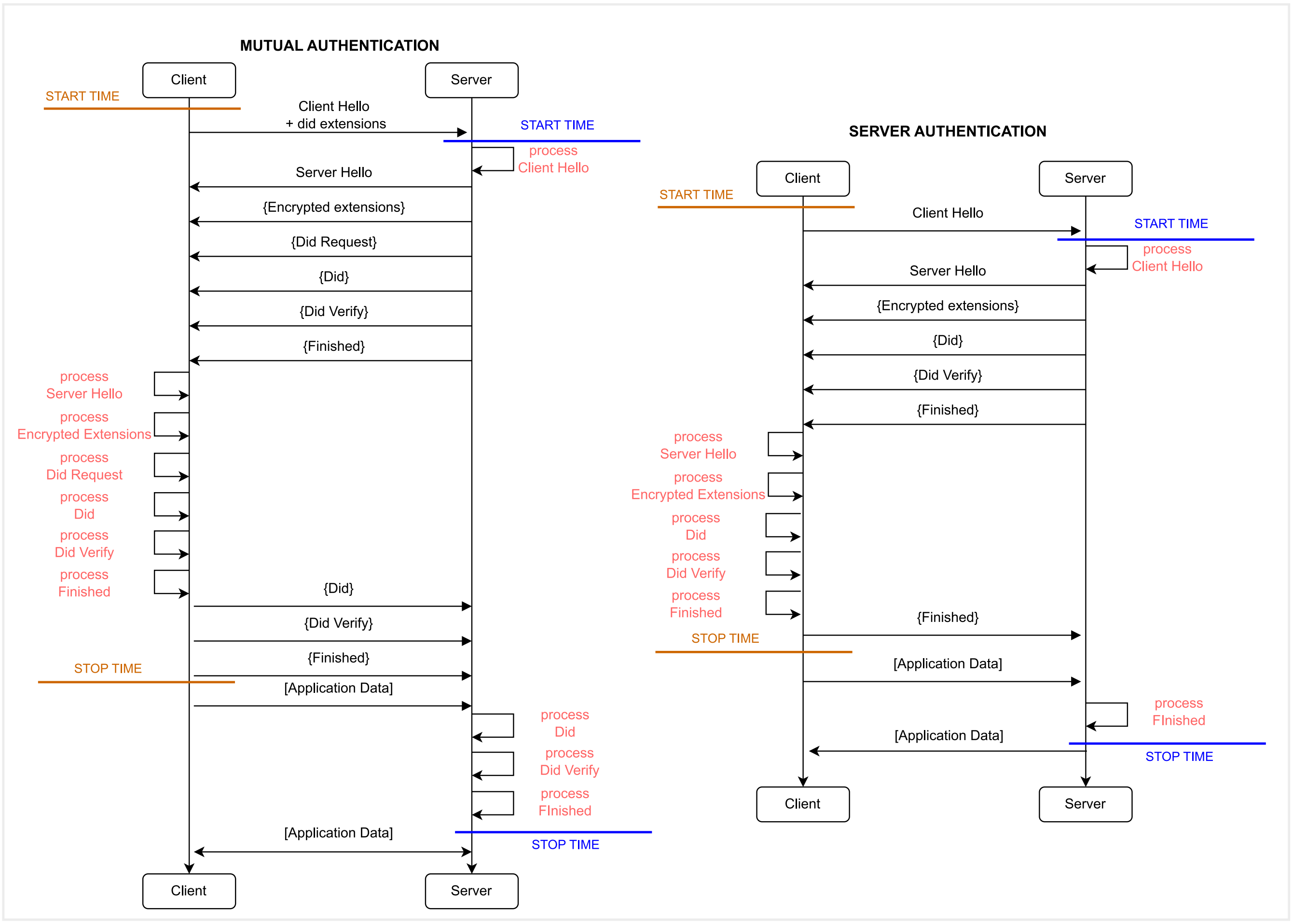


Figure 4.3: Operations included in the interval of time measured by the Client and the Server in a mutual authentication (**left**) and server authentication (**right**) scenario in the TLS handshake protocol with DIDs.

Both server and mutual authentication (Figure 4.4) with DIDs are an order of magnitude greater with respect to the certificate authentication scenario, due to the resolve operation of a DID document on the IOTA ledger. In particular in a server authentication scenario the average handshake time on client side is **385,36 ms**, very similar to the one on server side which is **385,54 ms**. This is due the fact that both the server and the client in their time interval will have to wait only for one resolve operation, while in the case of a mutual authentication the server's times on the chart almost doubles the client's ones. This is because the server during its interval will have to wait for the client's resolve operation of its DID document and then himself perform a resolve operation of the client's DID, whereas the client after sending its Finished message stops the counter. The average handshake time on client's side is **416,74 ms**, while on server's side is **738,80 ms**.

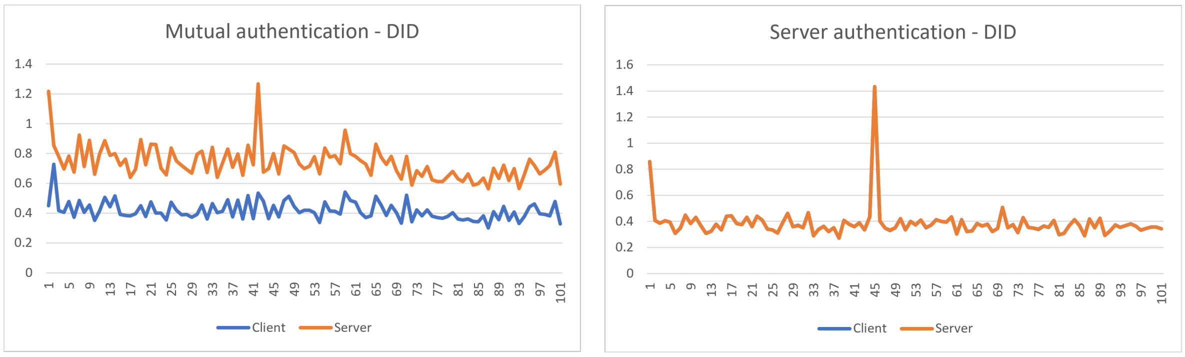


Figure 4.4: Client and Server time graph of TLS handshake with DIDs. Mutual authentication (**left**) and Server authentication (**right**).

4.2.3 Cross Authentication

Cross authentication is always mutual, but the two endpoints uses different authentication mechanisms.

Client-X.509 & Server-DID

Figure 4.5 illustrates the operations included in the time intervals where the client authenticates with an X.509 certificate and the server with a DID. The chart (Figure 4.6) shows that the handshake times are similar on client and server side, with an average of **476,71 ms** for the client and **484,28 ms** for the server, due to the fact the resolve operation performed by the client of the server's DID involves both endpoints.

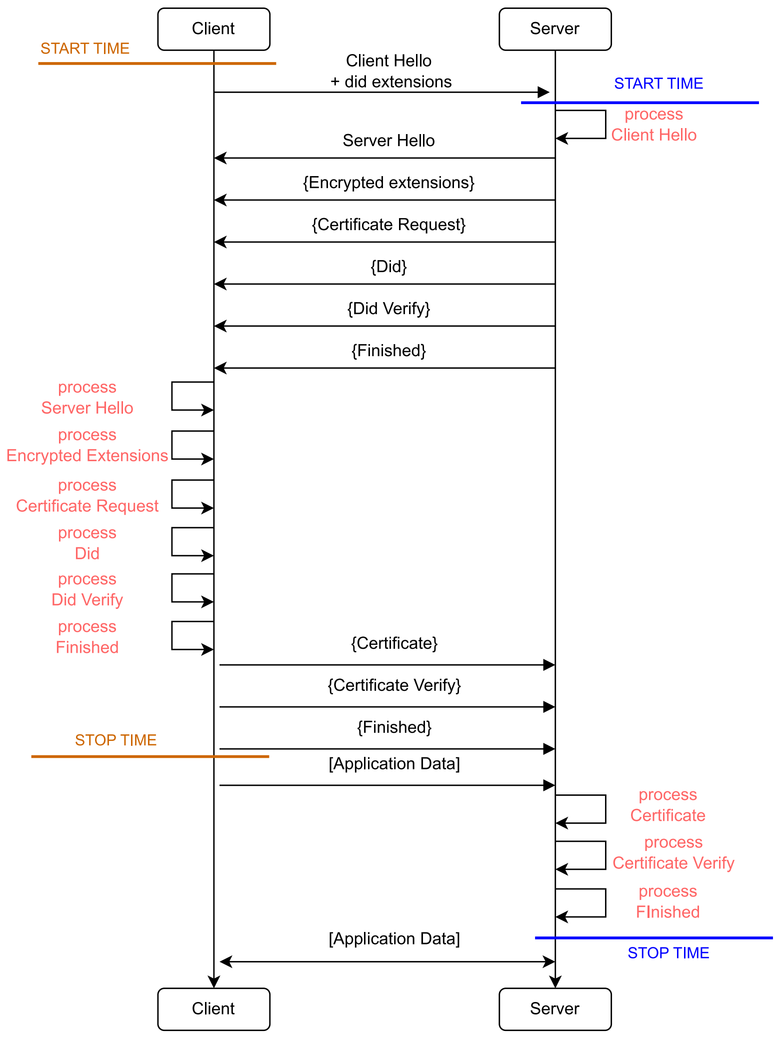


Figure 4.5: Operations included in the interval of time measured by the Client and the Server in a cross-authentication (Client-X.509 & Server-DID) scenario in the TLS handshake protocol.

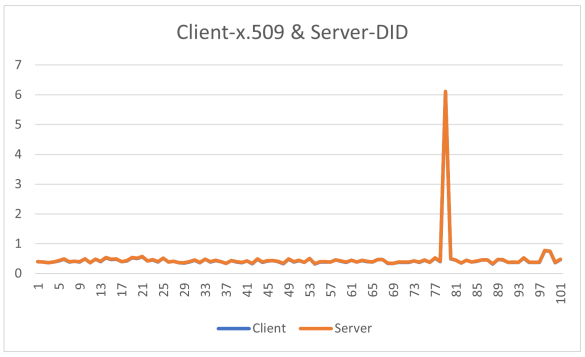


Figure 4.6: Time graph of Client:X.509-Server:DID authentication handshake.

Client-DID & Server-X.509

In the second case of cross authentication the client authenticates with a DID and the server with an X.509. The chart (Figure 4.8) shows quite different results between the client and the server. The client now will not perform any resolve operation on the IOTA ledger because it receives an X.509 certificate from the server, while the server receives the client DID and performs the resolve of the DID document. The average time on the client's side is **95,12 ms**, on the server is **435,84** ms.

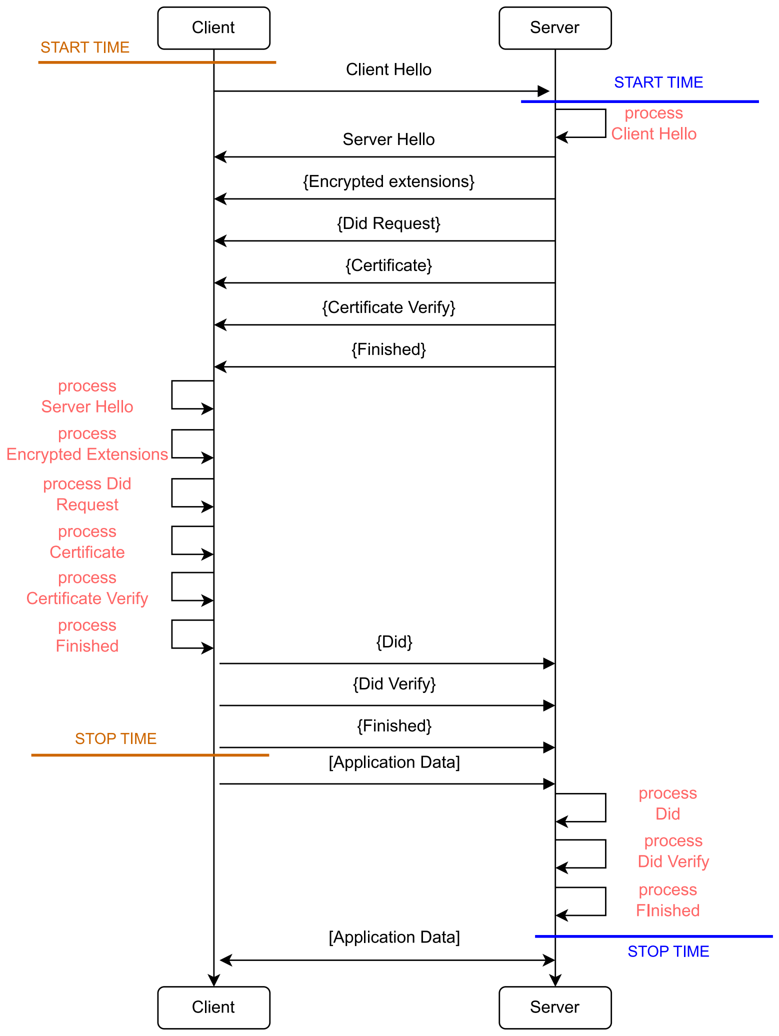


Figure 4.7: Client and Server time interval in a cross-authentication (Client-DID & Server-X.509) scenario in the TLS handshake protocol.

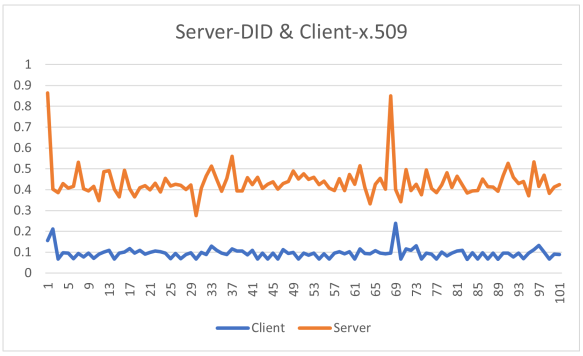


Figure 4.8: Time graph of Client:DID-Server:X.509 authentication handshake.

Chapter 5

Conclusions

In conclusion, the goal of this work was to introduce the Self-Sovereign Identity paradigm in the TLS handshake protocol and demonstrate how to get rid of centralised Certification Authorities in the handshake protocol, by replacing them with self-owned Decentralized Identifiers. In the future the WAM protocol should be improved to catch up with the time performances of the original TLS handshake protocol, so as all the protocols that will be designed to interact with the different DLT's. Furthermore, we know in the original handshake protocol an endpoint rarely consults, at the time of the handshake, the CRL or the OCSP server to validate the peer's certificate chain, but most likely resorts to shortcut techniques such as pushed CRL or OCSP stapling. In the SSI-aware TLS handshake similar extensions could be designed to reduce to the minimum the authentication time. In the end since DIDs are self-issued, the identity of a party cannot be proved. In fact this solution works for small groups of actors who trust each other by sharing a PSK (like in WAM) or something similar. Thus the ultimate goal of this work will be to integrate VCs with DIDs to guarantee the authentication property of TLS even in an untrusted environment.

Appendix A

User Manual

This section illustrates how to install and execute the code of the DID Method library developed for the IOTA Tangle and the SSI-aware version of OpenSSL 3.0.

A.1 DID Method

The DID Method library provides methods to *create*, *update*, *resolve* and *revoke* DID documents.

A.1.1 Requirements

The DID Method requires the following dependencies to work properly.

cJSON

cJSON is an ultra-lightweight library used to parse and generate JSON objects. It is written in ANSI C to support a wide variety of platforms and compilers and it consists of a single source file (cJSON.c) and header file (cJSON.h) which can be directly downloaded and included into a project [12].

iota.c

It is the iota C client library needed to interact with the IOTA ledger and will be employed by WAM [13].

WAM

WAM is the cryptographic protocol utilized to securely read and write data to the IOTA Chrysalis Tangle.

libsodium

Sodium is a cross-platform, cross-language and easy-to-use cryptographic library which can be installed on linux platforms with the command [14]:

```
sudo apt install libsodium-dev
```

A.1.2 Installation

The DID Method library is stored on the git server at Links Foundation and can be downloaded with:

```
git clone git-guest@gitserver:/git/GUEST/C_CRUD.git
```

To install it we enter the new directory and we launch the bash script:

```
cd C_CRUD
./build.sh
```

The `build.sh` script (figure A.2) contains the necessary instructions to output the static library `libdidmethod.a` and a `./demo/demo` application to play around with CRUD operations on the IOTA Chrysalis Tangle. The script initially clones from github the `dev` branch of the iota.c repository then builds it and installs it in the local system. Furthermore it downloads the WAM library from the Links Foundation's git server and in the end the `cmake` tool executes the `CMakeLists.txt` file (Figure A.1) to generate the `MakeFile` that outputs the final library.

```
cmake_minimum_required(VERSION 3.9)
project(wam)
include_directories(
       ${CMAKE_CURRENT_SOURCE_DIR}/WAM
       ${CMAKE_CURRENT_SOURCE_DIR}/iota.c/build/include/
       ${CMAKE_CURRENT_SOURCE_DIR}/iota.c/build/include/cjson/
       ${CMAKE_CURRENT_SOURCE_DIR}/iota.c/build/include/client/
       ${CMAKE_CURRENT_SOURCE_DIR}/iota.c/build/include/crypto/
       ${CMAKE_CURRENT_SOURCE_DIR}/iota.c/build/include/core/
)
link_directories(${CMAKE_CURRENT_SOURCE_DIR}/iota.c/build/lib/)
add_library(didmethod STATIC did_method.c
    ${CMAKE_CURRENT_SOURCE_DIR}/WAM/WAM.c
    ${CMAKE_CURRENT_SOURCE_DIR}/WAM/wam-wrapper.c)
target_link_libraries(didmethod PUBLIC iota_crypto crypto iota_core
    iota_client cjson curl sodium)
set_property(TARGET didmethod PROPERTY POSITION_INDEPENDENT_CODE ON)
add_subdirectory(demo)
```

Figure A.1: Content of the `CMakeLists.txt` file

```
#!/bin/bash
set -e
DIR="./iota.c"
if [ -d "$DIR" ]; then
       # Take action if $DIR exists. #
       echo "iota.c already present -> skip download"
else
       echo "Downloading IOTA C Library from Git..."
       git clone -b dev https://github.com/iotaledger/iota.c.git
       echo "Building IOTA C Library..."
       cd iota.c
       mkdir build && cd build
       cmake -DCMAKE_C_COMPILER=gcc -DCMAKE_CXX_COMPILER=g++
           -DCryptoUse=libsodium -DIOTA_WALLET_ENABLE:BOOL=TRUE
           -DCMAKE_INSTALL_PREFIX=$PWD -DWITH_IOTA_CLIENT:BOOL=TRUE
           -DWITH_IOTA_CORE:BOOL=TRUE ..
       make all
       make install
       cd ..
       cd ..
fi
DIR="./WAM"
if [ -d "$DIR" ]; then
       # Take action if $DIR exists. #
       echo " WAM already present -> skip download"
else
       echo "Downloading WAM Library from Git..."
       git clone git-guest@gitserver:/git/GUEST/WAM.git
fi
echo "Building DID Method..."
cmake .
make
echo ''Finished''
```

Figure A.2: Content of the `build.sh` script

A.2 SSI-aware OpenSSL 3.0

A.2.1 Installation on UNIX-like platforms

The SSI-aware version of OpenSSL can be downloaded from the Links Foundation's git server by executing:

```
git clone git-guest@gitserver:/git/GUEST/OpenSSL.git
```

then we run the script that generates the `MakeFile` (Figure A.3) needed to build OpenSSL by running the following commands:

```
cd OpenSSL
./openssl-config.sh
```

In `openssl-config.sh` the `--prefix` option lets the user choose the installation directory of the OpenSSL executable and libraries.

Now we are ready to build and install the new version of OpenSSL with:

```
make
make install
```

```
./config --debug --prefix=path/to/openssl
    -Wl,--enable-new-dtags,-rpath,'$(LIBRPATH)' -I/path/to/C_CRUD/WAM
    -I/path/to/C_CRUD/iota.c/build/include
    -I/path/to/C_CRUD/iota.c/build/include/cjson
    -I/path/to/C_CRUD/iota.c/build/include/client
    -I/path/to/C_CRUD/iota.c/build/include/crypto
    -I/path/to/C_CRUD/iota.c/build/include/core -I/path/to/C_CRUD
    -L/path/to/C_CRUD/iota.c/build/lib/ -L/path/to/C_CRUD -ldidmethod
    -liota_crypto -liota_core -liota_client -lcjson -lcurl -lsodium
```

Figure A.3: Content of the `MakeFile`

A.2.2 Usage

OpenSSL provides the `openssl` command line tool to perform various cryptographic operations such as performing a client/server TLS handshake. In order to accomplish this a user should navigate to the directory where they installed OpenSSL:

```
cd path/to/openssl
```

In the `/bin` subdirectory the user will find the `openssl` executable file and can run their desired application providing. Below we will check out some `openssl` configurations to perform the TLS handshake under several authentication scenarios. Let's also suppose that an endpoint's identity information, such as the X.509 certificate and its private key, are stored in the `/bin` folder and both the TLS client and server will run on a local system.

TLS 1.3 Handshake with server authentication - x.509

Server side

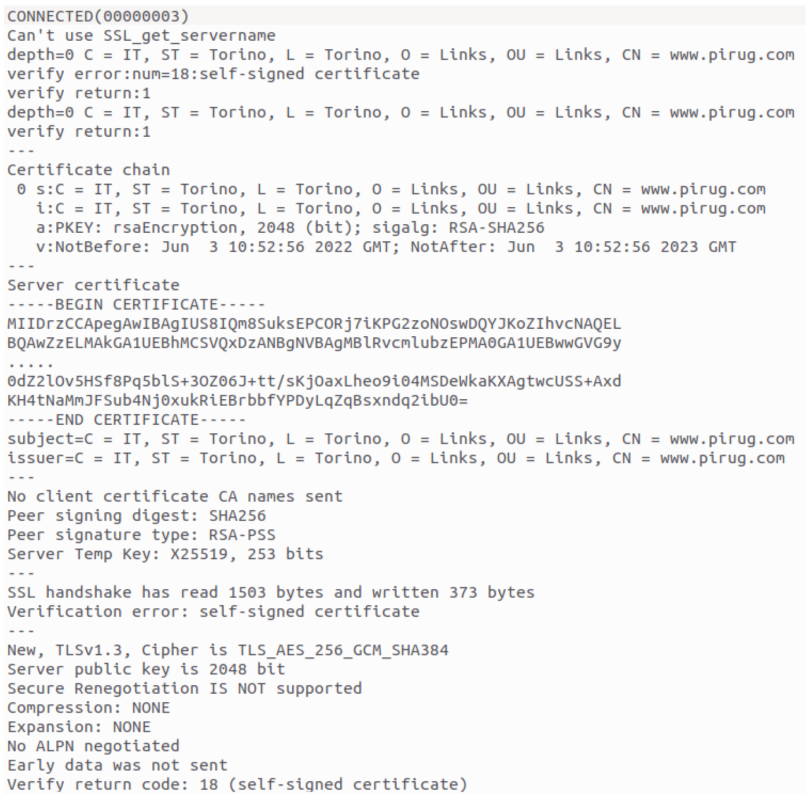
```
./openssl s_server -cert ./server_certificate.pem -key ./server_cert_pkey.pem
    -accept 44330 -www
```

- `s_server`: sets up a TLS server;
- `-cert ./server_certificate.pem`: loads into `openssl` the server's x.509 certificate contained in `server_certificate.pem` file;
- `-key ./server_cert_pkey.pem`: loads into `openssl` the server's private key contained in the `server_cert_pkey.pem` file;
- `-accept 44330`: the server will be listening for incoming TCP connections on port 44330;
- `-www`: sends information back to the client when it connects, like used ciphers and other session parameters.

Client side

```
./openssl s_client -connect localhost:44330
```

- `s_client`: sets up a TLS client;
- `-connect localhost:44330`: specifies the host and the port to connect to.



```
CONNECTED(00000003)
Can't use SSL_get_servername
depth=0 C = IT, ST = Torino, L = Torino, O = Links, OU = Links, CN = www.pirug.com
verify error:num=18:self-signed certificate
verify return:1
depth=0 C = IT, ST = Torino, L = Torino, O = Links, OU = Links, CN = www.pirug.com
verify return:1
---
Certificate chain
 0 s:C = IT, ST = Torino, L = Torino, O = Links, OU = Links, CN = www.pirug.com
   i:C = IT, ST = Torino, L = Torino, O = Links, OU = Links, CN = www.pirug.com
   a:PKEY: rsaEncryption, 2048 (bit); sigalg: RSA-SHA256
   v:NotBefore: Jun  3 10:52:56 2022 GMT; NotAfter: Jun  3 10:52:56 2023 GMT
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIDrzCCApegAwIBAgIUS8IQm8SuksEPCORj7iKPG2zoNOswDQYJKoZIhvcNAQEL
BQAwZzELMAkGA1UEBhMCSVQxDzANBgNVBAgMBlRvcmlubzEPMA0GA1UEBwwGVG9y
.....
0dZ2lOv5HSf8Pq5blS+3OZ06J+tt/sKjOaxLheo9i04MSDeWkaKXAgtwcUSS+Axd
KH4tNaMmJFSub4Nj0xukRiEBrbbfYPDyLqZqBsxndq2ibU0=
-----END CERTIFICATE-----
subject=C = IT, ST = Torino, L = Torino, O = Links, OU = Links, CN = www.pirug.com
issuer=C = IT, ST = Torino, L = Torino, O = Links, OU = Links, CN = www.pirug.com
---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: RSA-PSS
Server Temp Key: X25519, 253 bits
---
SSL handshake has read 1503 bytes and written 373 bytes
Verification error: self-signed certificate
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server public key is 2048 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 18 (self-signed certificate)
```

Figure A.4: Client output of a successful TLS 1.3 handshake using x.509 certificates in OpenSSL.

TLS 1.3 Handshake with mutual authentication - x.509

Server side

```
./openssl s_server -cert ./server_certificate.pem -key ./server_cert_pkey.pem
    -accept 44330 -www -verify 1
```

- `s_server`: sets up a TLS server;
- `-cert ./server_certificate.pem`: loads into `openssl` the server's x.509 certificate contained in `server_certificate.pem` file;
- `-key ./server_cert_pkey.pem`: loads into `openssl` the server's private key contained in the `server_cert_pkey.pem` file;
- `-accept 44330`: the server will be listening for incoming TCP connections on port 44330.
- `-www`: sends information back to the client when it connects, like used ciphers and other session parameters;
- `-verify 1`: requests a certificate from the client and specifies the client certificate chain should contain only one certificate.

Client side

```
./openssl s_client -connect localhost:44330 -cert ./client_certificate.pem
    -key ./client_cert_pkey.pem
```

- `s_client`: sets up a TLS client;

- `-connect localhost:44330`: specifies the host and the port to connect to;
- `-cert ./client_certificate.pem`: loads into `openssl` the client's x.509 certificate contained in `client_certificate.pem` file;
- `-key ./client_cert_pkey.pem`: loads into `openssl` the client's private key contained in `./client_cert_pkey.pem` file.

TLS 1.3 Handshake with server authentication - DID

Server side

```
./openssl s_server -accept 44330 -www -nocert -did
    did:ott:A132F780A4FFF3D439A59A81453B968AA30C4A7687EBFCB1BCE0DC7A2DE0635A
```

- `s_server`: sets up a TLS server;
- `-accept 44330`: the server will be listening for incoming TCP connections on port 44330.
- `-www`: sends information back to the client when it connects, like used ciphers and other session parameters;
- `-nocert`: no certificate is used;
- `-did did:ott:A132F780A4FFF3D439A59A81453B968AA30C4A7687EBFCB1BCE0DC7A2DE0635A`: loads the server's DID into `openssl`.

Client side

```
./openssl s_client -connect localhost:44330 -did_methods ott,eth
```

- `s_client`: sets up a TLS client;
- `-connect localhost:44330`: specifies the host and the port to connect to;
- `-did_methods ott,eth`: the DID methods the client supports.

TLS 1.3 Handshake with mutual authentication - DID

Server side

```
./openssl s_server -accept 44330 -www -nocert -verify 1 -did
    did:ott:A132F780A4FFF3D439A59A81453B968AA30C4A7687EBFCB1BCE0DC7A2DE0635A
    -did_methods ott,eth
```

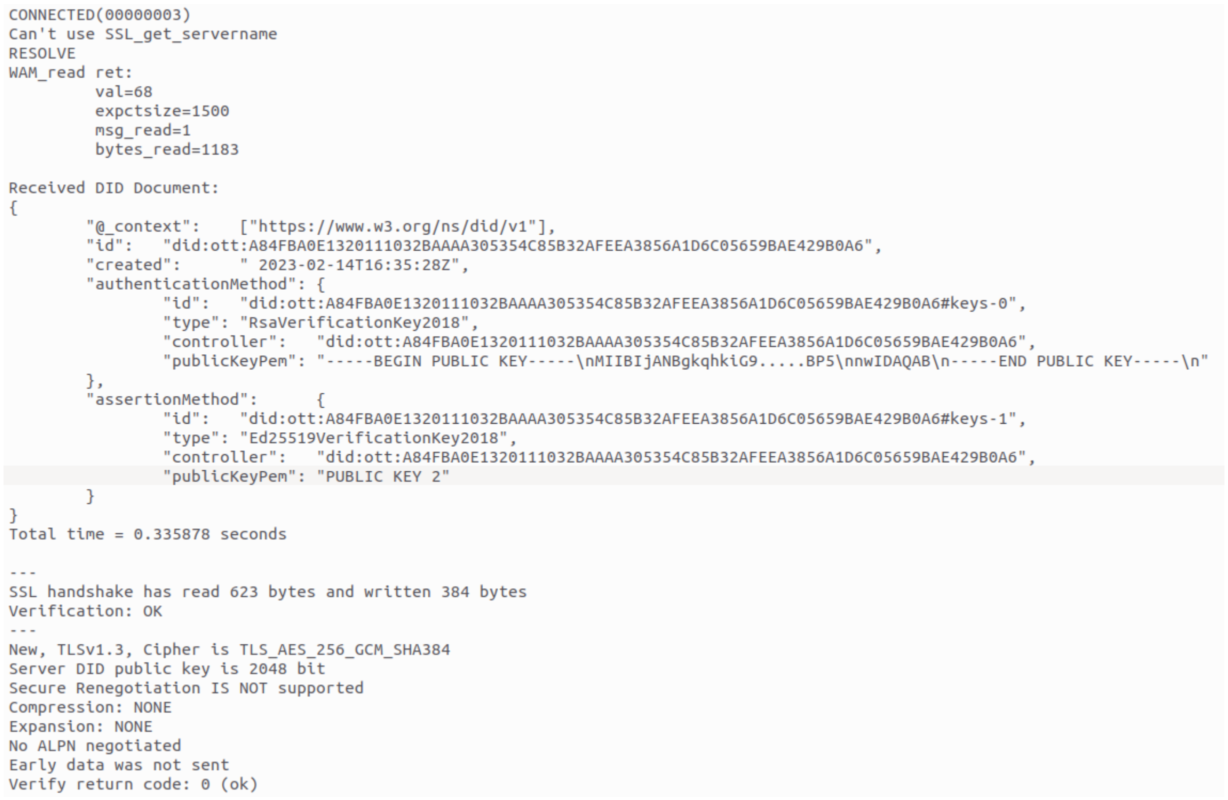
- `s_server`: sets up a TLS server;
- `-accept 44330`: the server will be listening for incoming TCP connections on port 44330.
- `-www`: sends information back to the client when it connects, like used ciphers and other session parameters;
- `-nocert`: no certificate is used;
- `-verify 1`: requests a DID from the client;
- `-did did:ott:A132F780A4FFF3D439A59A81453B968AA30C4A7687EBFCB1BCE0DC7A2DE0635A`: loads the server's DID into `openssl`;

- `-did_methods ott,eth`: the DID methods the server supports.

Client side

```
./openssl s_client -connect localhost:44330 -did
    did:ott:43FAB3D59E0FB5611622CC24E3ECD659D249622333F66C7AAF32E9A958367992
    -did_methods ott,eth
```

- `s_client`: sets up a TLS client;
- `-connect localhost:44330`: specifies the host and the port to connect to;
- `-did did:ott:43FAB3D59E0FB5611622CC24E3ECD659D249622333F66C7AAF32E9A958367992`: loads the client's DID into `openssl`
- `-did_methods ott,eth`: the DID methods the client supports.



```
CONNECTED(00000003)
Can't use SSL_get_servername
RESOLVE
WAM_read ret:
         val=68
         expctsize=1500
         msg_read=1
         bytes_read=1183

Received DID Document:
{
        "@_context":    ["https://www.w3.org/ns/did/v1"],
        "id":   "did:ott:A84FBA0E1320111032BAAAA305354C85B32AFEEA3856A1D6C05659BAE429B0A6",
        "created":      " 2023-02-14T16:35:28Z",
        "authenticationMethod": {
                "id":   "did:ott:A84FBA0E1320111032BAAAA305354C85B32AFEEA3856A1D6C05659BAE429B0A6#keys-0",
                "type": "RsaVerificationKey2018",
                "controller":   "did:ott:A84FBA0E1320111032BAAAA305354C85B32AFEEA3856A1D6C05659BAE429B0A6",
                "publicKeyPem": "-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9.....BP5\nnwIDAQAB\n-----END PUBLIC KEY-----\n"
        },
        "assertionMethod":      {
                "id":   "did:ott:A84FBA0E1320111032BAAAA305354C85B32AFEEA3856A1D6C05659BAE429B0A6#keys-1",
                "type": "Ed25519VerificationKey2018",
                "controller":   "did:ott:A84FBA0E1320111032BAAAA305354C85B32AFEEA3856A1D6C05659BAE429B0A6",
                "publicKeyPem": "PUBLIC KEY 2"
        }
}
Total time = 0.335878 seconds

---
SSL handshake has read 623 bytes and written 384 bytes
Verification: OK
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server DID public key is 2048 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 0 (ok)
```

Figure A.5: Client output of a successful TLS 1.3 handshake using DIDs in OpenSSL.

TLS 1.3 Handshake with cross authentication - Client(DID) & Server(x.509)

Server side

```
./openssl s_server -accept 44330 -www -verify 1 -cert
    ./server_certificate.pem -key ./server_cert_pkey.pem -did_methods ott,eth.
```

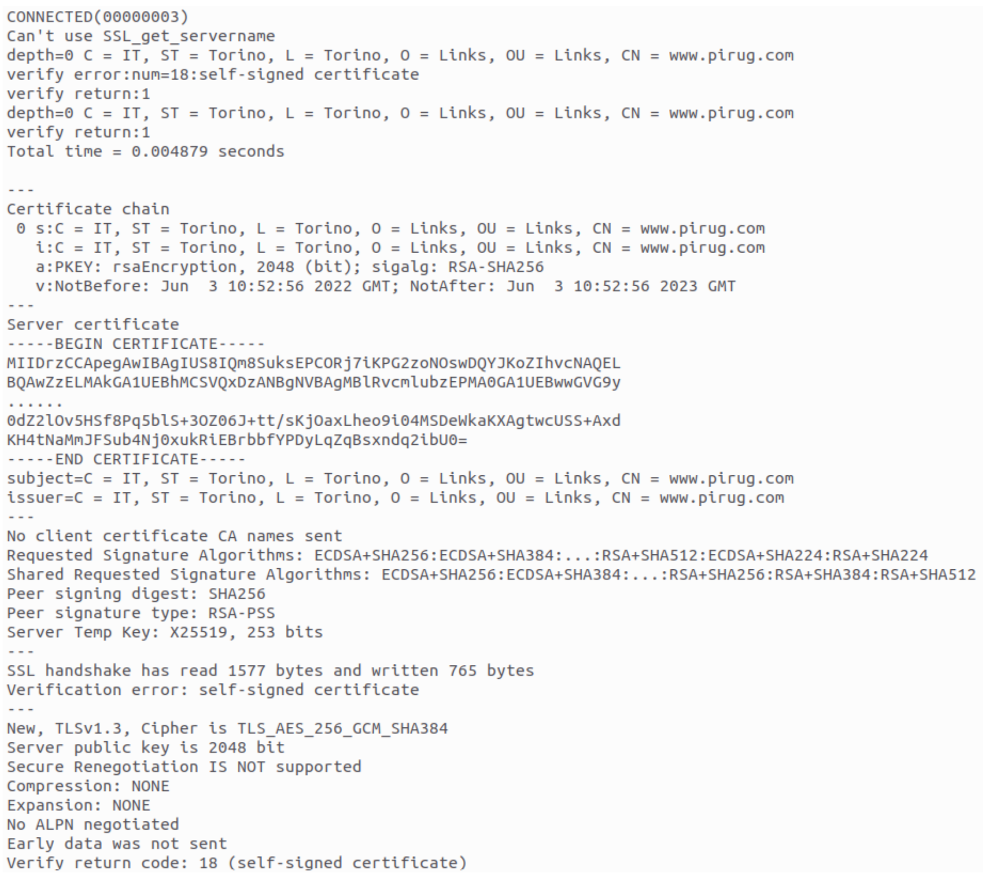
- `s_server`: sets up a TLS server;
- `-accept 44330`: the server will be listening for incoming TCP connections on port 44330.
- `-www`: sends information back to the client when it connects, like used ciphers and other session parameters;

- `-verify 1`: requests a DID from the client;
- `-cert ./server_certificate.pem`: loads into `openssl` the server's x.509 certificate contained in `server_certificate.pem` file;
- `-key ./server_cert_pkey.pem`: loads into `openssl` the server's private key contained in the `server_cert_pkey.pem` file;
- `-did_methods ott,eth`: the DID methods the server supports.

Client side

```
./openssl s_client -connect localhost:44330 -did
    did:ott:43FAB3D59E0FB5611622CC24E3ECD659D249622333F66C7AAF32E9A958367992
```

- `s_client`: sets up a TLS client;
- `-connect localhost:44330`: specifies the host and the port to connect to;
- `-did did:ott:43FAB3D59E0FB5611622CC24E3ECD659D249622333F66C7AAF32E9A958367992`: loads the client's DID into `openssl`



```
CONNECTED(00000003)
Can't use SSL_get_servername
depth=0 C = IT, ST = Torino, L = Torino, O = Links, OU = Links, CN = www.pirug.com
verify error:num=18:self-signed certificate
verify return:1
depth=0 C = IT, ST = Torino, L = Torino, O = Links, OU = Links, CN = www.pirug.com
verify return:1
Total time = 0.004879 seconds

---
Certificate chain
 0 s:C = IT, ST = Torino, L = Torino, O = Links, OU = Links, CN = www.pirug.com
   i:C = IT, ST = Torino, L = Torino, O = Links, OU = Links, CN = www.pirug.com
   a:PKEY: rsaEncryption, 2048 (bit); sigalg: RSA-SHA256
   v:NotBefore: Jun  3 10:52:56 2022 GMT; NotAfter: Jun  3 10:52:56 2023 GMT
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIDrzCCApegAwIBAgIUS8IQm8SuksEPCORj7iKPG2zoNOswDQYJKoZIhvcNAQEL
BQAwZzELMAkGA1UEBhMCSVQxDzANBgNVBAgMBlRvcmlubzEPMA0GA1UEBwwGVG9y
......
0dZ2lOv5HSf8Pq5blS+3OZ06J+tt/sKjOaxLheo9i04MSDeWkaKXAgtwcUSS+Axd
KH4tNaMmJFSub4Nj0xukRiEBrbbfYPDyLqZqBsxndq2ibU0=
-----END CERTIFICATE-----
subject=C = IT, ST = Torino, L = Torino, O = Links, OU = Links, CN = www.pirug.com
issuer=C = IT, ST = Torino, L = Torino, O = Links, OU = Links, CN = www.pirug.com
---
No client certificate CA names sent
Requested Signature Algorithms: ECDSA+SHA256:ECDSA+SHA384:...:RSA+SHA512:ECDSA+SHA224:RSA+SHA224
Shared Requested Signature Algorithms: ECDSA+SHA256:ECDSA+SHA384:...:RSA+SHA256:RSA+SHA384:RSA+SHA512
Peer signing digest: SHA256
Peer signature type: RSA-PSS
Server Temp Key: X25519, 253 bits
---
SSL handshake has read 1577 bytes and written 765 bytes
Verification error: self-signed certificate
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server public key is 2048 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 18 (self-signed certificate)
```

Figure A.6: Client output of a successful TLS 1.3 handshake where the client authenticates with DID and the server with X.509.

TLS 1.3 Handshake with cross authentication - Client(x.509) & Server(DID)

Server side

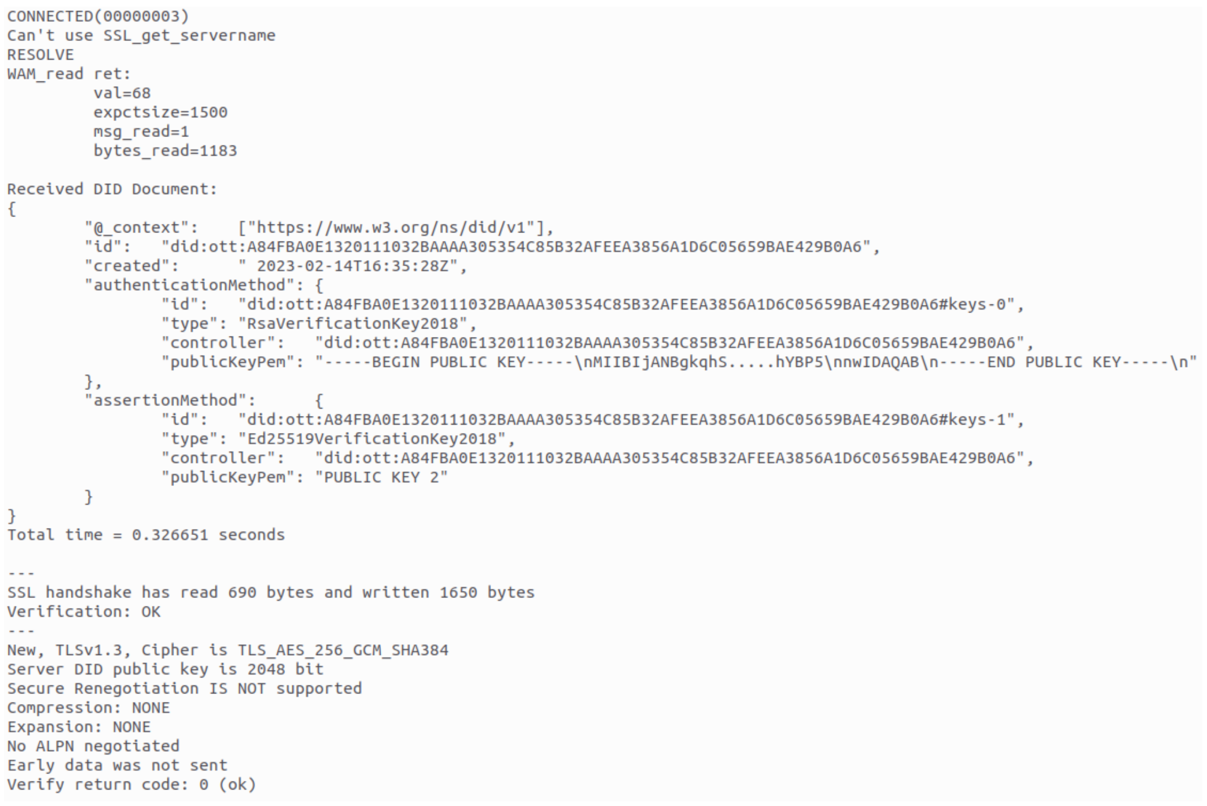
```
./openssl s_server -accept 44330 -www -nocert -verify 1 -did
    did:ott:A132F780A4FFF3D439A59A81453B968AA30C4A7687EBFCB1BCE0DC7A2DE0635A
    -did_methods ott,eth
```

- `s_server`: sets up a TLS server;
- `-accept 44330`: the server will be listening for incoming TCP connections on port 44330.
- `-www`: sends information back to the client when it connects, like used ciphers and other session parameters;
- `-nocert`: no certificate is used;
- `-verify 1`: requests a DID from the client;
- `-did did:ott:43FAB3D59E0FB5611622CC24E3ECD659D249622333F66C7AAF32E9A958367992`: loads the server's DID into `openssl`;
- `-did_methods ott,eth`: the DID methods the server supports.

Client side

```
./openssl s_client -connect localhost:44330 -cert ./client_certificate.pem
    -key ./client_cert_pkey.pem -did_methods ott,eth
```

- `s_client`: sets up a TLS client;
- `-connect localhost:44330`: specifies the host and the port to connect to;
- `-cert ./client_certificate.pem`: loads into `openssl` the client's x.509 certificate contained in `client_certificate.pem` file;
- `-key ./client_cert_pkey.pem`: loads into `openssl` the client's private key contained in `./client_cert_pkey.pem` file;
- `-did_methods ott,eth`: the DID methods the client supports.



```
CONNECTED(00000003)
Can't use SSL_get_servername
RESOLVE
WAM_read ret:
         val=68
         expctsize=1500
         msg_read=1
         bytes_read=1183

Received DID Document:
{
        "@_context":    ["https://www.w3.org/ns/did/v1"],
        "id":   "did:ott:A84FBA0E1320111032BAAAA305354C85B32AFEEA3856A1D6C05659BAE429B0A6",
        "created":      " 2023-02-14T16:35:28Z",
        "authenticationMethod": {
                "id":   "did:ott:A84FBA0E1320111032BAAAA305354C85B32AFEEA3856A1D6C05659BAE429B0A6#keys-0",
                "type": "RsaVerificationKey2018",
                "controller":   "did:ott:A84FBA0E1320111032BAAAA305354C85B32AFEEA3856A1D6C05659BAE429B0A6",
                "publicKeyPem": "-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhS.....hYBP5\nnwIDAQAB\n-----END PUBLIC KEY-----\n"
        },
        "assertionMethod":      {
                "id":   "did:ott:A84FBA0E1320111032BAAAA305354C85B32AFEEA3856A1D6C05659BAE429B0A6#keys-1",
                "type": "Ed25519VerificationKey2018",
                "controller":   "did:ott:A84FBA0E1320111032BAAAA305354C85B32AFEEA3856A1D6C05659BAE429B0A6",
                "publicKeyPem": "PUBLIC KEY 2"
        }
}
Total time = 0.326651 seconds

---
SSL handshake has read 690 bytes and written 1650 bytes
Verification: OK
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server DID public key is 2048 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 0 (ok)
```

Figure A.7: Client output of a successful TLS 1.3 handshake where the client authenticates with X.509 and the server with DID.

Appendix B

Developer Manual

B.1 SSI-aware OpenSSL

B.1.1 The `ssl` directory

`statem_local.h` is a key file in the `ssl` directory, since it hosts two fundamental structures in the library: `SSL_CTX` (alias of `ssl_ctx_st`) and `SSL` (alias of `ssl_st`). The first one represents the configuration and state of an SSL/TLS context. It contains information such as the SSL/TLS version to use, the set of available cipher suites, the trusted certificate authorities, etc. It is typically used to create new SSL/TLS connections. The second one represents the state of an individual SSL/TLS connection. It contains information such as the current cipher suite, the session key, and the peer's certificate. This structure is used to maintain the state of an existing SSL/TLS connection.

For the SSI-aware TLS model I have added in this file two new structures, one that recalls the other, to manage DID documents and made them available to the `SSL` and `SSL_CTX` structures. The body of these structures is symmetrical to the ones that handle X.509 certificates.

```
typedef struct did_pkey_st DID_PKEY

struct did_pkey_st {
        unsigned char *did;
        size_t did_len;
        uint8_t did_method;
        EVP_PKEY *privatekey;
};
```

The `DID_PKEY` (alias of `did_pkey_st`) structure stores all the fields of an endpoint's DID document that are considered relevant for the handshake, such as the string that identifies the DID, its length and DID method. Furthermore, the `EVP_PKEY *privatekey` field corresponds to the authentication method private key and will be indispensable for the endpoint to construct the DID Verify message. `EVP_PKEY *privatekey` is an OpenSSL internal structure to save public and private keys.

```
typedef struct did_st {
        DID_PKEY *key;
        DID_PKEY pkeys[SSL_PKEY_NUM];
} DID;
```

The DID (alias of `did_st`) structure provides an array of `DID_PKEY` elements. `SSL_PKEY_NUM` is a constant and represents the number of asymmetric key types supported in OpenSSL, thus an endpoint's DID document is classified by its authentication method key type. When a DID document will be uploaded into this structure will be placed in a specific position in the array, leaving the rest of the array empty, and the `DID_PKEY *key` is a pointer to the above-mentioned DID document in the array.

An endpoint should also need to store some information about the peer's DID document after resolving it, such as the authentication method public key that can be saved in the `EVP_PKEY *peer_did_pubkey` variable. This information will be essential to verify the peer's signature when we receive its DID Verify message. Anyway this element is only present in the `SSL` structure, since it belongs to a specific TLS connection.

In the `SSL` structure I have also added the `authn_method` variable, which is an integer, that will be employed by an endpoint to figure out whether to authenticate itself with an X.509 certificate (value 0) or a DID (value 1).

B.1.2 The `ssl/statem` subdirectory

The `ssl/statem` directory contains the files for the handshake state machine. The `extensions.c` file contains general methods to manage the extensions in the handshake messages, while the files `extensions_clnt.c` and `extensions_srvr.c` provide the implementation of client-specific and server-specific extensions respectively. Similarly, the `statem.c` contains general handshake state machine functions while `statem_clnt.c` and `statem_srvr.c` implement client-specific and server-specific handshake messages. The functions in these two files observe the syntax: `tls_construct_...` and the name of the message to send when constructing a message, `tls_process_...` and the name of the message to process when receiving a message. The two files I've added, the header one (`statem_local_did.h`) and the source one (`statem_local_did.c`) include functions for the new handshake messages (e.g. DID Request, DID and DID Verify) following the syntax of the `statem_clnt.c` and `statem_srvr.c` files. These functions are discussed in the `statem_local_did.c` file.

It's worth noting that in the following methods the `WPACKET` structure is utilized to store data that will be sent through the network, while the `PACKET` structure contains the data that comes from the network.

Client Methods

`EXT_RETURN tls_construct_ctos_supported_did_methods (...)`

This is the function that constructs the client's *supported did methods* extension. If the client supports DID methods this function writes them into a `WPACKET`, otherwise it does nothing.

Input:

- `SSL *s`: the structure that contains the TLS connection parameters;
- `WPACKET *pkt`: the supported DID methods will be written here;
- `unsigned int context`: expresses the state of the handshake, in this case is `SSL_EXT_CLIENT_HELLO` since the construction of this extension happens in Client Hello;
- `X509 *x`: this parameter is not `NULL` only when the extension is applied to a specific certificate in the Certificate message, so here it is NULL;
- `size_t chainidx`: equals to zero, since this field is still related to Certificate message extensions.

Output: if the client has DID methods to send it will return the enum value `EXT_RETURN_SENT`, otherwise `EXT_RETURN_NOT_SENT`.

`int tls_parse_stoc_supported_did_methods(...)`

This function parses the DID methods the server has sent to the client in the Did Request message and saves them into the `s->ext.peer_supporteddidmethods` variable.

Input:

- `SSL *s`: the structure that contains the TLS connection parameters;
- `PACKET *pkt`: reads the incoming DID methods from this variable;
- `unsigned int context`: in this case is `SSL_EXT_TLS1_3_DID_REQUEST` since the parsing of this extension happens while processing the Did Request message;
- `X509 *x`: NULL;
- `size_t chainidx`: equals to zero.

Output: returns 1 in case of successful outcome, 0 otherwise.

`MSG_PROCESS_RETURN tls_process_did_request(...)`

This function processes the Did Request message received from the server. It first saves the did request context into `s->pha_context` and then parses the two extensions it is supposed to receive: *supported did methods* and *did signature algorithms.* Successively it sets the signature algorithms it has in common with the server and checks if its DID has a DID method compatible with one of those sent by the server by invoking `tls1_process_sigals()` and `tls1_process_supported_did_methods()` respectively.

Input:

- `SSL *s`: the structure that contains the TLS connection parameters;
- `PACKET *pkt`: the DID Request message is read from here.

Output: in case everything goes smoothly it returns the `MSG_PROCESS_CONTINUE_READING` enum value to express the processing of this message is over and we can proceed with the next one, if something goes wrong returns the `MSG_PROCESS_ERROR` enum value.

`MSG_PROCESS_RETURN tls_process_server_did(...)`

Upon a DID message received from the server the client calls this function. First, it reads the context, the DID method and the DID from `pkt`. Afterwards it uses the resolve operation supplied by `libdidmethod.a` to retrieve the server's DID document from the ledger. It does some internal operations to transform the server's authentication method public key from an array of `unsigned char` into a `EVP_PKEY` structure and saves it into the `s->session->peer_did_pubkey` variable. In the end it saves the current handshake state for when it receives the DID Verify message.

Input:

- `SSL *s`: the structure that contains the TLS connection parameters;
- `PACKET *pkt`: the DID message is read from here.

Output: If all the operations listed above succeed it returns the `MSG_PROCESS_CONTINUE_READING` enum value to express the processing of this message is over and we can proceed with the next one, otherwise it returns the `MSG_PROCESS_ERROR` enum value.

`int tls_construct_client_did(...)`

When constructing the DID message the client writes in `pkt` the DID request context, the DID method of its DID and the actual DID. The first one is always zero since post-handshake authentication has not been implemented yet, while the other two are extracted from the `s->did->key` variable where `did` is a pointer to a `did_st` element, that as we have seen earlier it contains the endpoint's DID document information.

Input:

- `SSL *s`: the structure that contains the TLS connection parameters;
- `PACKET *pkt`: the DID message is read from here.

Output: In case of success it returns 1, otherwise 0.

Server Methods

`int tls_parse_ctos_supported_did_methods(...)`

This method parses the DID methods the client has sent to the server in the Client Hello message and saves them into the `s->ext.peer_supporteddidmethods` variable.

Input:

- `SSL *s`: the structure that contains the TLS connection parameters;
- `PACKET *pkt`: reads the incoming DID methods from here;
- `unsigned int context`: in this case is `SSL_EXT_CLIENT_HELLO` since the parsing of this extension happens while processing the Client Hello message;
- `X509 *x`: NULL;
- `size_t chainidx`: equals to zero;

Output: returns 1 in case of successful outcome, 0 otherwise.

`EXT_RETURN tls_construct_stoc_supported_did_methods(...)`

The server invokes this function to construct the *supported did methods* extension to send to the client in the Did Request message. The only difference from the corresponding client's method on is that if the server has previously received the *supported did methods* extension from the client it must send the list of DID methods they both have in common and that is stored into the `s->shared_didmethods` variable, otherwise it must send the full list of supported DID methods held in `s->ext.supporteddidmethods`.

Input:

- `SSL *s`: the structure that contains the TLS connection parameters;
- `WPACKET *pkt`: the DID methods will be written here;
- `unsigned int context`: expresses the state of the handshake, in this case is `SSL_EXT_TLS1_3_DID_REQUEST` since the construction of this extension happens in the Did Request message;
- `X509 *x`: NULL;
- `size_t chainidx`: zero.

Output: if the server has DID methods to send it will return the `EXT_RETURN_SENT`enum value, otherwise the `EXT_RETURN_NOT_SENT` enum value.

`int tls_construct_did_request(...)`

A server that constructs a Did Request must insert into the WPACKET the *DID request context* that must always be zero (post-handshake authentication has not been implemented yet) and the two extensions *supported did methods* and *did signature algorithms*, through the `tls_construct_extensions()` function.

Input:

- `SSL *s`: the structure that contains the TLS connection parameters;
- `WPACKET *pkt`: the content of the Did Request message is written here.

Output: in case of success it returns 1, otherwise 0.

`int tls_construct_server_did(...)`

This function constructs the server's DID message by writing in `*pkt` the DID request context (value zero), the DID method of the server's DID and the actual DID.

Input:

- `SSL *s`: the structure that contains the TLS connection parameters;
- `WPACKET *pkt`: the content of the Did Request message is written here.

Output: in case of success it returns 1, otherwise 0.

`MSG_PROCESS_RETURN tls_process_client_did(...)`

Similarly to the client, the server processes the peer's DID using the `resolve` function of `libdidmethod.a` and stores the client's public key into the `s->session->peer_did_pubkey` variable.

Input:

- `SSL *s`: the structure that contains the TLS connection parameters;
- `PACKET *pkt`: reads the DID message from here.

Output: in case of success returns the `MSG_PROCESS_CONTINUE_READING` enum value to express the processing of this message is over and we can proceed with the next one, otherwise it returns the `MSG_PROCESS_ERROR` enum value.

General Methods

`int tls_construct_did_verify(...)`

Both the client and the server employ the same function to construct the DID Verify message. The function first verifies the endpoint's `s->did->privatekey` is not NULL, then it prepares the content to be signed and signs it calling `get_did_verify_tbs_data()` and `EVP_DigestSign()` respectively and finally writes in `*pkt` the DID signature algorithm utilized and the content of the signature.

Input:

- `SSL *s`: the structure that contains the TLS connection parameters;
- `WPACKET *pkt`: the content of the DID Verify message is written here.

Output: in case of success returns 1, otherwise 0.

`MSG_PROCESS_RETURN tls_process_did_verify(...)`

Also the processing of the DID Verify message is equal for both the client and the server. An initial check verifies the peer's public key type contained in `s->session->peer_did_pubkey` matches the signature algorithm included in the DID Verify message. Additionally it reads the content of the signature contained in `*pkt` and puts it in the `data` variable, then calculates the hash of the content that was supposed to be signed invoking the `get_did_verify_tbs_data()` and stores it in `hdata`. Finally it verifies the signature through the `EVP_DigestVerify()` function by feeding in input the `data` and `hdata` variables.

Input:

- `SSL *s`: the structure that contains the TLS connection parameters;
- `PACKET *pkt`: the content of the DID Verify message is read from here.

Output: in case of success returns the `MSG_PROCESS_CONTINUE_READING` enum value to express the processing of this message is over and we can proceed with the next one, otherwise it returns the `MSG_PROCESS_ERROR` enum value.

B.1.3 The `ssl` general files

Among the general files in the `ssl` directory I included a new source file (did.c) and its header file (ssl_local_did.h) to supply some utility functions to manage the new components contained inside the `SSL` and `SSL_CTX` structures.

`int tls13_set_server_did_methods(...)`

This method is invoked by the server during the post processing of the client hello and it represents the real novelty in this new version of OpenSSL. It allows the server to establish how it should authenticate itself: if the client did not send any DID methods the `s->auth_method` will be set to `CERTIFICATE_AUTHN` (value 0) as the server should authenticate with the Certificate and Certificate Verify message, otherwise if the client has sent the *supported did methods* extension the server will set `s->auth_method` to `DID_AUTHN` (value 1). Moreover, in this last case if also the server has a list of supported DID methods, it stores into `s->shared_didmethods` the DID methods it has in common with the client.

Input:

- `SSL *s`: the structure that contains the TLS connection parameters;

Output: returns 1 in case of success, 0 otherwise.

`int ssl_has_did(...)`

Since server authentication is always mandatory in TLS, this function is invoked by the server together with `ssl_has_cert()` when setting up the TLS connection since the server must possess at least either an X.509 certificate or a DID.

Input:

- `SSL *s`: the structure that contains the TLS connection parameters;
- `int idx`: value between 0 and `SSL_PKEY_NUM`;

Output: returns 1 in case of success, 0 otherwise.

`DID *ssl_did_new(...)`

This function gets invoked from an external function `SSL_CTX_new_ex()` employed by both the client and the server, in `s_client.c` and `s_server.c` respectively, to initialize the handshake context (`SSL_CTX`). In the specific it instantiates the `DID *did` variable.

Output: returns a pointer to the `DID` structure in case of success, 0 otherwise.

`DID *ssl_did_dup(...)`

Both the endpoints make use of the `SSL_new()` function to create a new TLS connection, which internally duplicates most of the content of the `SSL_CTX` element into the `SSL` one, so the ssl_did_dup() function duplicates the content of `SSL_CTX->did` into `SSL->did`.

Input:

- `DID *did`: pointer to the element to duplicate;

Output: returns a pointer to the new element in case of success, a NULL pointer otherwise.

`int send_did_request(SSL *s)`

The server resorts to this method when in the writing state of the handshake has sent the *Encrypted Extensions* message and has to figure out whether client authentication is requested or not. If the `-verify` option in the `s_server` configuration is present then the server should send a DID Request message.

Input:

- `SSL *s`: the structure that contains the TLS connection parameters;

Output: returns 1 in case of client authentication required, 0 otherwise.

B.1.4 The apps directory

The `apps` directory include the files to implement the various OpenSSL applications. The `apps/s_server.c` and `apps/s_client.c` files contain the code of the `s_server` and `s_client` applications respectively. I have equipped these two files with two brand new options: the `-did` option permits to specify the endpoint's DID string while the `-did_methods` one allows to list all the endpoint's supported DID methods. Both the DID and DID methods will be later on uploaded into an instance of the `SSL_CTX` to initialize the handshake context.

The s_server_main() parses all the options received , initializes the `SSL_CTX *ctx` variable through the `SSL_CTX_new_ex()` function loads into it the parsed options and starts listening for incoming TLS connections. Once a new connection is received the `SSL_new()` function to instantiate an `SSL` structure is invoked.

The `s_client_main()` similarly to `s_server_main()` parses all the options, instantiates the `SSL_CTX` variable through the SSL_CTX_new_ex() function, loads into it the parsed options, duplicates its content into an instance of the `SSL` structure calling the `SSL_new()` function and starts the handshake.

Below, we will go through the methods that allow to manage the options I have added in `s_client` and `s_server`.

`int ssl_ctx_set_did_methods()`

Fills the content of the `ctx->ext.supportedidmethods` variable with the list of DID methods present in the `-did_methods` option.

Input:

- `SSL_CTX *ctx`: the structure that contains the handshake context;
- `const char *did_methods`: the list of DID methods contained in the `-did_methods` option.

Output: returns 1 in case of success, 0 otherwise.

`int set_did_key_stuff()`

Loads into `ctx->did` the DID document private key (`EVP_PKEY *key`), the DID string and its DID method.

Input:

- `SSL_CTX *ctx`: the structure that contains the handshake context;
- `EVP_PKEY *key`: the DID document private key loaded from memory.
- `const char *did`: the DID contained in the `-did` option

Output: returns 1 in case of success, 0 otherwise.

Bibliography

[1] A. Preukschat and D. Reed, "Self-sovereign identity", Manning Publications, 2021

[2] O. Dib and K. Toumi, "Decentralized identity systems: architecture, challenges, solutions and future directions", Annals of Emerging Technologies in Computing (AETiC), Print ISSN, 2020, pp. 2516–0281

[3] M. Alizadeh, K. Andersson, and O. Schelén, "Comparative analysis of decentralized identity approaches", IEEE Access, 2022

[4] N. El Ioini and C. Pahl, "A review of distributed ledger technologies", OTM Confederated International Conferences" On the Move to Meaningful Internet Systems", 2018, pp. 277–288

[5] World Wide Web Consortium (W3C), https://www.w3.org/TR/did-core

[6] World Wide Web Consortium (W3C), https://www.w3.org/TR/vc-data-model/

[7] Trust Over IP Foundation, https://trustoverip.org/

[8] D. Cooper, S. Santesson, S.Farrell, S. Boeyen, and R. Housley, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile." RFC-5280, May 2008, DOI 10.17487/RFC5280

[9] National Cyber Security Centre, https://www.ncsc.gov.uk/collection/in-house-public-key-infrastructure/introduction-to-public-key-infrastructure/ca-hierarchy

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

[11] The OpenSSL project, http://www.openssl.org/

[12] Dave Gamble, https://github.com/DaveGamble/cJSON

[13] IOTA, https://github.com/iotaledger/iota.c

[14] libsodium, https://doc.libsodium.org/