POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Self-Sovereign Identity aware TLS handshake with MbedTLS

Supervisor
prof. Antonio Lioy

Candidate
Alberto Solavagione

Tutor
LINKS FOUNDATION
PhD. ing. Andrea Vesco
PhD. ing. Alberto Carelli

Academic year 2022-2023

Summary

In the digital age, online identity has become a critical part of our daily lives and has been managed through centralized and federated identity models. However, both models have drawbacks, such as a lack of user control and privacy concerns. The presence of intermediaries, such as certification authority and identity providers, is a fundamental requirements in both models. Nevertheless, they represent possible targets of cyber attacks, resulting in as data breaches and identity theft. The Self-Sovereign Identity is a new paradigm in digital identity that seeks to address these issues by putting users in control of their identity and personal data, avoiding the involvement of centralized authorities or third-party intermediaries. One of the key building blocks of the SSI model is the Decentralized Identifier (DID), which is a unique identifier anchored on a Distributed Ledger Technology (DLT). DIDs enable entities to authenticate and authorize identity-related transactions and interactions, without relying on a centralized authority.

Currently, the implementation of SSI solutions is limited, particularly in the realm of IoT systems. This is because IoT systems have highly constrained hardware and software capabilities, which further complicates their integration with the Self-Sovereign paradigm. The main purpose of this thesis is to bring within this constrained IoT world the SSI model, in order to define an ecosystem of devices that, making use of their digital identity, can communicate securely over network. The main idea is to integrate the usage of DIDs within the TLS handshake, to create a secure communication channel using an SSI-aware approach. By introducing a decentralized authentication mechanism, third party entities are no longer required for identity management. In this context, the DID Documents, to which the DIDs refer, are stored onto the DLT, which serves as a Root-of-Trust, leveraging its inherent property of data immutability.

Timing performance measurements are gathered to evaluate the impact of the additional SSI features. Such performances are measured on three different platforms, ie, x86/x64, ARM (Raspberry Pi) and STM32 board based on ARM Cortex-M4, according to the different TLS authentication models (server only and mutual) and according to different peer identity key types and signature algorithms (RSA2048-SHA256 and ECDSA-p256-SHA256).

Acknowledgements

I want to express my gratitude to Professor Antonio Lioy for his supervision. I'm also deeply thankful to Ing. Andrea Vesco and Ing. Alberto Carelli of the Cybersecurity team at LINKS Foundation for their invaluable advice, constant support, and expertise during the completion of this project, as well as for making life at the company stimulating and never dull. I'm grateful to my family for their unwavering support throughout my university journey. Without them, this accomplishment wouldn't have been possible.

Contents

Chapter 1

Introduction

"The Internet of Things (IoT) has enabled as well as boosted several applications that will radically change everyday life, including smart building, environmental monitoring, smart energy grids, and intelligent transportation. Unlike traditional IT systems, IoT deployments will be directly exposed and reachable over the Internet, and massively composed of constrained devices equipped with limited resources."[1]

However, these devices introduce risks that can potentially represent a threat to the environment in which they are employed and to the privacy of those who use them, since they often process sensitive informations. For this reason the security of these devices is crucial, but it is also equally difficult to ensure it given the variety of ecosystems where these devices are used and given their resource-constrained nature.

Among the main secure communication protocols used in this world there is the Transport Layer Security (TLS), which allows to provide confidentiality, integrity and authenticity of communications between entities. Among these properties, authentication is perhaps the most important, as in the absence of it the other properties might be compromised.

In order to authenticate themselves, IoT devices must have their digital identity, which nowadays is based on a centralized model which makes use of X.509 certificates issued by Certification Authorities (CAs). CAs are vulnerable targets and are subjected to attacks every day and removing a CA from the status of "trusted", as soon as it is compromised, is not such a simple process as it may seem, since revoking a popular CA can lead to problems of world-wide compatibility and network communication. An even worse case is when a CA, either by mistake or because it has been compromised, issues a fake certificate, but decides to not disclose it publicly, putting the safety of end users at risk.

1.1 Target of the Thesis

Hence the purpose of this thesis, namely to modify an existing security protocol as the TLS and adapt it to the Self Soverign Identity paradigm, based on a decentralized digital identity model, which relies on Distributed Ledger Technology (DLT)

such as Blockchain or Tangle, and on Decentralized Identifiers (DIDs).
DLTs provide a decentralized infrastructure for registering events on a distributed verifiable register, to which all devices in the network can access and rely. One of its properties is the data immutability and integrity which makes the DLT the Root of Trust.
DIDs, instead, can be seen as addresses pointing to a block on the DLT where is recorded in an immutable way all the cryptographic material that represents the identity of the subject. These two technologies together allow to create an authentication model where there is no central authority needed to certify your identity, concept on which the Self-Sovereign Identity paradigm is based.

Among the various TLS libraries, MbedTLS [2] was chosen as it is the most widely used library in IoT environments, which is the main focus of the thesis. Currently, SSI solutions implementation is limited, particularly in the realm of IoT systems, due to the highly constrained hardware and software capabilities of such systems, which further complicates their integration with the Self-Sovereign paradigm. The other objective of this thesis is to bring the SSI model into this constrained IoT world, in order to define an ecosystem of devices that can communicate securely over the network by leveraging their digital identity.

In the next chapters, the concepts of DID, DID Document and DID Method, fundamental to understand how the Self Soverign paradigm works, will be explored, together with the design and implementation of a new SSI-aware TLS handshake model.

Chapter 2

Background and related work

2.1 Self-Sovereign Identity

In the digital age, online identity has become a critical part of our daily lives. For many years, centralized and federated identity models have been the norm for managing online identity. In the centralized identity model, a central authority or organization controls and manages users' identity and personal data, such as login credentials, personal information, and authentication tokens. This approach has been widely used by social media platforms, email providers, and other online services.

However, centralized identity models have several drawbacks, including lack of user control and privacy concerns. In response, federated identity models emerged as an alternative approach, which allows users to use their digital identity across multiple organizations and services while still maintaining control of their personal data. Examples of federated identity include single sign-on (SSO) and OpenID Connect.

Despite these advances, both centralized and federated identity models still rely on intermediaries, such as identity providers or third-party services, to manage users' identity and data. This creates a risk of data breaches, identity theft, and loss of privacy, as intermediaries may mishandle users' data or be targeted by malicious actors.

Self-Sovereign Identity (SSI) is a new paradigm in digital identity that seeks to address these issues by putting users in control of their identity and personal data. With SSI, users can create, store, and manage their identity and personal data in a secure and decentralized manner, without relying on intermediaries. SSI uses Distributed Ledger Technologies (DLT) to create a tamper-proof, verifiable, and user-controlled digital identity that can be used across multiple services and organizations.

This approach, therefore, brings advantages to both individuals as mentioned above, and organizations, reducing resource consumption, risk of cyber attacks or lawsuits, by storing less user data.

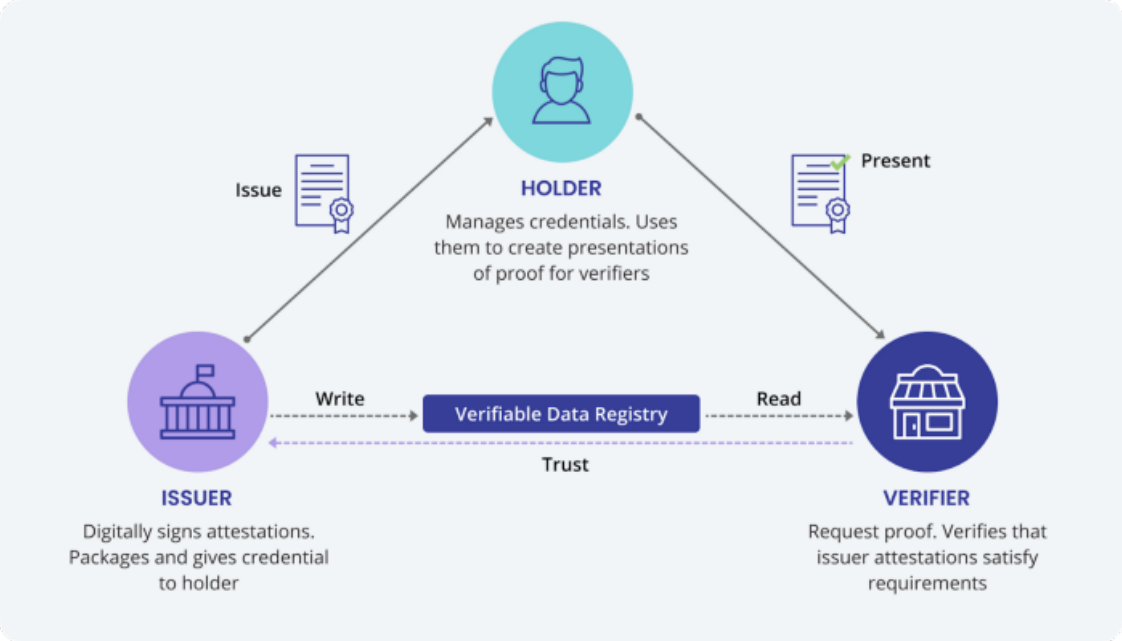

Figure 2.1. SSI Trust Triangle [3].

In a SSI environment there are three main actors participating (Figure 2.1):

- **Issuer**: entity with the authority to issue Verifiable Credentials
- **Holder**: entity which creates its own DID and receives a Verifiable Credential from an Issuer and present proofs of claims from one or more credentials when requested by verifiers.
- **Verifier**: entity checking the Holder's Verifiable Credential by verifing the proofs received

The relationship between the issuers, holders, and verifiers is called the trust triangle, because these participants must trust eachother in order to make this system work. These entities can be either people, organizations or even devices, for examples IoT devices.

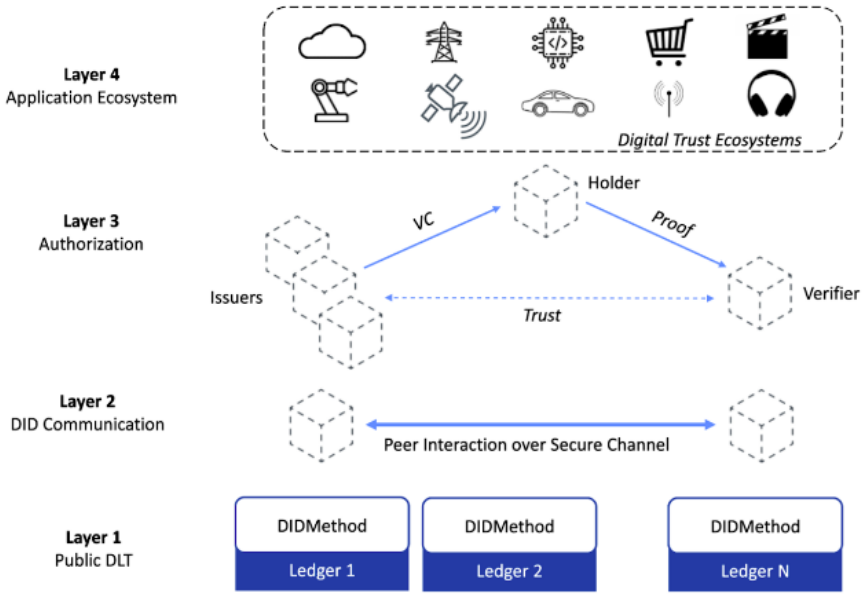

Figure 2.2. SSI stack

The all process of issuing and verifying a Verifiable Credential, involves the other two pillars of the SSI stack (Figure 2.2), which are the Distributed Ledger Technology and the Decentralized Identifiers.

2.2 Distributed Ledger Technology (DLT)

Distributed Ledger Technology is *"a non-centralized system for recording events. These systems establish sufficient confidence for participants to rely upon the data recorded by others to make operational decisions. They typically use distributed databases where different nodes use a consensus protocol to confirm the ordering of cryptographically signed transactions. The linking of digitally signed transactions over time often makes the history of the ledger effectively immutable."*[4]

The consensus algorithm serves to provide synchronization, ensuring the equality of all the copies of data on the various nodes. There are several types of consensus algorithms, which vary in complexity and scalability. Generally the simplest algorithms tend to be the least scalable ones, since are poorly optimized and require lots of use of resources.

Two of the main distributed ledger technologies are Blockchain and DAG.

The Blockchain

The blockchain is a DLT in which transaction records are stored as a chain of blocks in a verifiable decentralized database. "Chain of blocks" means any type of data linked together using cryptographic mechanisms. New transactions result in a new block, in which additionally, a hash of the transaction and a hash from the previous block will be recorded to provide a link between the blocks. So, if one transaction in a block changes, the newly calculated hash will differ from the already registered

hash, making the block invalid. In a blockchain there will always be two types of distinct entities, the transaction issuers and transaction verifiers (miners).

Every miner should verify and agree on the new block before it is added. This process is defined by the consensus algorithms. As a result of the consensus algorithm, one of the miners wins the ability to share the new block with the network. The other miners are required to verify the block to ensure that the data and hashes are valid. This process, often, require a lot of resources and doesn't scale well with high transaction loads, making this algorithm a bottleneck. This happens because the process is sequential so a verifier cannot validate a transaction if the previous one has not been already validated.

In this technology the hash function can be seen as a fingerprint, since modifying a data is not trivial given the decentralized nature of the system, where a copy of the Blockchain is shared between all participants. As soon as a new node joins the network, it receives a copy of all records. So, to mount an attack, the attacker needs to control more then 50% of the nodes in the network, to be able to successfully change the data in the Blockchain.

The Directed Acyclic Graph

The Directed Acyclic Graph (DAG) is not made by a chain of blocks like blockchains, but its structure consists of a graph with vertices connected by edges, such that following the directions pointed by the edges will never create a closed loop. The vertices can be seen as the transactions registered into the DLT, whereas edges represent the related validation processes that occurs among them.

Instead of mining, DAG networks use a process called "transaction confirmation." In a DAG, each node maintains a local copy of the ledger and is responsible for validating transactions. When a new transaction is made, it is first broadcast to a subset of nodes in the network called "witnesses." These witnesses verify the validity of the transaction and attach it to the DAG. The transaction is considered confirmed once it has been verified by a certain number of witnesses, which varies depending on the specific DAG implementation. Once a transaction is confirmed, it is considered final and cannot be reversed. The transaction confirmation process in DAG networks has several advantages over mining in blockchain networks, including lower energy consumption and faster transaction times. This aspect makes this DLT architecture suitable for IoT devices and systems with limited resources.

One of most widespread DLTs implementing the DAG is the Tangle, made by IOTA (See Section 2.6), which is the distributed ledger used is this project.

Figure 2.3. Blockchain vs Tangle [5].

2.3 Decentralized Identifiers (DIDs)

A Decentralized Identifier [4] is a new type of identifier that allows an entity to be recognized in a digital, decentralized and univocal way. DIDs can be seen as URIs that associate a DID subject (a person, organization, IoT device) with a DID document, which is a resources, located on a DLT, containing the cryptographic material that allows an entity to prove its association with the DID.

An entity can have more that one DID, since the generation of Decentralized Identifiers is a process in which the entity has complete control. Having multiple DIDs can be useful in maintaining a separation of identities and interactions, making difficult to track a user on the internet and thus guarantee more privacy.

Starting from a DID, the informations contained in the DID Document can be retrieved through the "resolve" operation implemented by a DID Method. A DID Method defines the mechanisms for creating, resolving, updating and revoking DIDs and DID Documents by making use of a specific DLT. If a DID was created using a particular DID Method, it can only be resolved through the "resolve" operation implemented by that specific method.

Several DID Methods specifications already exists, but in this project it was decided to develop a new one more suitable to our case.

The DID format is a simple string made of three parts (Figure 2.4):

- the DID URI scheme identifier
- the DID Method identifier
- the DID method-specific identifier

The first one indicates that the encountered URI string follows the DID scheme. The second one specifies the DID Method used in the process of creation of the DID and that must be used in the process of resolution.
The last one is a method-specific identifier, which ensure the uniqueness of the DID in relation to that DID Method, and points to a specific memory location, of the DLT, containing the DID Document.

Figure 2.4. A simple example of a decentralized identifier (DID) [4].

2.3.1 DID Document

A DID Document contains a set of data describing the DID subject, such as cryptographic material used to authenticate and prove association with the DID. Typically this document format is JSON and may contain different keys, used for different purposes and saved under fields, whose names refer to the use made of them.

In the Figure 2.5 we can see an example of DID Document. Looking at the list of attributes (there may be others) we find:

- `@context`: contains a URL pointing to a resource describing the DID Document scheme.
- `id`: indicates the the subject to which the document refers.
- `created`: includes a timestamp of when the document was created.
- `authenticationMethod`: this field is the most important and as the name suggests, contains the informations about the cryptographic material used for authentication processes. It contains public keys, such as RSA keys, inside the `publicKeyPem` field. Each key is associated with the type, recognizable by the attribute `type` and an `id` that is the concatenation of the DID of the `controller`, the one in control of the key, and a string ("#keys-1"), in order to make it unique within the document. The `controller` very often coincides with the DID subject, but sometimes it can be a third party to which the use of that key has been delegated. This is a method example but there could be also others like `assertionMethod`, which may contain a key used in the Verifiable Credentials verification process (See Section 2.4).

Finally, the standard for DIDs provides for further fields that will not be explored in this discussion, as they were not so relevant for this project.

```
{
    "@context": [
        "https://www.w3.org/ns/did/v1"
    ],
    "id": "did:example:123456789abcdefghi",
    "created": "2022-01-10T02:01:35.923161",
    "authenticationMethod": {
        "id": "did:example:123456789abcdefghi#keys-1",
        "type": "RsaVerificationKey2018",
        "controller": "did:example:123456789abcdefghi",
        "publicKeyPem": "-----BEGIN PUBLIC KEY-----
                        ..........................
                        ..........................
                        -----END PUBLIC KEY-----\n"
    }
}
```

Figure 2.5. DID Document example.

2.4 Verifiable Credentials (VCs)

A Verifiable Credential is a data structure containing a set of attributes related to an subject, serialized in a cryptographically verifiable format, representing all the information that a physical credential represents. Unlike their physical counterpart, VCs are more tamper-evident and more trustworthy and bring a lot advantages not only to individuals but also to the organizations that issue and verify them. For example a Verifying organization can verify the credential instantly without needing to contact the entity that issued it. A Issuing organization, instead, can release the credential in a fast and secure way, reducing the manual work needed and the risk of fraud.

In the VC data model there are three principal components involved: *claims*, *credentials* and *presentations*.

Claims are statements about a subject and consist of subject-property-value relationships. This model permits to express a large variety of information about a subject by combining together different set of claimns. In order to be able to trust these claims and have a valid Verifiable Credential, other pieces of information are needed.

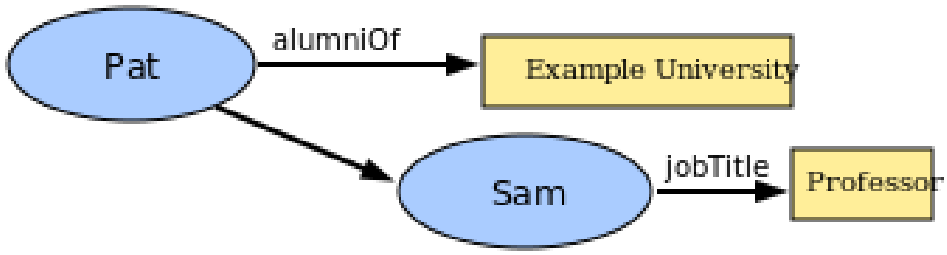

Figure 2.6. Multiple claims can be combined to express a graph of information [6].

A credential is a set of one or more claims made by the same entity along with a proof, which is usually a digital signature, to detect tampering and verify the authorship of the credential. There could be also some metadata describing for example the issuer, the expiration date and the public key to be used for the verification of the proof. A subject may be in possess of different credentials and would like, in certain situations, to be able to show more than one to a verifier or sometimes only a portion of them. The information that a subject shares with a verifier, regarding his credentials, are called verifiable presentations.

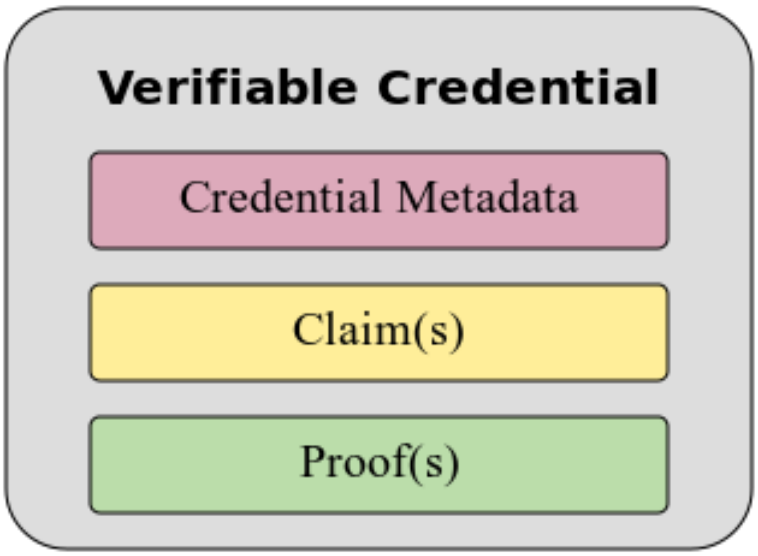

Figure 2.7. Basic components of a verifiable credential [6].

A presentation is a data structure expressing data from one or more verifiable credentials and is composed in a way such that is possible to verify the authenticity, integrity and validity of the information contained. The data in a presentation, since could be coming from different credentials, might have been issued by different actors. It is also present a proof but in this case is made by the Holder of the credentials.

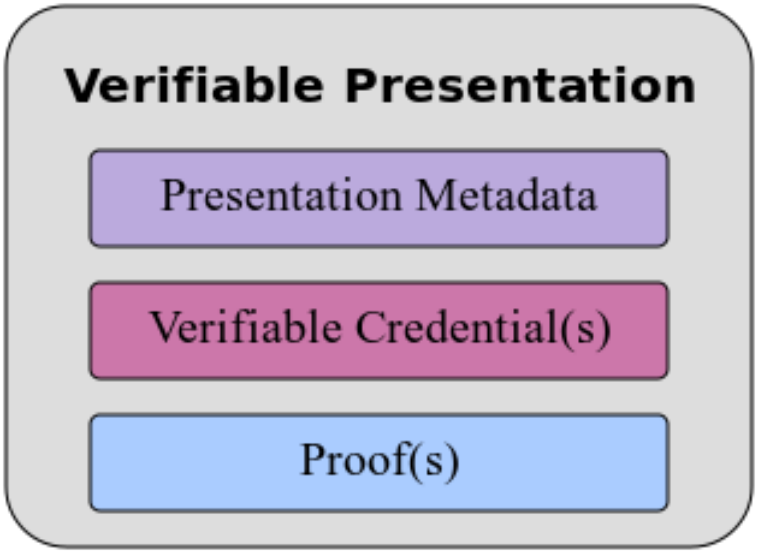

Figure 2.8. Basic components of a verifiable presentation. Image taken from [6]

2.5 DLTs, DIDs and VCs together

All the concepts explained so far, can be mapped/embedded in the so-called Trust Triangle, described in Section 2.1. The interactions among the entities employs the DLT as the Root-of-Trust and the concept of DID and VC as means that allow peer authentication.

Let us take as an example an Holder who wants to access the service provided by a Verifier, but in order to do it he has to create his digital identity and authenticate himself to the Verifier.

The pricipal steps occurring during this process are:

1. The three peers proceed to create their DIDs together with some cryptographic material, that consists of two key pairs, where the public portions are going be saved into the fields `authenticationMethod` and `assertionMethod` of the DID Document. Respectively the first key pair will serve to establish a secure comunication channel between them, while the second one to create and validate the proof linked to the Verifiable Credential. To do all this they have to interface with a DID Method and a DLT.

2. The holder request to the Issuer the Verifiable Credential he needs to access the service provided by the Verifier. First a new connection is opened between Holder and Issuer and an exchange of their DIDs takes place. Then they respectively read the DID Document of the other from the DLT. They now have the cryptographic material needed to establish a secure communication channel via a challenge-response protocol.

3. After that, the Issuer releases the credential together with a cryptographic proof generated through his private key created previously. To validate the

proof the Holder needs to verify the signature contained in it through the public key found in the Issuer's DID Document.

4. Finally the Holder establish a new comunication channel with the Verifier following the same steps as before. The user then proceeds to send the Verifier a Verifiable Presentation signed by himself that contains the identity information required for authentication. Now that the Verifier has received the presentation and has read the Holder and Issuer DID Documents from the DLT, can proceed with verifying the VP and VC proofs. If everything goes well, the last step is to assess whether the Issuer is trusted or not. This can be done thanks to a list of peers that the verifier interrogates and in which an implicit trust is placed.

2.6 IOTA

IOTA [7] IOTA is a non-profit foundation that developed the Tangle a feeless distribute ledger technology designed to run on IoT devices. The Tangle model is based on Directed Acyclic Graph DLTs, and therefore shares their features and advantages described in the Section 2.2.

The problem with IOTA and their Tangle is that currently is present a central node called "Coordinator" which is in charge to publish with regularity a new transaction called "Milestone". This is a signed message trusted and used by all the other nodes to confirm other transactions. In fact *"messages in the Tangle are considered for confirmation only when they are directly or indirectly referenced by a milestone that nodes have validated"*[8]. So for the moment, the "Tangle" can be seen as a solution non completely decentralized, but the IOTA Foundation stated that this is a temporaly solution and that from the next version IOTA 2.0 [9], the Coordinator will no longer be present.

To allow an IoT device to easily comunicate with the Tangle, the Cybersecurity team at the LINKS Foundation [10] developed a layer 2 protocol called WAM.

2.7 Wrapped Authenticated Messages (WAM)

WAM is a cryptographic protocol for exchanging data with the IOTA Chrysalis Tangle. It allows to write and read data on the Tangle in a secure way. It is designed to be executed on an IoT constrained environment, thanks to its small memory footprint and the possibility to compile it even without the need of an operating system.

The WAM protocol makes use of Chrysalis indexation payload to write/read data on Tangle. The indexation payload is made of an index and some arbitrary data which in this case represent the WAM message. WAM makes use of this design to link data with each other like a chain, where each data points to the next one via its index. If someone wants to read the last written data, he just needs to start from any index and walk the chain to the end.

Finally, since by default the data written on the Tangle are in plaintext, WAM takes care of security by encrypting and anthenticating the data before every write/read operation, using a method called Authenticated Encryption with Associated Data (AEAD)[11].

A WAM message is composed by different fields:

- APPDATA_LEN
- APPDATA
- PUB_KEY
- NEXT_IDX
- SIGN

APPDATA and APPDATA_LEN contain the data to be written on the Tangle and their length. The NEXT_IDX field is the information needed to construct the chain of messages. It contains the index of the next IOTA Chrysalis message, which encapsulates the next WAM message. The algorithm which generates the indexes can be summarized in this way:

1. starting from a random source, two seeds are generated
2. from the two seeds, two key pairs are generated
3. a digest of the public key of the first pair is calculated and that is the message index
4. the public key used before is saved in the file PUB_KEY of the message for future verifications
5. the same thing is done for the other public key and that is the NEXT_IDX

The SIGN field contains a digital signature of previous fields digest computed with the private key corresponding to the public key used in the computation of the index and saved in PUB_KEY. This field is not for authentication purposes, but only serves to maintain the integrity in the message chain.

When a message is read, two verifications are performed. The first one is the verification of the signature with the PUB_KEY by recomputing the digest of the fields. The second one is the index verification by computing the digest of the PUB_KEY and see if it matches the message index. This is done to prevent an attacker from hijacking the next message of the chain. A malicious user can read the NEXT_IDX field from the last message and know the index where to write his data. But to be able to write a valid message that passes the checks, he would need to know the public key from which the index was generated in order to insert it in the PUB_KEY field of his message.

An additional security feature offered by WAM is data confidentiality, since any information present on the Tangle is publicly available. The encryption is computed

through a nonce and a symmetric key, pre-shared betweek a group of IoT devices. The PSK is needed to restrict the access to data to anyone outside that group. Note that this solution has the problem that the PSK represents a single point of failure, in fact if an attacker is able to discover the encryption key the whole system is compromised. However this solution is temporary and the WAM developers at LINKS Foundation [10] are already working on a more robust mechanism that does not make use of a shared secret which is difficult to protect.

2.8 TLS 1.2

Transport Layer Security (TLS) [12] is a cryptographic protocol designed to provide secure communication channels over a network. The TLS protocol aims mainly to provide confidentiality, integrity, and authenticity through the use of X.509 certificates [13], between two or more endpoints. This protocol evolved from another encryption protocol called Secure Sockets Layer (SSL), that was developed by a company called Netscape. In fact, initially the TLS 1.0 version was named SSL 3.1, but when the protocol was publicly presented the name was changed to clarify that was not anymore related to the Netscape company.

Since then the protocol has continued to evolve until the latest version released, which is the TLS 1.3. However, this section will present the version 1.2, which is the previous version of this protocol. And the reason is as follows. In this project the development environment is IoT constrained, where devices possess limited capabilities both from a power and memory point of view. For this reason, among the various existing implementations of the TLS, the Mbed TLS library was the most suitable.

Mbed TLS [2] is a C library that implements the TLS protocol together with various cryptographic primitives and utility functions. It has been designed to run easily on embedded devices thanks to its small memory footprint. Unfortunately at the time of writing, the long term support version of Mbed TLS was 2.28, which did not support the TLS 1.3 version and for this reason TLS 1.2 was the only available option.

Before analyzing the TLS 1.2 in detail, let us spend some words on the TLS protocol in general. The protocol phase leading to the creation of the secure comunication channel is called handshake. This phase, regardless of the protocol version considered, can generally be divided into three main steps:

- Negotiation: client and server negotiate the protocol version, the cipher suite and other parameters to be used during the handshake.
- Authentication: server and client (in case of mutual authentication) can authenticate themselves to the other party through different mechanism which make use of X.509 Certificates and digital signatures.
- Key Exchange: when client and server trust each other they can proceed to derive a symmetric session key that will be used for the encryption of data once the handshake phase is finished and the tls channel has been created.

To understand in particular how the TLS 1.2 protocol works, it is necessary to analyze the messages exchanged between client and server during the handshake phases previously summarized.

2.8.1 Handshake

The TLS 1.2 handshake is represented by the following diagram and investigated further below.

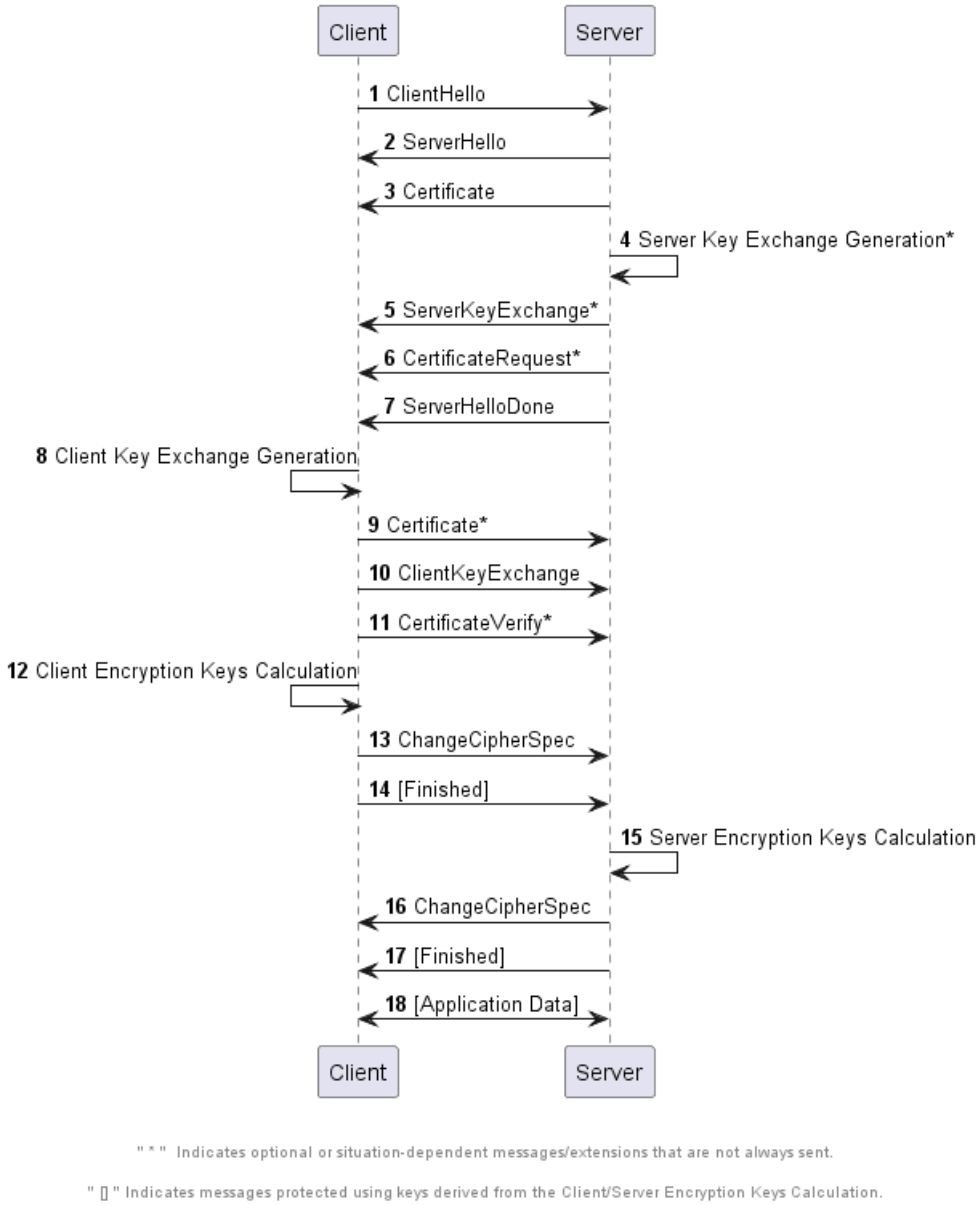

Figure 2.9. TLS 1.2 Handshake

The handshake starts with a negotiation phase that includes two messages: the ClientHello (Figure 2.10) and ServerHello (Figure 2.11).

First of all, client and server must agree on the protocol version to use, in this case TLS 1.2. Then they exchange a random nonce that will be involved in subsequent cryptographic computations. Finally, the two parties must agree on the algorithms to use during the rest of the handshake.
These algorithms are:

- Key exchange algorithm: used to exchange a symmetric encryption key
- Digital signature algorithm: used in the authentication process
- Encryption algorithm: used to provide confidentiality of the data transmitted after the handshake on the established TLS channel
- Integrity algorithm: used to provide integrity of the data exchange during and after the TLS handshake

A combination of these algorithms defines a so-called cipher suite. The client will communicate to the server via ClientHello the list of cipher suite that supports, then the server will choose one from this list and respond with the ServerHello. Now the two parties have successfully agreed on how to complete the rest of the handshake.

In figures 2.10 and 2.11 it is possible to notice that the two messages contain other information under the name of Extensions. The Extensions add functionalities that were not originally intended for the protocol. The current list of available extensions is maintained by IANA [14], where each of them is identified by a value recorded in the TLS ExtensionType Registry [15].

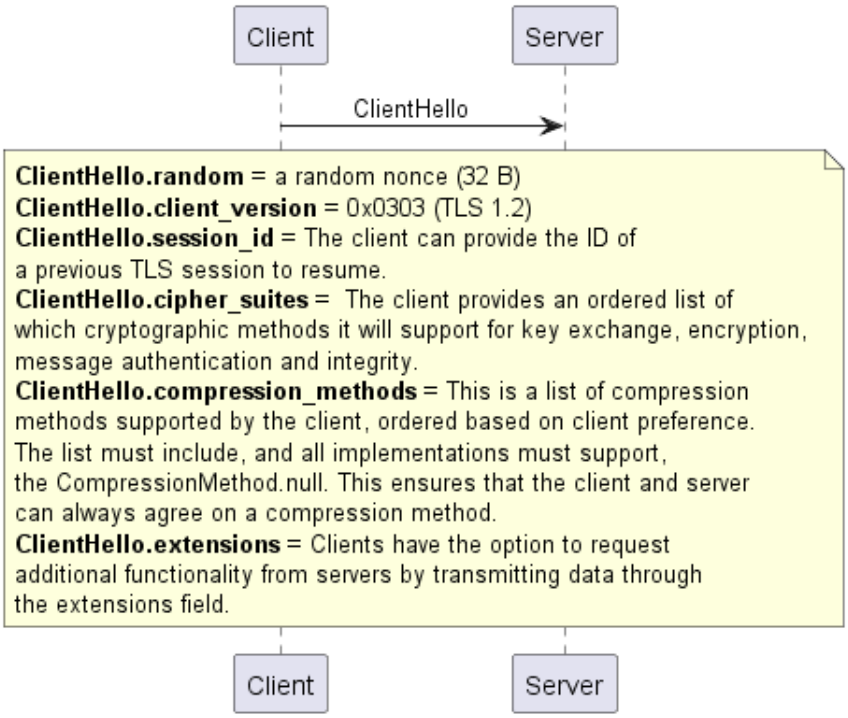

Figure 2.10. ClientHello

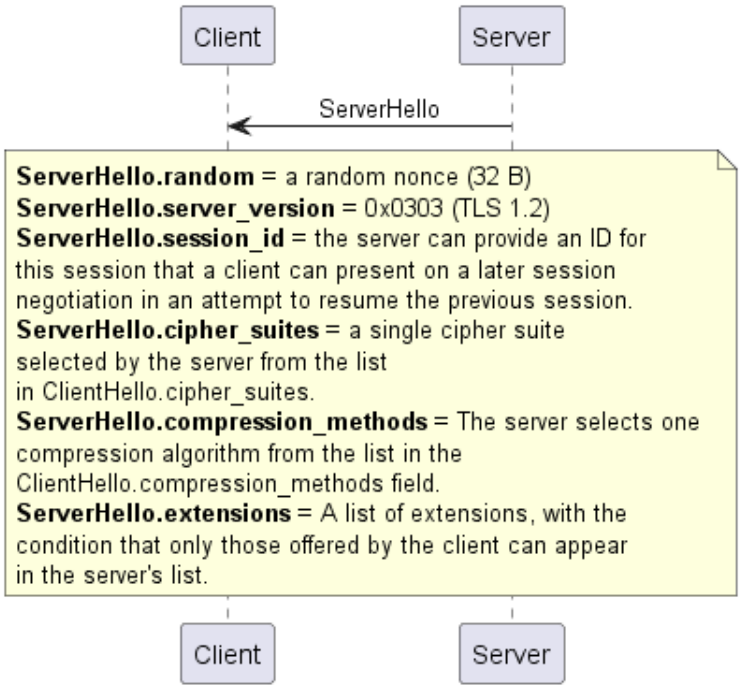

Figure 2.11. ServerHello

The server to authenticate itself sends a Certificate message (Figure 2.12) after the ServerHello. This message contains an X.509 certificate [13] representing the server's identity signed by a Certification Authority (CA). In the certificate is also specified the algorithm used in the computation of the signature, which must be supported and adopted by the client in the certificate's verification process.

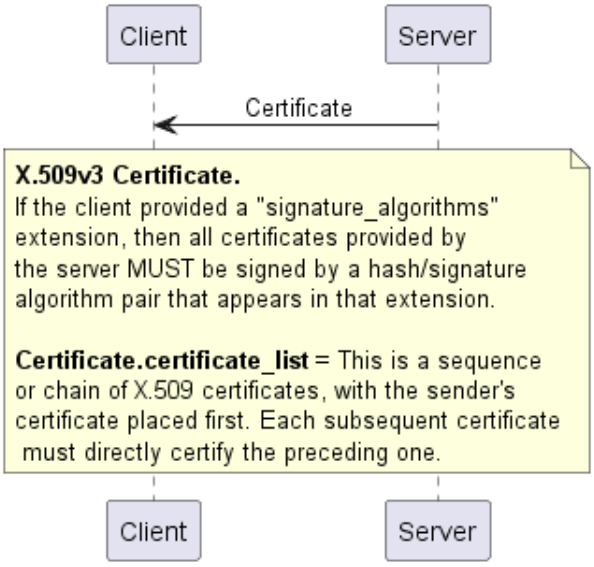

Figure 2.12. Certificate

The client, to check the validity of the certificate, verifies the signature, the chain of trust (certificate chain up to the root CA certificate) and the revocation/-expiration status. If these steps are successful, the client is certain that it has received a valid certificate, but is still not sure if the server is who it claims to be. In order to achieve this, the server needs to prove that it is the owner of the private key paired with the public key present in the certificate. The methods used to accomplish this vary according to the key exchange algorithm chosen during the negotiation phase, which could be based on RSA or Diffie-Hellman (Elliptic-curve Diffie-Hellman in case of Elliptic Curve algorithms). The successful outcome of both methods leads the client to trust the server and share a common value called pre-master secret with it, from which they can derive a session key used for the symmetric encryption of data.

In the RSA key exchange, the client creates the pre-master secret as a random sequence of bytes and use the server's public key retrieved from the certificate to encrypt it (Figure 2.13). Then it sends the result in the ClientKeyExchange message (Figure 2.14). The server receive the message and decrypts it with its private key to obtain the same value. In this case the server has implicitly proved to the client that it owns the private key, because otherwise it could not decrypt the message to get the pre-master secret and derive the session key.

In the (EC)DH key exchange, the pre-master secret is not generated by the client, but is derived by both party. Client and server generate a key pair (Figures 2.13 and 2.15) and send the public portion in the ClientKeyExchange (Figure 2.14) and ServerKeyExchange (Figure 2.16) messages respectively. If the server's certificate contains a (EC)DH key, then that key could be used and it would not be necessary to generate a new one to be sent in the ServerKeyExchange message, unless an "ephemeral" approach is adopted, as later explained. Now both sides have the material needed to calculate the pre-master secret by combining their private key with the other's public key. However, the client must be able to trust the server before continuing, which in this case has not yet authenticated itself. To prove its identity the server sends its (EC)DH public key together with a signature computed with its identity's private key, the one paired with the public key contained in the certificate.

The second approach has some advantages over the first, which made it the only key exchange algorithm available in TLS 1.3, in contrast to the RSA-based algorithm that has been deprecated.
The first advantage is that the pre-master secret is never sent over the network, but is calculated locally by both sides, making the computation of the session key much more secure.
The second advantage is definitely the most important one and concerns the concept of forward secrecy. The (EC)DH key pair can be regenerated at each session making a private key leakage less harmful, since an attacker would only be able to decrypt the traffic trasmitted during that particular session. In this case we talk about Ephemeral Diffie-Hellman key exchange algorithm.

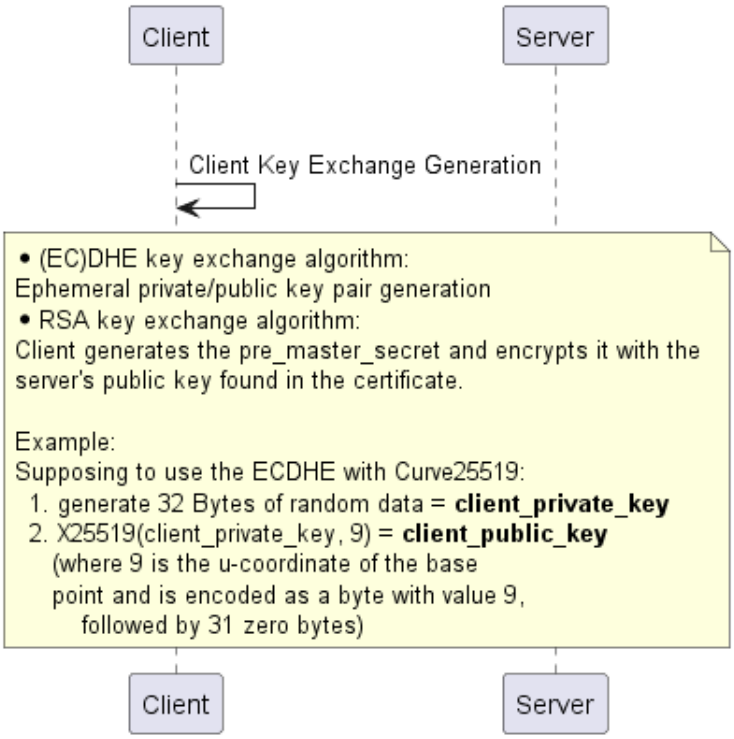

Figure 2.13. Client Key Exchange Generation

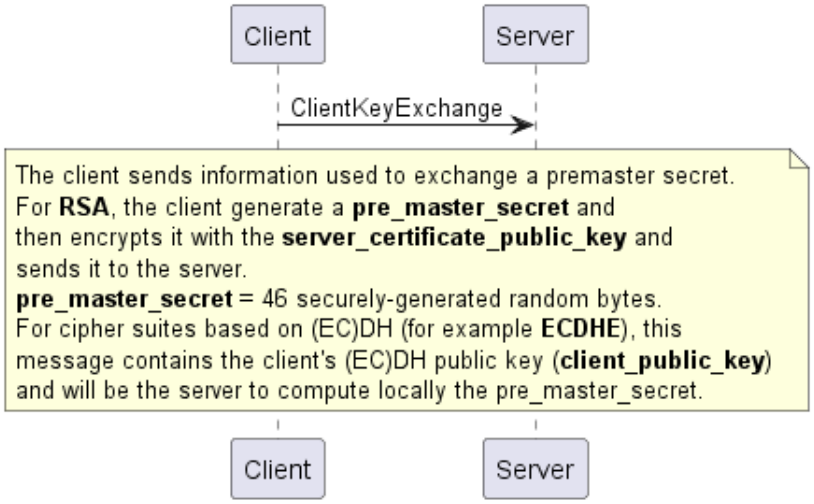

Figure 2.14. Client Key Exchange

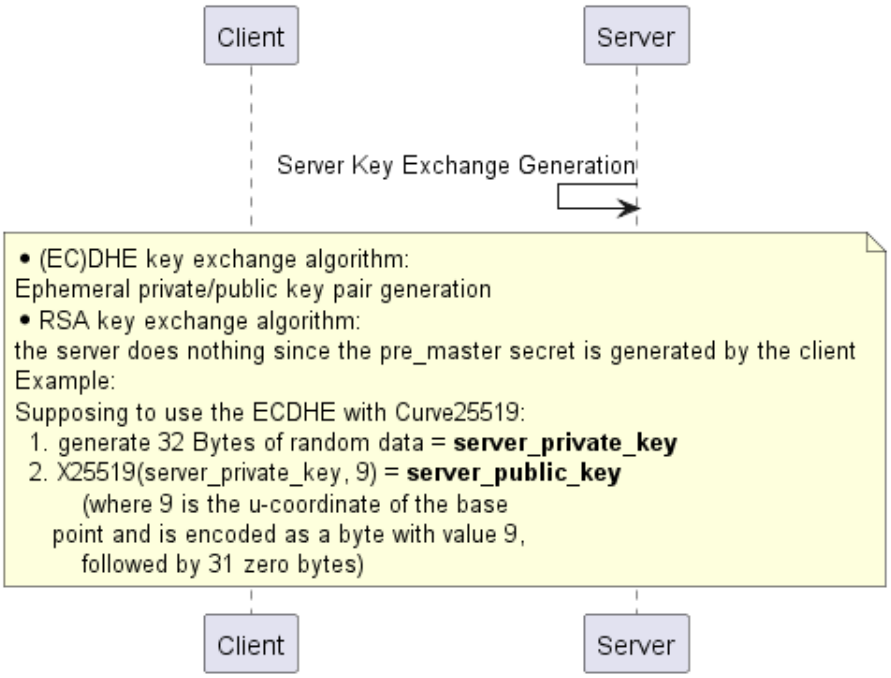

Figure 2.15. Server Key Exchange Generation

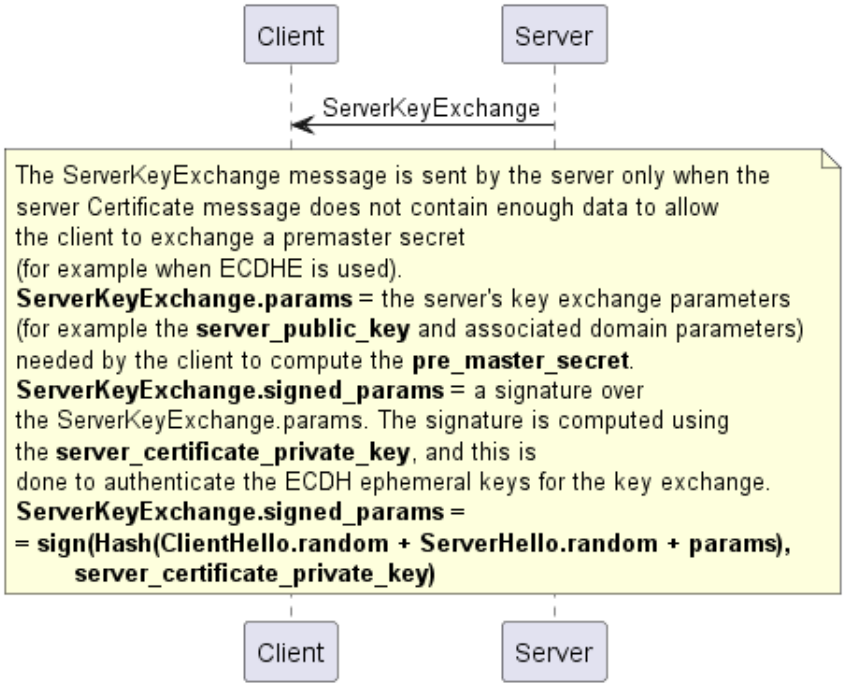

Figure 2.16. Server Key Exchange

Once the key exchange phase is completed, both client and server share a pre-master secret from which they can derive the symmetric session key. Therefore, the client sends a ChangeCipherSpec message, signaling that it has calculated the session key and that all following messages will be encrypted, followed by a Finished message (Figure 2.17) with the purpose to verify that the key exchange and

authentication processes terminated correctly and have not been tampered. The server repeat the same step(s) and replies with its ChangeCipherSpec and Finished messages (Figure 2.18). From now on the entire comunication is encrypted with the session key.

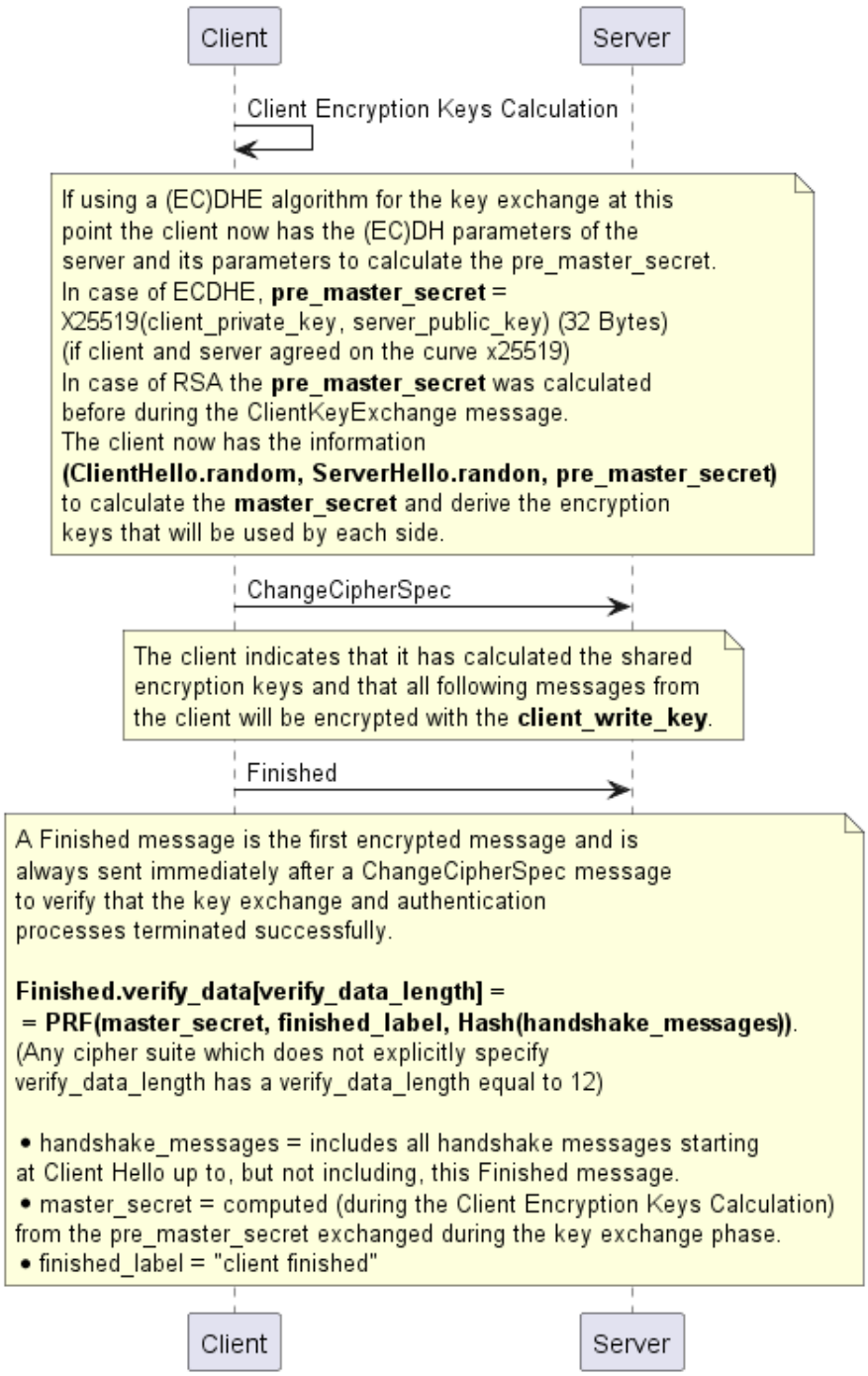

Figure 2.17. Client Encryption Keys Calculation, ChangeCipherSpec and Finished

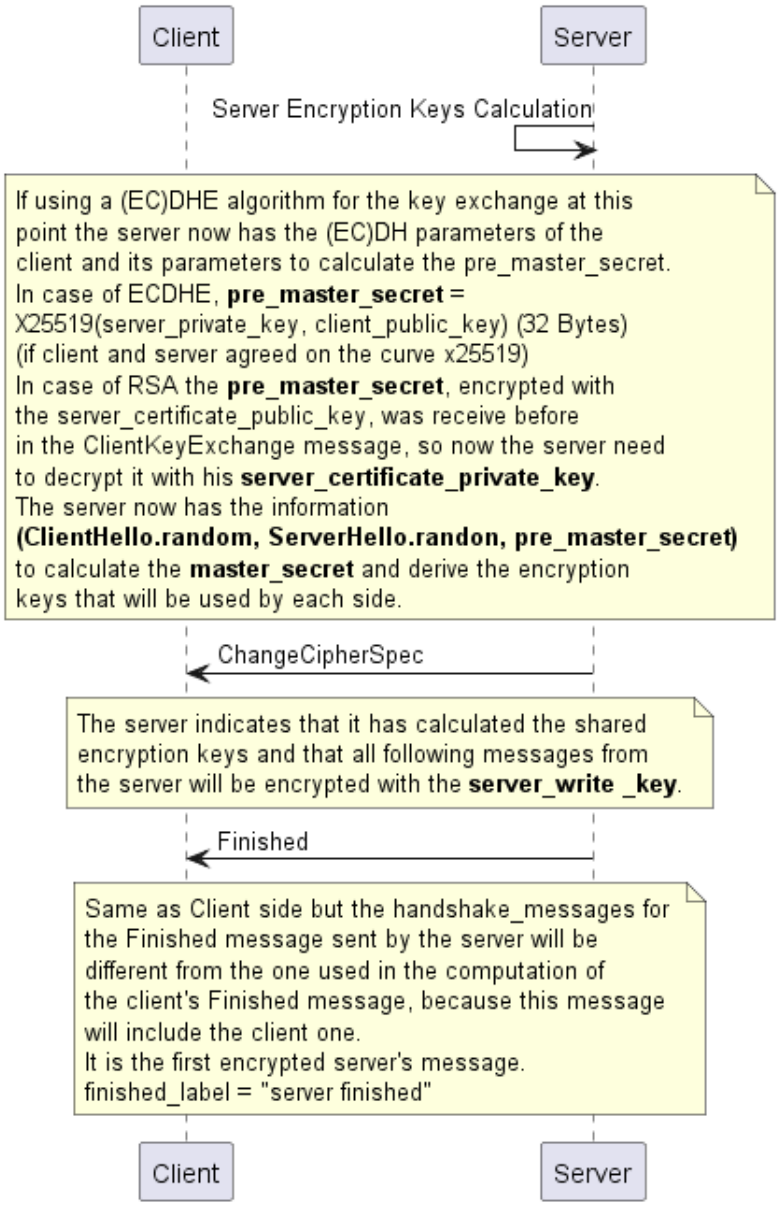

Figure 2.18. Server Encryption Keys Calculation, ChangeCipherSpec and Finished

In the previous analysis, it was discussed that in certain cases, the server needs to authenticate itself to the client. However, in some scenarios, mutual authentication may be necessary, where both the server and the client must verify each other's identities. In this case the server, after the ServerKeyExchange message, will send a CertificateRequest message (Figure 2.19), to which the client will respond with its certificate followed by the ClientKeyExchange message and a CertificateVerify message (Figure 2.20). The CertificateVerify message will contain a digital signature of the digest of all previously exchanged handshake messages. This has two purposes, the first is to prove to the server that the client has the private key paired with its certificate's public key, while the latter is to authenticate, in case of (EC)DHE key exchange algorithms, the ephemeral key generated during that session. Server side this proof was given by the signature of the (EC)DHE parameters contained

directly in the ServerKeyExchange message (Figure 2.16).

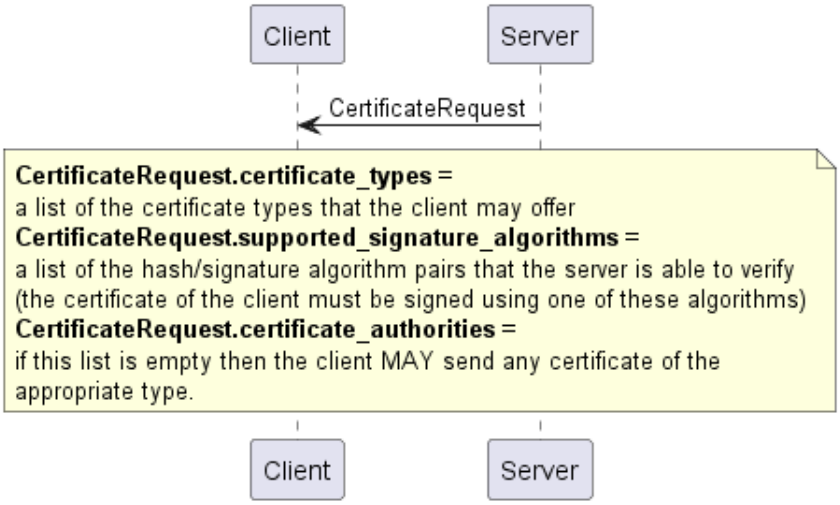

Figure 2.19. Certificate Request

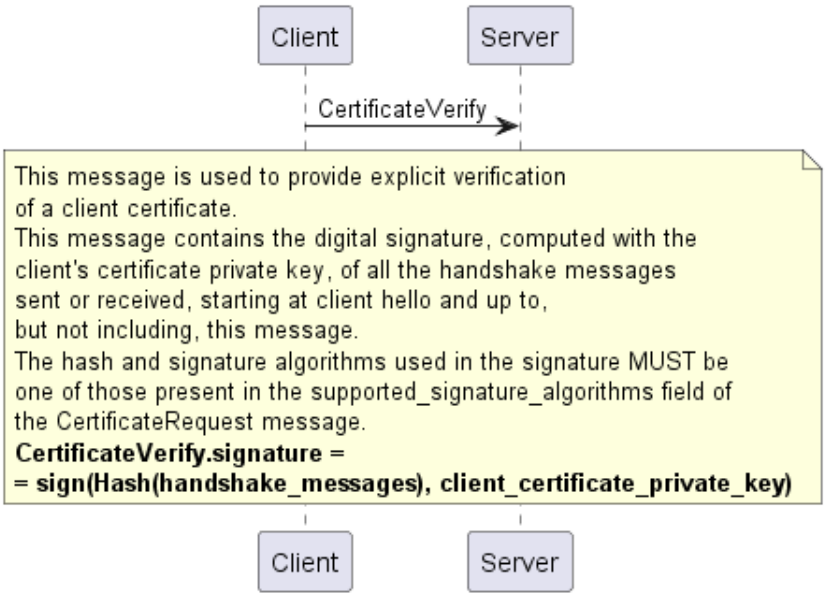

Figure 2.20. Certificate Verify

2.8.2 Computational details

This section describes in detail the computations which lead, starting from the `pre_master_secret`, to the generation of the `master_secret` and the respective symmetric keys, with a final reference to the `verify_data` present in the Finished message, since derived in the same way.

The RFC5246 [16] defines a data expansion function, `P_hash(secret, data)`, that uses a single hash function to expand a secret and a seed into an arbitrary quantity of data.

```
P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
                       HMAC_hash(secret, A(2) + seed) +
                       HMAC_hash(secret, A(3) + seed) + ...

A() is defined as:
       A(0) = seed
       A(i) = HMAC-Hash(secret, A(i-1))
```

P_hash can be iterated as many times as necessary to produce the required quantity of data.

The TLS protocol, to derive some of its secrets, internally makes use of a pseudo random function PRF defined this way:

```
PRF(secret, label, seed) = P_hash(secret, label + seed)
```

Now from the pre_master_secret we can derive the master_secret:

```
master_secret =
= PRF(pre_master_secret, "master secret", ClientHello.random
   + ServerHello.random)[0..47]
= P_hash(pre_master_secret, "master secret" +
   ClientHello.random + ServerHello.random)
```

The master_secret is 48 Bytes long, so only two PRF iterations are needed:

```
seed = "master secret" + ClientHello.random +
   ServerHello.random
A(0) = seed
A(1) = HMAC-Hash(pre_master_secret, A(0))
A(2) = HMAC-Hash(pre_master_secret, A(1))
master_secret = HMAC_hash(secret, A(1) + seed)[0..31] +
   HMAC_hash(secret, A(2) + seed)[0..15]
```

The last step is to derive the final encryption keys from the master_secret. The master_secret is expanded into a sequence of bytes, which are splitted to a client_write_MAC_key, a server_write_MAC_key, a client_write_encryption_key, and a server_write_encryption_key. Some AEAD ciphers may additionally require a client_write_IV and a server_write_IV for the implicit part of the nonce.

To generate the key material, compute:

```
key_block = PRF(master_secret, "key expansion",
   ServerHello.random + ClientHello.random)
```

until enough output has been generated.

Then the key_block is partitioned as follows:

```
1. client_write_MAC_key[mac_key_length]
2. server_write_MAC_key[mac_key_length]
3. client_write_key[enc_key_length]
```

4. server_write_key[enc_key_length]
5. client_write_IV[fixed_iv_length]
6. server_write_IV[fixed_iv_length]

The same pseudo random function described above is applied in the calculation of the verify_data field contained in the Finished message. The verify_data is built from the master_secret and the digest of the payload of all the handshake records exchange.

```
Finished.verify_data[verify_data_length] = PRF(master_secret,
    finished_label, Hash(handshake_messages))
```

In details:

```
seed = "client finished" + SHA256(all handshake messages)
A(0) = seed
A(1) = HMAC-SHA256(master_secret, A(0))
verify_data = HMAC-SHA256(master_secret, A(1) + seed)[0..11]
```

Now that the concepts of TLS and in particular of TLS 1.2 have been better deepened, the next step is to dig into the details of a new DID Method, used for developing an SSI-aware version of the TLS 1.2 handshake, described in the Section 3.2.

Chapter 3

Design and Implementation

In this chapter is explained the work done to develop a new DID Method and a model of the TLS handshake based on the SSI paradigm.

3.1 SnD DID Method

The DID Method used in this project for the creation of an SSI-aware TLS handshake has been named SnD and was specifically created for this project. It interfaces with the IOTA Tangle using the WAM protocol described in section 2.7, developed by the LINKS Foundation. The CRUD operations offered by this method are based on the `WAM_read`, `WAM_write`, and `WAM_channel` primitives. The `WAM_channel` contains information necessary for opening a communication channel to the Tangle, such as the node to connect to, the index for reading or writing a message, and a pre-shared key for data confidentiality. Essentially, this DID Method acts as a wrapper for the WAM protocol.

In the figure 3.1 below it is possible to have a clearer image of what is meant by channel and how the operations on the DIDs work.

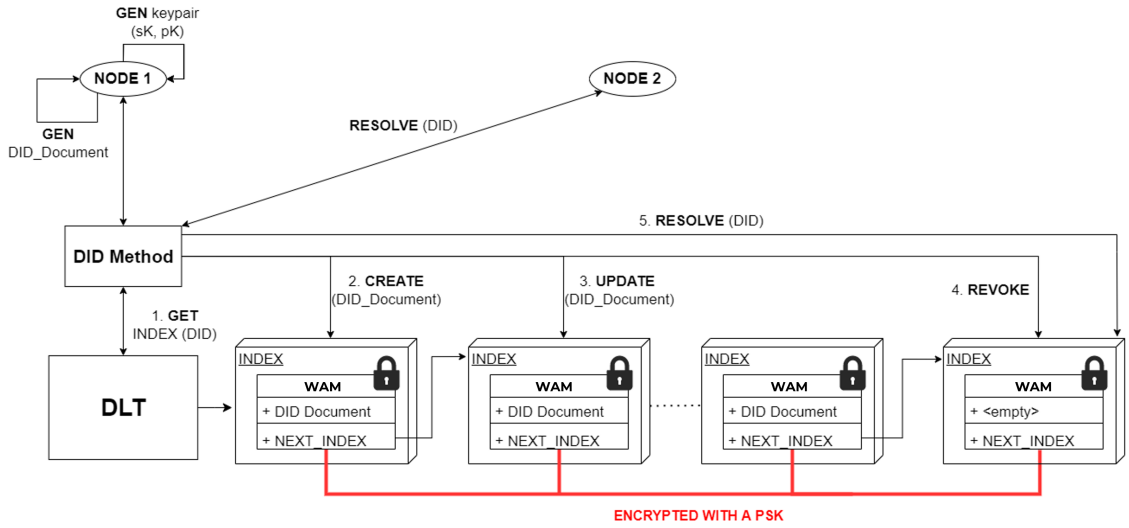

Figure 3.1. DID Method: SnD

The scenario includes two nodes, one that wants to create its own identity, update and revoke it, while the other one deals only with recovering the identity of the first starting from its DID. The first node starts with the generation of its key pair. The public part will be inserted inside the created DID Document. When the `CREATE` function of the DID Method is called, a communication channel with the underlying DLT is immediately created and a pair of indices is generated. The first index represents the starting point of the channel, so where the first message will be written, while the second index represents the `NEXT_INDEX` field of the message and indicates the position where the next message will be written. The `CREATE` function is also responsible for generating, from the public key of the node, the DID Document in JSON format to be inserted in the WAM message. At the end the `CREATE` function returns to the caller its new DID, where the DID Method-Specific Identifier (Figure 2.4) is nothing more than the `INDEX` in which the DID Document was written.
When the node wants to update its identity, for example by changing its key pair, it will invoke the `UPDATE` function, which will generate a new DID Document and always write it to the last generated `NEXT_INDEX` in the chain.
The `REVOKE` operation, instead, is simply an `UPDATE` operation where where an empty message is written.
Finally the second node wants to retrieve the DID Document of the first one. Once it has received the other's DID, it calls the RESOLVE operation, which takes in input the DID, estracts the INDEX (Method-Specific Identifier) and starts a loop of read operations. The purpose of this loop is to be sure to always retrieve the last valid DID Document of the chain, and works by following the `NEXT_INDEX` chain as long as there is a valid message and a not revoked DID Document.

It is important to note that every message written on the channel, and therefore belonging to the chain, is encrypted using a pre-shared key. So only those who have the key will be able to successfully complete the `RESOLVE` operation of a DID. This permits to restrict the access to the channel to a limited number of nodes sharing the PSK. The same protection is applied in the opposite case, where an external

node not knowing the PSK tries to write its message into the chain. In this case the other nodes will recognize that message as invalid and ignore it.

3.1.1 APIs

The SnD Method exposes four APIs:

```
int create_(method_ *methods, char *did_new)
```

This function creates the DID Document and saves it on the IOTA Tangle. It takes in input a `did_new` pointer, that at the end will contain the generated DID, and a `methods` parameter, which contains the information, such as a public key, to be included in the `authenticationMethod` (Figure 2.5) and `assertionMethod` fields. The function starts by creating a structure `WAM_channel` representing the channel. This structure contains also the starting index where the message will be written, the next index and the pre-shared key for the encryption. Then the DID Document in JSON format is created and inserted in a WAM message. Finally this message is sended to the Tangle by calling the `WAM_write` function. If this operation is successful the state of the channel is saved in memory for future operations.

```
int update_(method_ *methods, char * did)
```

This function updates the DID by creating a new DID Document to be written on the DLT at the end of the channel's chain of messages. So the first thing that it does is loading the channel state previously saved containing the endpoint and indexes information. Then it generates a new DID Document starting from the `methods` parameter, as the `create_` operation, and writes it at the last index of the chain. The new DID is return to the caller through the `did` pointer parameter.

```
int revoke_(char *did)
```

This function permits to revoke a DID Document. It works the same way as the `update_`, but the only difference is that the message written on the Tangle is empty and the NEXT_INDEX fields is filled with all zeros.

```
int resolve_(did_document_ *didDocument, char *did)
```

This function permits to retrieve a DID Document starting from the corresponding DID. The `didDocument` parameter represents a structure to be filled with the DID Document's information retrieved from the Tangle, while the `did` parameter contains the DID to be resolved. This function starts with the initialization of a new WAM_channel. In order to do this the pre-shared key and the Tangle endpoint information are needed. The channel index is estracted from the DID. Then this function performs a WAM_read operation in loop, starting from the index setted before and at every cicle updating the read index, until the last message of the chain is encoutered. If this message contains a valid and not revoked DID Document then its value is returned through the `didDocument` parameter.

3.2 SSI-aware TLS 1.2 handshake

In this section will be presented the analysis and implementation choices that led to the modification of a protocol, robust and widespread, such as TLS and the development of a new model based on a decentralized digital identity paradigm called Self-Sovereign Identity.

3.2.1 Case study

The goal is to integrate the use of DIDs into the TLS handshake in order to establish a secure channel in the Layer 2 of the SSI stack (Figure 2.2). With DIDs, a decentralized authentication mechanism is introduced, which means that no central authority is needed to certify one's identity. Unlike the use of X.509 certificates, where the root of trust was the Certification Authority that issued the specific certificate, in this case it becomes the DLT.

To avoid to re-implement the majority of the TLS protocol functionalities, an existing cryptographic library for secure communication has been considered. A suitable library should enable a widespread adoption of SSI paradigm, such as constrained IoT devices. In particular for this project were used STM32 Discovery boards (Figure 3.2), produced by STMicroelectronics [17], equipped with an STM32L4S5VIT6 MCU, based on Arm®Cortex®-M4 core, with only 640 KB of RAM.

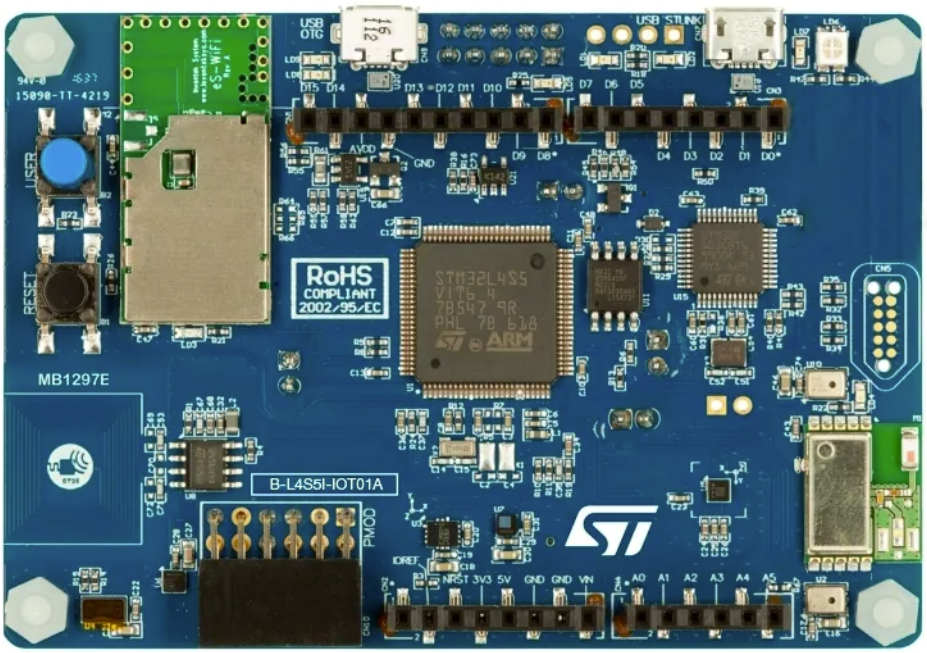

Figure 3.2. STM32 Discovery board

For this reason, the library choice fell on Mbed TLS [2], a lightweight and fast library with a small memory footprint. At the time of development, the latest LTS (Long Term Support) of the library is v2.28, which supports TLS 1.2 [16].

3.2.2 Model

Starting from the TLS 1.2 handshake, a model based on the SSI paradigm was developed, in which the use of certificates was replaced by the use of Decentrilized Identifiers.

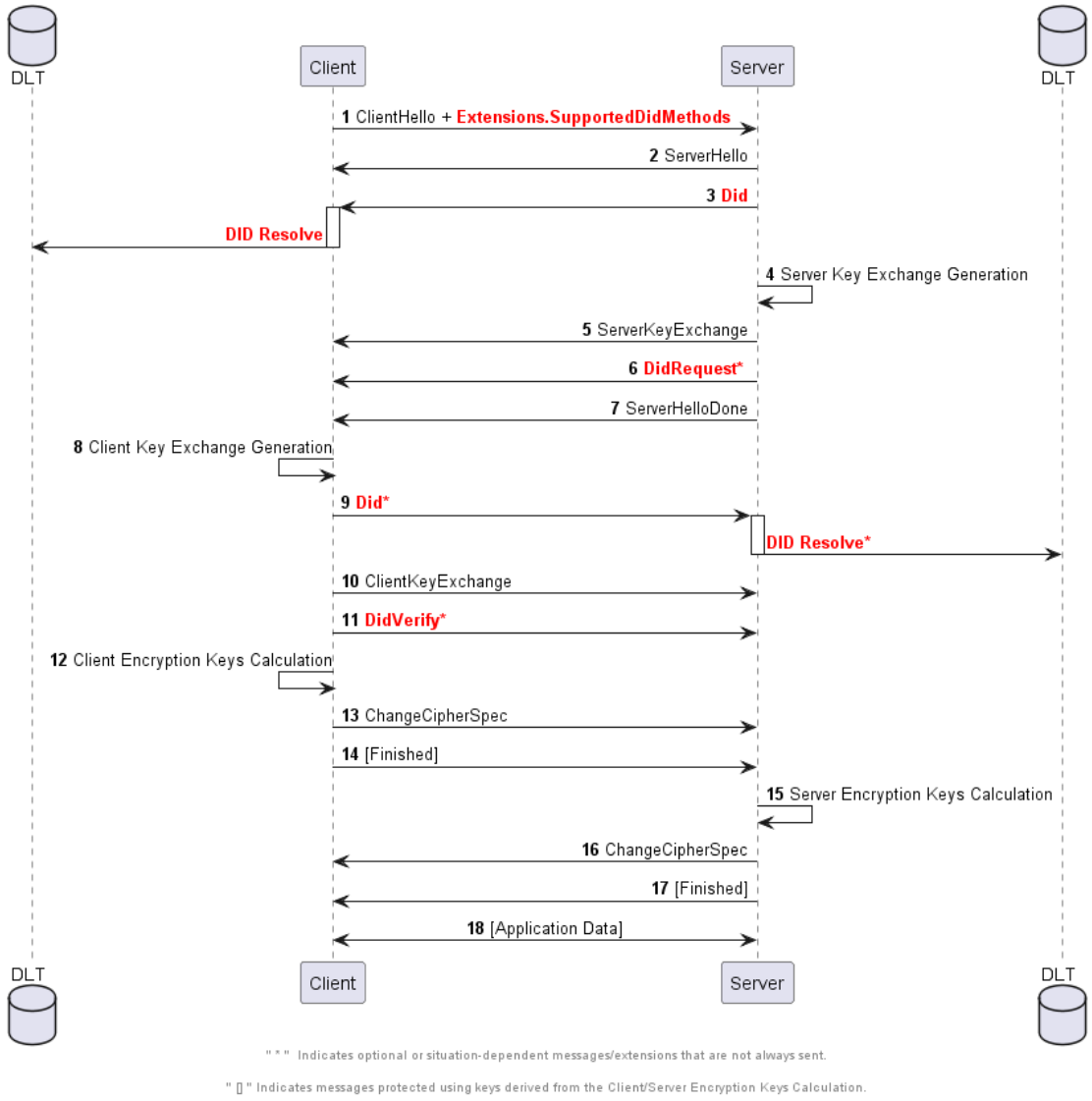

Figure 3.3. SSI-aware TLS 1.2 handshake

The handshake always begins with the negotiation phase in which client and server agree on the cipher suite and on the other session parameters to be used in the remaining part of the handshake. In this model it was assumed to use only cipher suites based on ephemeral key exchange algorithms which are more secure. To enable the DID support, a new extension has been introduced in the ClientHello called **SupportedDidMethods**. The latter has two purposes: the first is to communicate to the server that the client intends to proceed with an authentication using DIDs instead of X.509 certificates, the second is to provide the server with the list of DID Methods supported by the client, so that it can verify that it has a DID generated from one of them and that the client can "resolve".

The connection is aborted if the server doesn't have a DID compatible with any of the client's DID Methods. Otherwise, if this extension is not present in the ClientHello, the handshake continues using the X.509 certificates, if supported.

At the end of the negotiation phase, the server will send a new message called **Did** containing its DID and an identifier indicating the DID Method to be used for the "resolve" operation. On the client, the corresponding DID Resolve is performed to retrieve the DID Document of the server containing its public key.

Then the handshake continues with the key exchange phase, in which is assumed to use an ephemeral algorithm as mentioned above. It starts on the server side with the generation of the ephemeral key pair, where the public part is sent to the client in the ServerKeyExchange message along with its digital signature.
Subsequently, in case the server needs the client authentication, a new message called **DidRequest** has been introduced, having the same purpose as the CertificateRequest message, but with some differences in its content.
Its purpose is therefore to report to the client that its authentication is required and that it must be done with DIDs. This message contains a list of DID Methods supported by the server, along with the `supported_signature_algorithm` field that was already present in the CertificateRequest.

At this point, the server reports via the ServerHelloDone message that it has completed its key exchange phase and that the client can proceed with its own. At this point the server reports via the ServerHelloDone message that it has completed its key exchange phase and that then the client can continue with its own. The client then proceeds with the sending of the **Did** message containing its DID, which must respect the constraints required in the DidRequest message. This step is performed exactly as server-side, but only if the client has previously been explicitly requested to authenticate via the DidRequest message. Otherwise the client continues directly with the key exchange phase by sending the ClientKeyExchange message.

Finally, the latest new message introduced in this model is called **DidVerify**. It is used by the client to provide an explicit proof of possession of the identity private key and it is only sent if the client has to authenticate and has previously sent a valid **Did** message. Its purpose and content is identical to that of the CertificateVerify message, the only difference is the latter is used during certificate based authentication.

After that, the handshake flow continues with the session key derivation and terminates with the Finished messages exchange, according to the TLS protocol (see Sect. 2.8).

3.2.3 Implementation details

Before delving into a formal definition of the new TLS extension and the three new TLS messages introduced, it is essential to have a comprehensive understanding of certain concepts such as "variable-length vectors," "opaque" types, and "enum" types.

"Variable-length vectors" are specified by indicating a range of valid lengths, including both the lower and upper bounds, using the notation `<floor..ceiling>`.

When these vectors are encoded, their actual length is placed before their contents in the byte stream. The length is represented as a number that takes up as many bytes as necessary to represent the vector's ceiling length.

The "opaque" type is used represent entities that consist of a single byte and contain unprocessed data.

In the following example, `vector_example` is a vector that must have between 100 and 400 bytes of the "opaque" type and cannot be empty. The actual length field takes up two bytes, which is large enough to hold the value 400.

```
opaque vector_example<100..400>
```

Finally, a new type of data called `DidMethod` needs to be introduced, which is defined as an "enum". Data of this type can only take on the values that are specified in its definition. Each definition creates a different type, and enum values can only be assigned or compared if they are of the same type. The `DidMethod` will consume one byte in the data stream, but it can only take on the value of 0 for now, as the "SnD" method is the only one available in this TLS model.

```
enum { snd(0), (255)
} DidMethod;
```

SupportedDidMethods Extension

This extension, as mentioned above, indicates that the client is willing to use the DID as the authentication method and also contains the list of DID Methods supported by the client.

From RFC [16] an extension is defined this way:

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

The first step to do is to define a new ExtensionType value that has not already been taken. For this extension the value "60" has been chosen.

```
enum {
  (...), supported_did_methods(60), (65535)
} ExtensionType;
```

The final step is to determine the contents of the extension, which is the information stored in the "extension_data" field. The "extension_data" field of this extension contains a value of type `SupportedDidMethods`, which is basically a variable-length vector of DidMethod enum values defined in this way:

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

Did Message

This message serves as an alternative to the Certificate message, which is used for authentication with X.509 certificates. It contains the DID of the client or server, along with a byte that indicates the DID Method used to create the DID.

In order to define a new handshake protocol message, it is necessary to assign first a new HandshakeType value to it. This value will identify the new message. For this message the chosen value was "30".

```
enum {
  hello_request(0), client_hello(1), server_hello(2),
  certificate(11), server_key_exchange(12),
  certificate_request(13), server_hello_done(14),
  certificate_verify(15), client_key_exchange(16),
  finished(20), certificate_url(21), certificate_status(22),
  did(30),
  (255)
} HandshakeType;
```

It is now possible to define a new message in this manner:

```
struct {
  HandshakeType msg_type;
  uint24 length;
  select (HandshakeType) {
    case hello_request: HelloRequest;
    case client_hello: ClientHello;
    case server_hello: ServerHello;
    case certificate: Certificate;
    case server_key_exchange: ServerKeyExchange;
    case certificate_request: CertificateRequest;
    case server_hello_done: ServerHelloDone;
    case certificate_verify: CertificateVerify;
    case client_key_exchange: ClientKeyExchange;
    case finished: Finished;
    case certificate_url: CertificateURL;
    case certificate_status: CertificateStatus;

    case did:    Did;
  } body;
} Handshake;
```

Where `Did` is defined as a structure where the first field represent the DidMethod value (`did_method`) and the second one is a variable-length vector containing the DID value (`did_value`).

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

DidRequest Message

This message will be sent by the server, instead of the CertificateRequest message, in order to request client authentication. This message includes a list of methods and a list of signature and hash algorithms supported by the server. This information allows the client to select a valid DID that meets the requirements of the server. The structure of this message is very similar to that of the CertificateRequest message, since the purpose is pretty much the same, except that the latter message includes a list of Certificate types and Certification Authorities trusted by the server.

As previously mentioned, to properly define this message according to RFC standards, a new HandshakeType value must be assigned to it.

```
enum {
  (...), did_request(31), (255)
} HandshakeType;
```

Then this Handshake message can be defined in the following manner:

```
struct {
  HandshakeType msg_type;
  uint24 length;
  select (HandshakeType) {

      ...

    case did_request:   DidRequest;
  } body;
} Handshake;
```

With the `DidRequest` defined as a structure that includes a variable-length vector of DidMethod values (`supported_did_methods`) and a variable-length vector of SignatureAndHashAlgorithm pairs (`supported_signature_algorithms`). The formal definition for the latter can be found in RFC 5246 [16]:

```
struct {
        DidMethod supported_did_methods<1..2^8-1>;
        SignatureAndHashAlgorithm
           supported_signature_algorithms<2..2^16-1>;
} DidRequest;
```

DidVerify Message

The last Handshake message to describe is the `DidVerify` message, which will be sent instead of the `CertificateVerify` message. It is used by the client to provide explicit proof of possession of the idenitity private key. This message is structurally identical to the `CertificateVerify` message. Therefore, the need to introduce a new message is simply to ensure consistency and clarity in the flow of the protocol. The exchange of a message called `CertificateVerify` during a DID-based SSI-aware handshake might lead to some confusion.

To the DidVerify message was assigned the HandshakeType value of "32".

```
enum {
  (...), did_verify(32), (255)
} HandshakeType;
```

To view the structure of the `CertificateVerify`, which has the same structure of this message, refer to RFC 5246[16].

3.2.4 Security Assessment

The changes made by this new model to the TLS protocol have made it necessary to conduct a security assessment in order to verify that the modifications have not introduced any security vulnerabilities in this SSI-aware version of TLS. The robustness of the protocol has been verified against two common man-in-the-middle(MITM) attack scenarios.

The two MITM attacks that will be examined are based on the assumption that the attacker does not have a valid DID. Unfortunately, it is not possible to verify the identity of the entity with whom one is communicating when using a DID-based authentication, as it is with a certificate. While it is possible to confirm that the entity in question possesses the public key associated with the private key contained in their DID Document, it cannot be confirmed that they are truly the entity they claim to be. This is because anyone can create their own DID using the various available DID Methods.

In an SSI ecosystem, authentication occurs using the next layer of Verifiable Credentials, which has not been addressed in this project. Instead, the solution to this problem in this project is found in the SnD DID Method, specifically through the use of the PSK to encrypt DID Documents written on the Tangle, as mentioned in the section 3.1. This means that anyone can create their own DID on the Tangle using SnD. As example, it is considered a scenario where a group of IoT nodes share a common PSK. Only those who have that key will be able to create a DID on the Tangle valid within that group of nodes. If an attacker attempts to impersonate one of the nodes by conducting a man-in-the-middle attack, during the creation of a TLS channel between a client node and a server node, and sending its DID created using a different PSK, the attack will not be successful. The client node will detect during the "resolve" operation that the attacker's DID Document was not created with the correct PSK. Taking into account this information, it is possible to proceed with the analysis of two types of MITM attacks.

Transparent Proxy Attack

In this scenario, the attacker acts as a transparent proxy, forwarding messages between the client and server. The attacker has access to all the information exchanged between the two parties, but lacks the ephemeral private keys needed to calculate the session key and decrypt the data. As a result, once both parties have sent the ChangeCipherSpec message, the traffic between them will be encrypted, rendering the attacker unable to view the exchanged data.

To defend against a transparent proxy attack, it is important for both the client and server to verify the authenticity of the DID and integrity of the handshake messages. It is also important for the client to ensure that it is communicating with the intended server, and not a malicious actor posing as the server. This protection as mentioned above is ensured by the SnD DID Method.

Based on this analysis, it can be concluded that the changes made to the protocol did not introduce a vulnerability that would make this version of TLS susceptible to attacks of this type.

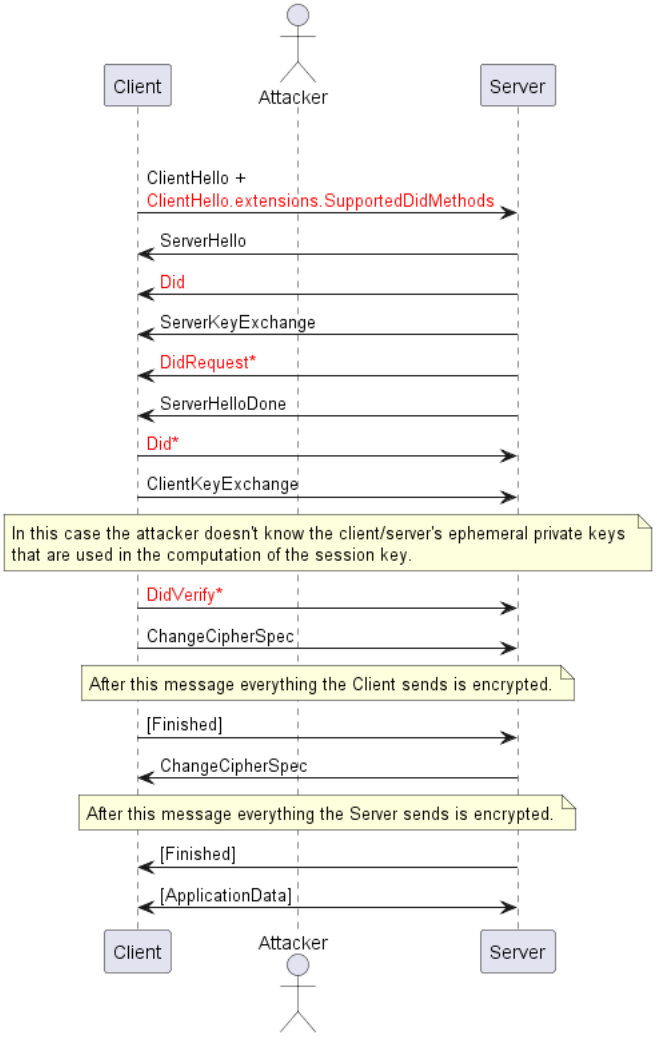

Figure 3.4. Transparent Proxy Attack

Active Proxy Attack

An active proxy attack is a type of man-in-the-middle (MITM) attack that involves the attacker inserting themselves into the communication between a client and a server, acting as a intermediary. The attacker actively modifies the messages exchanged between the client and server, as opposed to simply forwarding them like in a transparent proxy attack.

During the TLS handshake, the client and server exchange information such as their supported cipher suites and DIDs. The attacker can intercept and modify this information, potentially downgrading the security of the connection or injecting malicious content. The attacker can also choose to terminate the connection and create a new one between the client and server, using their own DID and keys. In this case, the client may not be aware that it is communicating with an attacker rather than the intended server. However, this is not possible since the protection offered by the SnD method prevents an attacker from obtaining a valid DID. Thus, the attacker would not be able to impersonate the server.

Nevertheless, the attacker may attempt to reuse the DID sent by the server and forge a new ServerKeyExchange message with its own ephemeral public key. However, this attack will still fail because the attacker must generate their own (EC)DHE parameters and include the public key in the ServerKeyExchange message with a valid signature. In order to do this, the attacker must possess the server's identity private key, which is associated with the DID that they forwarded. The client's verification process will fail withouth the server's private key.

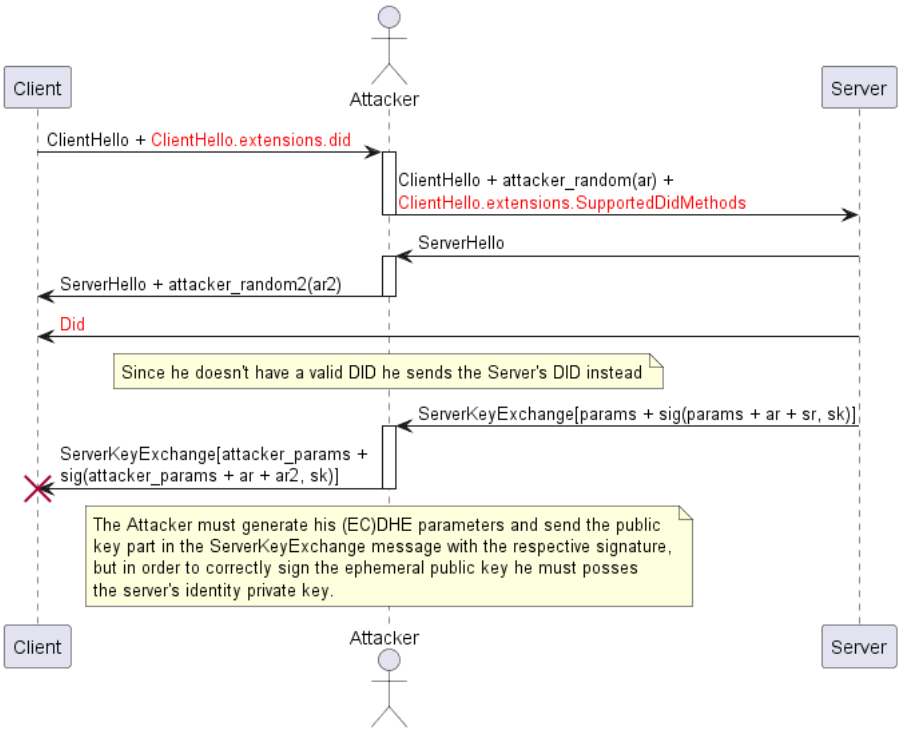

Figure 3.5. Active Proxy Attack

Chapter 4

Test and result analysis

In this chapter, the performance of the SSI-aware version of Mbed TLS, which uses Decentralized Identifiers for authentication, is compared to the standard version that uses X.509 Certificates. The tests has been carried out onto three different hardware platforms: an `Intel i7-6700HQ @ 2.6GHz (x86_x64)` processor [18], a `Raspberry Pi 4 Model B` [19], and a `STM32 Discovery board` equipped with an `STM32L4S5VIT6 MCU @ 120 MHz` [20]. It is worth noting that the DID Method integrated within Mbed TLS for the SSI-aware TLS handshake has been implemented on top of the WAM protocol for the x86_64 and Raspberry Pi 4 Model B, and on top of the L2Sec [21] protocol for the STM32L4S5VIT6. The L2Sec protocol was specifically developed and optimized by STMicroelectronics for their boards.

The handshake duration was recorded from the client perspective on all three platforms, with a total of 100 samples collected. The server was hosted on a `x86_64 system with an i7-12700H @ 2.30 GHz` processor [22] in each case. The tests were designed to measure the handshake time both with server authentication only and with mutual authentication. They were further divided into sub-tests based on the type of key used for identity (RSA or Elliptic Curve) on both the client and server sides, and the signature algorithm used (RSA2048-SHA256 or ECDSA-p256-SHA256).

In the scenario of certificate-based authentication, certificates with a "chain of trust" of lenght two were employed. The "chain of trust" is a series of digital certificates that verify the legitimacy of the certificate used for secure communication. This sequence starts with a trusted root Certification Authority (CA) and concludes with the certificate being considered. Regarding DID-based authentication, freshly created DID Documents were used. These documents have not received any update (see Sect. 3.1), resulting in a chain of DID Documents on the Tangle with a length of one. Throughout these tests, all operations related to DIDs were performed by interacting with the public and remote Tangle networks, specifically utilizing the IOTA Chrysalis DevNet.

4.1 TLS Handshake - Average Time Performances

The bar charts depicted in figures 4.1, 4.2, and 4.3 illustrate the average times calculated over 100 standard TLS Handshakes, which only include server authentication, across the three previously described platforms. The comparison is between the conventional X.509 certificate authentication, which is already specified in the TLS protocol by the RFC 5246, and the new SSI-aware version developed in this project, which employs Decentralized Identifiers as the authentication mechanism.

Observing the charts, it can be seen that on the left columns, the times of the two handshakes are represented where the server's identity key pair is RSA 2048 type and RSA2048-SHA256 is used as the digital signature algorithm, while on the right columns, the keys used are of the Elliptic Curve type derived from the NIST P-256 curve, and ECDSA-p256-SHA256 is used as the signature algorithm, both in the case of certificates and DIDs. The total time calculated for the SSI-aware handshake has been divided into two, in order to clearly highlight the average time taken for the DID Resolve compared to the rest of the handshake. The operations that have the greatest impact on the handshake time are certainly cryptographic operations, especially operations such as creating and verifying digital signatures. Comparing signature algorithms such as RSA2048-SHA256 and ECDSA-p246-SHA256, it can be noticed that the first algorithm is slower in generating the signature, but faster in verifying it. With that being said, in both types of handshakes, with certificates and with DIDs, these two operations can be found during some steps both on the client and server sides. On the server side, in the creation of the ServerKeyExchange message, where an ephemeral key exchange algorithm is used, the message will contain the generated public key together with the signature of it computed using the server's identity private key, while on the client side, in the verification of the aforementioned signature.

However, there is also a fundamental difference between the two types of handshakes regarding how they handle authentication. In the case with certificate-based authentication, the client performs "chain of trust" verification, which involves verifying the signatures on the certificates in the chain up to the root CA. This results in a higher computational load as the length of the chain increases, which does not occur with Decentralized Identifiers. From the charts, it can be seen that these operations have an impact on the average speed of the handshake, effectively making the SSI model faster than the standard model on all three architectures, this of course without considering the extra time given by the DID Resolve operation.

However, it must be considered that the handshake as a whole is the sum of all the various operations, including the "resolve". This step significantly affects the performance of the x86_64 and RaspberryPi 4 architectures, as having a lot of computational power, the time solely spent on the CPU-bound operations of the handshake is clearly lower than the time spent waiting for the DID Resolve to end, which is architecture-independent.

A remarkable result is obtained on STM32L4S5VIT6, which being an architecture with quite limited resources and also lacking a hardware accelerator, cryptographic operations have such a high impact that in this case the "resolve" time is almost negligible. In fact, in Figure 4.3 it can be seen that overall the SSI model

of TLS has better average performance compared to its counterpart.

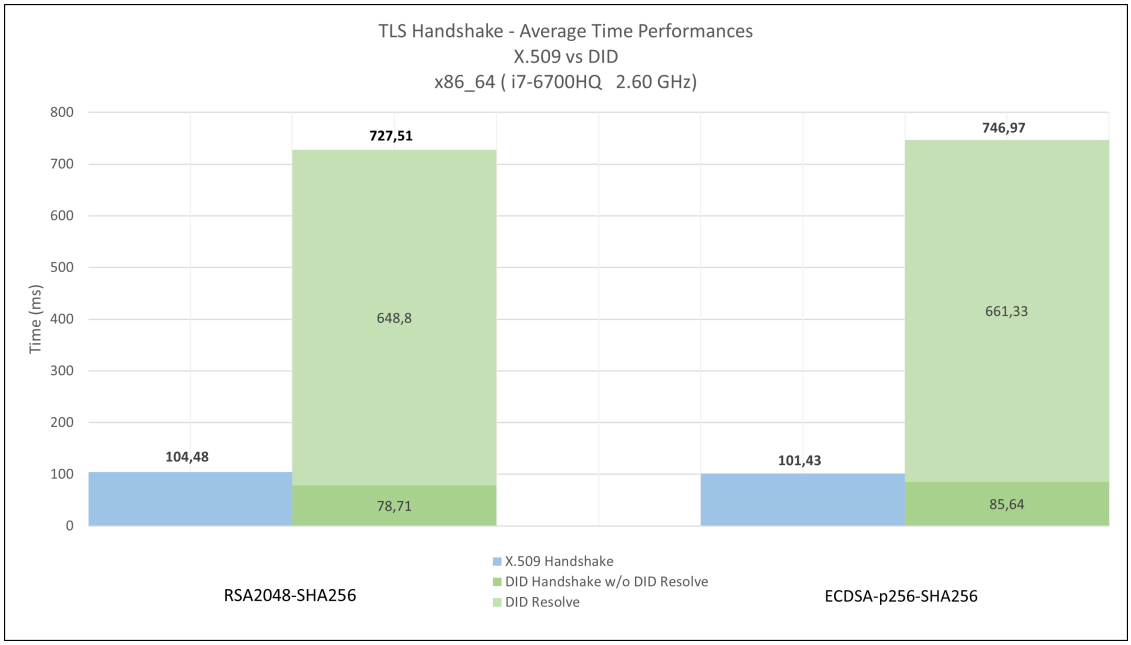

Figure 4.1. x86_64 - TLS Handshake - Average Time Performances

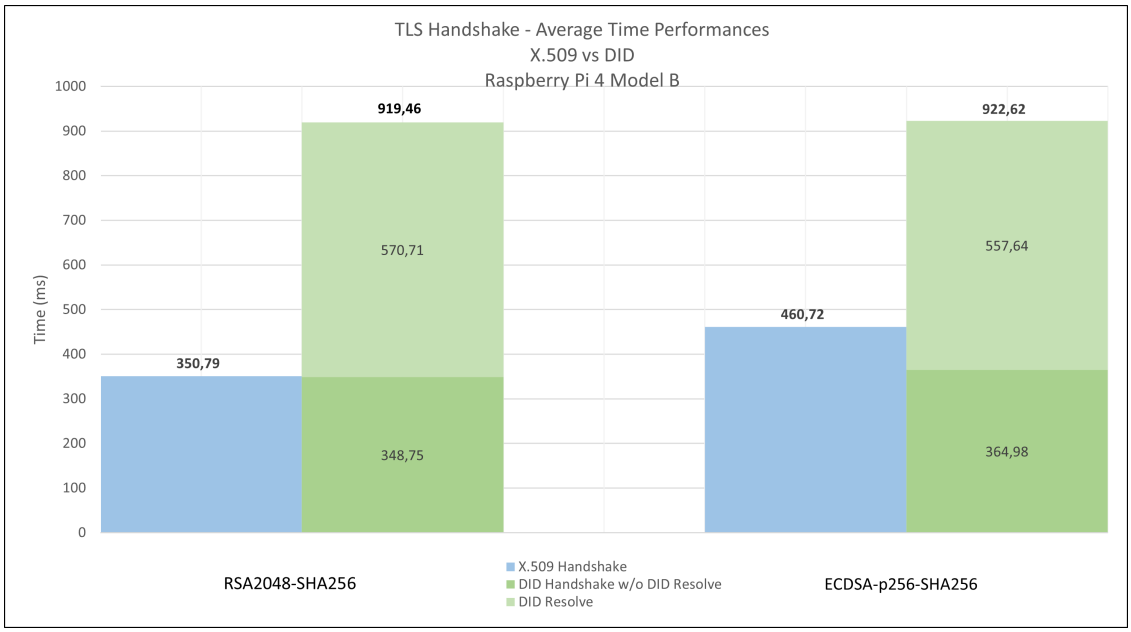

Figure 4.2. RaspberryPi 4 Model B - TLS Handshake - Average Time Performances

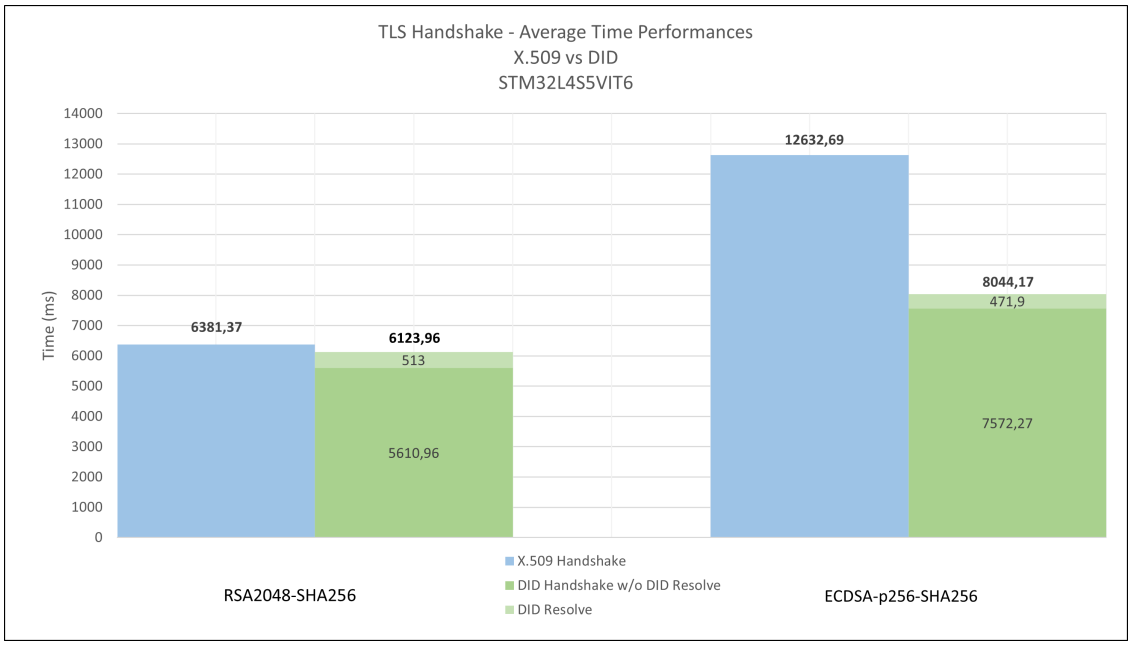

Figure 4.3. STM32L4S5VIT6 - TLS Handshake - Average Time Performances

4.2 Mutual TLS Handshake - Average Time Performances

The bar charts in Fig. 4.4, 4.5 and 4.6 represent the average time of a TLS handshake with mutual authentication, calculated for both models. As in the previous case, the performance was compared based on the type of key present in the DID Document or Certificate and therefore the signature algorithm used. Considering the two algorithms RSA2048-SHA256 and ECDSA-p256-SHA256, various combinations of them were tested on both the client and the server. This is because, in measuring the client-side handshake in a mutual authentication scenario, the client, in addition to verifying the signature of the parameters present in the ServerKeyExchange, as in the previous case, must also generate a signature to be sent in the CertificateVerify/DidVerify message. By testing the various combinations of the two algorithms on both the client and server sides, the time of the handshake can be measured as the signature algorithm applied to the previous two operations changes.

It is important to note that the results of tests conducted on the x86_64 architecture, as displayed in the chart, are impacted by the use of an Intel Core i7-6700HQ processor which has hardware acceleration for cryptographic operations. This processor is equipped with Intel's Advanced Encryption Standard New Instructions (AES-NI), which is a set of instructions that enhances the performance of cryptographic operations, including digital signatures.

These results are significantly different from those seen in other architectures where the combination of ECDSA on the client side and RSA on the server side tends to be the most efficient, while the reverse combination is typically slower. This is because the process of verifying a signature using the ECDSA-p256-SHA256

algorithm is generally slower than using RSA2048-SHA256, while the generation of a signature using ECDSA-p256-SHA256 is faster. However, this pattern may not apply to the x86_64 system due to the presence of hardware acceleration.

As an example, consider a server with an elliptic curve key in its certificate, and a client with an RSA key. In this scenario, the server would sign using the ECDSA-p256-SHA256 algorithm and the client would verify using the same algorithm. The client would then sign using RSA2048-SHA256, while the server would verify using RSA2048-SHA256. This scenario is expected to be the slowest, but on the x86_64 architecture, this may not be the case due to the hardware accelerator.

When comparing the two TLS models, also in this case a better result is obtained with the SSI-aware version when testing the protocol on an STM32L4S5VIT6 architecture, despite the presence of the additional latency caused this time by two DID Resolve operations, one on the client side and one on the server side. These graphs do not depict the time spent on "resolve" compared to the rest of the CPU-bound operations of the handshake, as it was not enough to just take the time of the two "resolves" and subtract them from the total. This is because while the server is busy performing its "resolve," the client is not waiting and continues its part of the handshake until reaching the Finished message, where it is actually waiting for the server's subsequent messages.

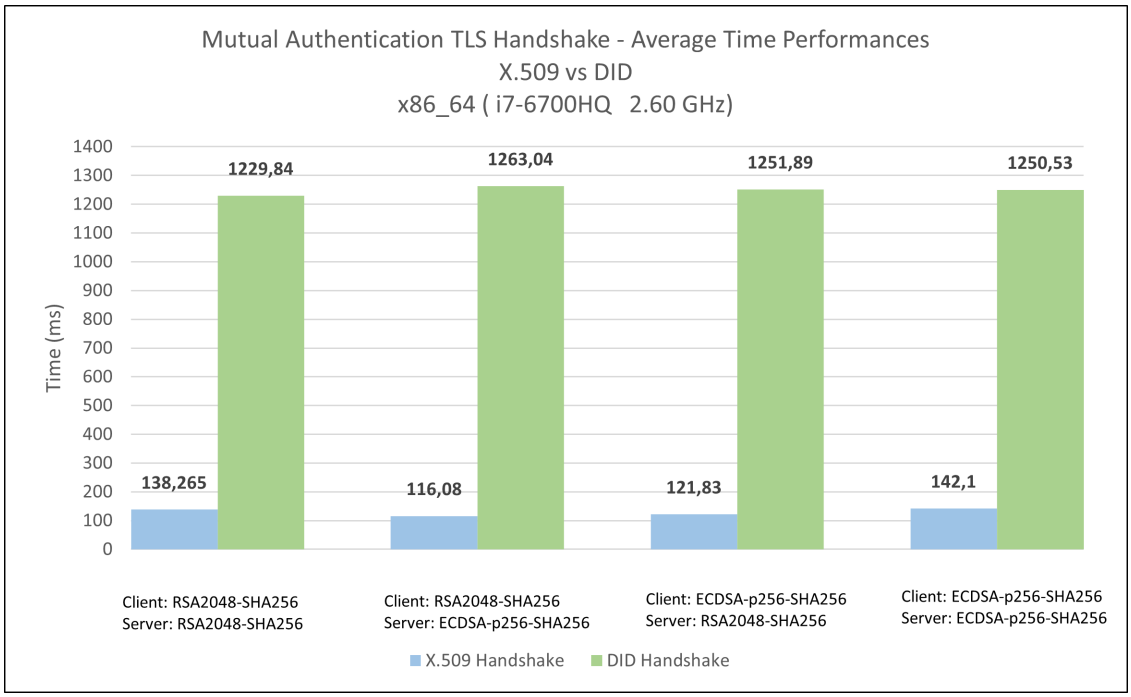

Figure 4.4. x86_64 - Mutual Authentication TLS Handshake - Average Time Performances

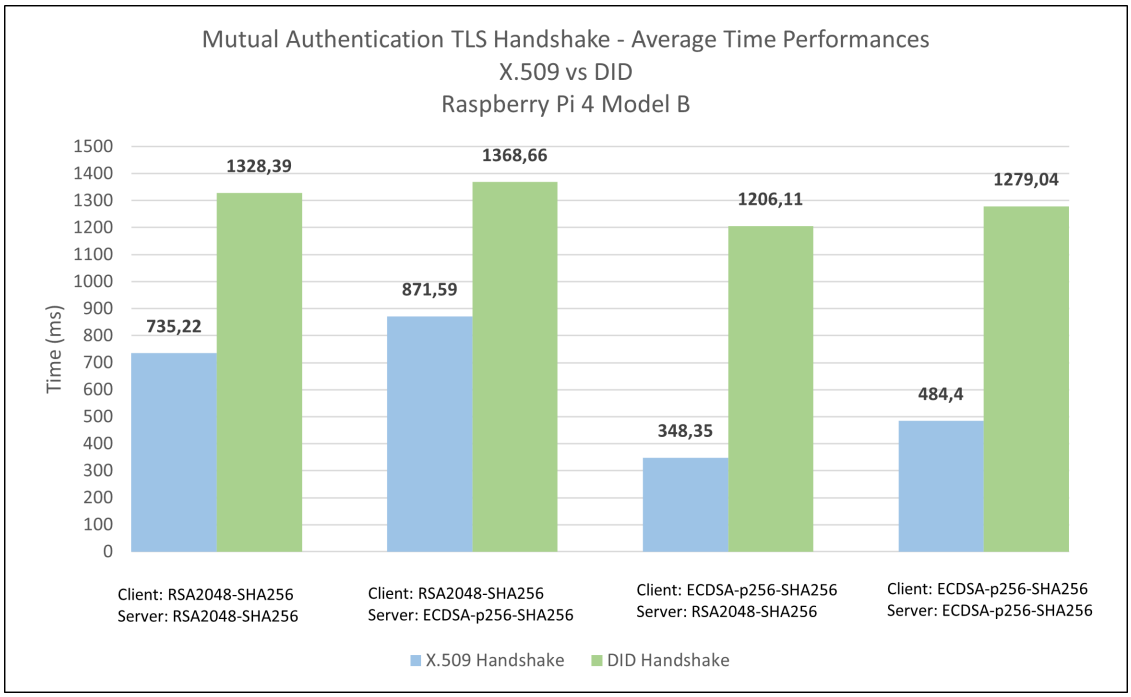

Figure 4.5. RaspberryPi 4 Model B - Mutual Authentication TLS Handshake - Average Time Performances

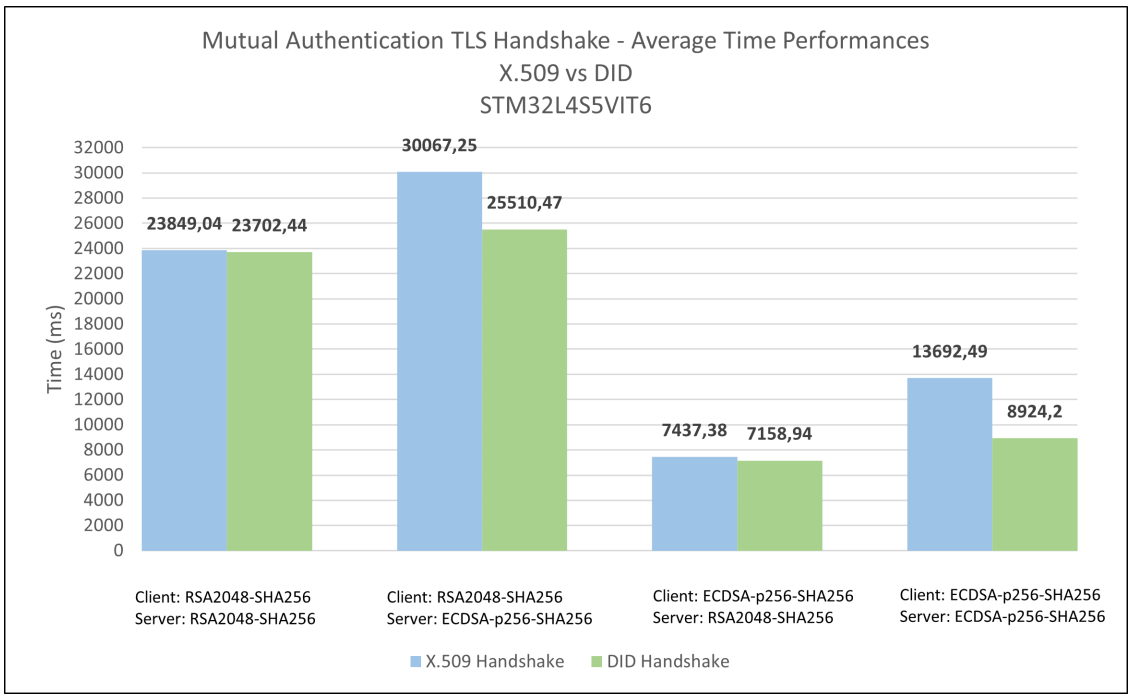

Figure 4.6. STM32L4S5VIT6 - Mutual Authentication TLS Handshake - Average Time Performances

4.3 DID Resolve (SnD) - Time Performances

The results of previous studies on the handshake with Decentralized Identifiers (DIDs) are highly variable due to the unpredictability and variability of the DID Resolve process. Several factors contribute to the resolve time (Fig. 4.7), including the internet network's capacity and speed, which can be improved by installing a private Tangle within a LAN network. The time and day of connection to the Tangle can also play a role, as there are peak times with more traffic and quieter times with fewer transactions. The type of IOTA network employed as DLT (such as the Mainnet, Shimmer, Chrysalis Devnet, etc.) can also impact results. Furthermore, the length of the DID Document chain, as each update of one's identity results in a new DID Document being written to the bottom of the chain, and during the DID Resolve, the entire chain must be traversed to reach the most recent DID Document. If the chain is very long, traversing through it can become time-consuming. Finally, the IOTA's Congestion Control Algorithm [23], which adjusts traffic based on load and request numbers, may also impact the resolve time. These factors contribute to the inconsistent results and make it difficult to accurately assess the performance of DID-based TLS handshakes.

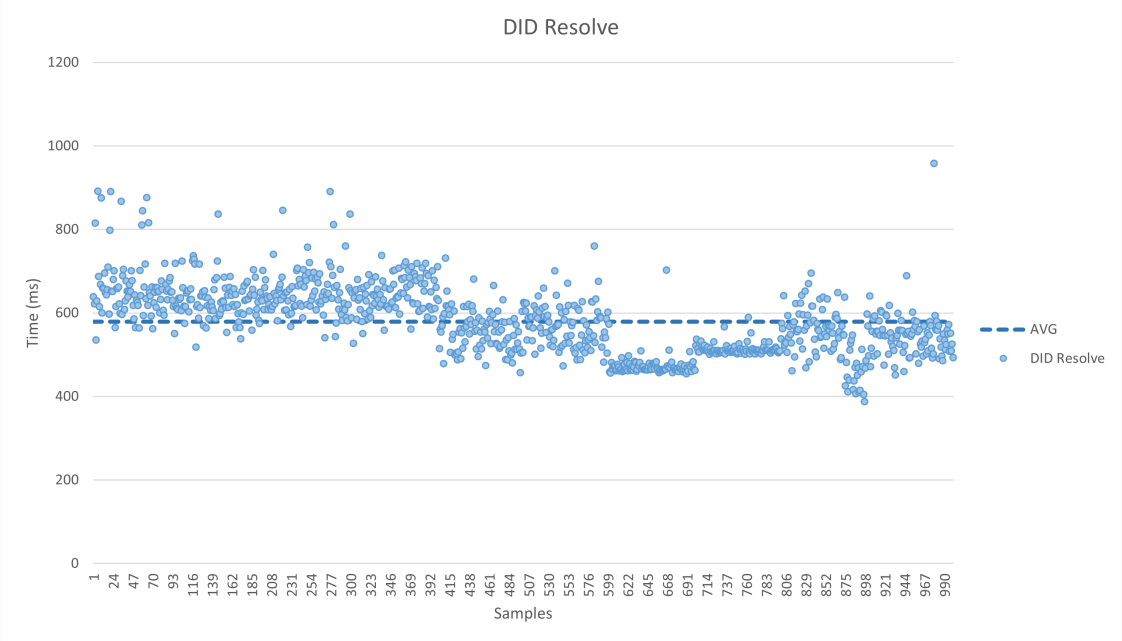

Figure 4.7. SnD - DID Resolve Time Performances

Chapter 5

Conclusion and future work

The main objective of this work was to integrate IoT-constrained devices into the SSI ecosystem by assigning them decentralized digital identities. With their decentralized digital identity, these IoT-constrained devices are capable of authenticating with each other through the use of the SSI-aware version of the TLS 1.2 protocol, which was implemented into the Mbed TLS library. The principal characteristic of this model is the use of Decentralized Identifiers as the authentication mechanism. This replaces the conventional X.509 Certificates, thereby abandoning a centralized approach in favor of a decentralized solution.

To enhance this solution even further, the possibility of integrating Verifiable Credentials can be explored. This would eliminate the need for an authentication mechanism internally implemented by the DID Method, as was the case with the Pre-Shared Key in SnD. Moreover, updating the SSI-aware model to include TLS 1.3 and integrating it into Mbed TLS is another potential goal for the future. This work serves as an introductory step towards adopting Self-Sovereign Identity as the new paradigm for digital identity and authentication, within a well-established protocol such as TLS.

Appendix A

User Manual

This appendix explains how to install and how to use the tools of this project.

A.1 DID Method: SnD

The DID Method used in this project for the creation, update, revoke and resolve of DIDs and DID Documents.

A.1.1 Requirements

The following dependencies are required for SnD to work properly.

cJSON

`cJSON` [24] is a lightweight, efficient, and flexible C library for parsing and generating JSON data. It is designed to be easy to use, with a simple and intuitive API, and it is designed to be easy to integrate into a wide variety of applications.

WAM

`WAM` is the cryptographic protocol for exchanging data with the IOTA Chrysalis Tangle explained at section 2.7.

libsodium

`libsodium` is a modern, easy-to-use crypto software library written in C programming language. It provides a high-level API for secure cryptographic operations and is designed to be fast and secure.

A.1.2 Installation

To install the SnD DID Method, one must enter the folder named `C_CRUD` within the project and run the `build.sh` script:

```
cd C_CRUD
bash build.sh
```

The `build.sh` script (Figure A.1) is designed to perform the compilation of the `iota.c` and `WAM` libraries, resulting in the creation of a `libdidmethod.a` static library and a `./demo/demo` application. The script begins by cloning the `dev` branch from the `iota.c` library repository and proceeding to compile it. Next, it clones the `WAM` library repository and, using the `cmake` tool, generates a Makefile that is used to output the final artifacts.

```
#!/bin/bash

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

Figure A.1. build.sh

A.1.3 Usage

Once the compilation has finished, the SnD demo application is ready for use. Upon running it, a screen similar to the one depicted below will appear on the terminal, prompting the user to select an action.

```
What do you want to do?
  1. Create a DID
  2. Update a DID
  3. Revoke a DID
  4. Resolve a DID
```

Here, you have four options:

1. you can create a new Decentralized Identifier. If this option is chosen, it will prompt the user to specify whether they wish to define an `authenticationMethod` and/or an `assertionMethod` and the path to the corresponding public keys to be included in the DID Document.
2. you can update your current DID. As for the "create" operation, the user will be prompted to define a new `assertionMethod` and/or `authenticationMethod`.
3. you can revoke your current DID
4. you can resolve any DID. At The user is asked to enter the DID to be "resolved".

A.2 SSI-aware Mbed TLS

A Self-Sovereign identity aware version of the TLS 1.2 protocol implemented in the Mbed TLS library version 2.28 [25].

A.2.1 Requirements

The following dependencies are required for SnD to work properly both for Linux and on the STM32 Discovery board [26].

SnD static library (libdidmethod.a)

It is built by following the procedure outlined above in the section A.1.2 for the SnD DID Method installation.

STM32CubeIDE

STM32CubeIDE is a comprehensive, multi-operating system development tool that

provides advanced C/C++ development capabilities, including peripheral configuration, code generation, code compilation, and debugging features for STM32 microcontrollers and microprocessors.

It can be installed direcly from the official site [27].

A.2.2 Installation

Desktop/RaspberryPi with Linux

The SSI-aware Mbeb TLS library can be easily installed on a desktop environment or a RaspberryPi running Linux by executing:

```
bash ssi_aware_build.sh </path/to/libdidmethod.a>
```

To execute the `ssi_aware_build.sh` script (Figure A.2), you need to pass the path to the directory that contains the `libdidmethod.a` static library as an argument.

```
#!/bin/bash
set -e
if [ -z "$1" ]
then
      exit -1
else
  if [[ $1 == */ ]]
  then
    var=${1%?}
  else
    var=$1
    fi
fi
export PATH_TO_DID_METHOD=$var
DIR=mbedtls_build
if [ -d "$DIR" ];
then
    echo "mbedtls_build already created"
else
  mkdir mbedtls_build
fi
cd mbedtls_build
cmake ..
cmake --build .
```

Figure A.2. ssi_aware_build.sh

STM32 board

To install the library on a STM32 board, a preconfigured project is available for download, which includes the library and a client application. The first thing to do is to open the project with STM32CubeIDE [27]. Once the project is open, it will need to be compiled. To do this with the mouse, click on the drop-down menu near the "Build" button, the one with the hammer symbol. From the menu, select the MBEDTLS-SSI option. Once the compilation is completed, connect the STM32 board to the PC and download the binary application into the flash of the microcontroller. The board will automatically reboot and communication can be established using a terminal emulator, such as Tera Term [28].

A.2.3 Usage

Desktop/RaspberryPi with Linux

Once the compilation is finished, in the `./mbedtls_build/programs/ssl` folder, there will be a client application (`ssl_client2`) and a server application (`ssl_server2`) with which to interact.

By running both applications and passing `-h` as a parameter, it will be possible to view the available options for both applications. If someone is interested, they can try out the various options as desired. However, only the main options for testing the library's functionality will be discussed here.

On the server side, we have these options:

- `./ssl_server2`

  Launches an instance of a server with a preloaded certificate and corresponding preloaded private key.

- `./ssl_server2 auth_mode=required`

  By doing this, it instructs the server to operate in mutual authentication mode, which always requires the client to provide authentication.

- `./ssl_server2 ca_file=<path/to/CA_certificate>`
  `crt_file=<path/to/own_certificate>`
  `key_file=<path/to/private_key>`
  `key_pwd=<priv_key_password>`

  Here, it is possible to configure the server with a top-level CA that it should trust by passing the path of the file containing the CA certificate in `ca_file`. Additionally, the server can be configured with a specific certificate by passing it through the `crt_file` parameter, then passing the private key through the `key_file` field and the corresponding password, if it exists, through the `key_pwd` field.

- `./ssl_server2 did=1`
  `did_value="did:example:1234567890abcdefg"`
  `did_key_file=<path/to/private_key>`

  Finally, in this way it is possible to configure the server to use DID-based authentication. By passing `did=1` as the first parameter, it enables the use of DIDs in the TLS handshake. The `did_value` parameter is used to pass the server's DID value, and the `did_key_file` parameter is used to pass the path of the private key associated with the public key contained in the DID Document.

On the client side, we have these other options:

- `./ssl_client2 server_addr=<server's IP address>`

  This command launches an instance of the client that connects to the server using traditional certificate-based authentication, whose IP address is passed through the `server_addr` parameter.

- `./ssl_client2 server_addr=<server's IP address> did=1`

  This enables the client to use DID-based authentication. As a result, the client will inform the server of its intention to continue the handshake using DIDs as the authentication method. If the server does not support them, it will send an abort signal to the client and the connection will be closed.

- `./ssl_client2 server_addr=<server's IP address> did=1`
  `did_value="did:example:1234567890abcdefg"`
  `did_key_file=<path/to/private_key>`

  Like the previous command, this enables the use of DIDs and also loads the client's identity within the application, as previously seen on the server side. This enables the client to authenticate itself when the server requests it.

STM32 board

In the case of using the application on an STM32 board, it is not possible to launch the application and pass values through command line. Therefore, when the board is accessed and remotely connected to using a terminal emulator such as Tera Term [28], a menu, as seen in Figure A.3, will appear on the screen, with a series of options for the user to choose from. In the STM32 project, only the client application's functionality was loaded, and the parameters that are usually passed through command line on a PC or Raspberry Pi were preconfigured within the code. Hence, if there is a need to modify the client's identity, such as its DID and private key, or the IP address of the server it needs to connect to, it would require manually changing the hardcoded values in the code.

```
1. Node info;
2. Get data message;
3. Send data message;
4. Send sensor message;
5. Send encrypted data;
6. Test functions;
7. SSI-aware MbedTLS - Option List;
8. SSI-aware MbedTLS - DID - server authentication;
9. SSI-aware MbedTLS - DID - mutual authentication;
10. SSI-aware MbedTLS - X.509 - server/mutual authentication;
11. SSI-aware MbedTLS - DID - session resumption;
12. SSI-aware MbedTLS - X.509 - session resumption;

0. Exit.
Choose one of the options:
```

Figure A.3. Client's menu on STM32 board

Appendix B

Developer Manual

B.1 SSI-aware Mbed TLS

In this appendix, the main functions developed during the creation of the SSI-aware version of the TLS 1.2 protocol in the Mbed TLS library are listed and analyzed. The programming style utilized in the Mbed TLS library was adopted as much as possible to promote organization and comprehensibility.

`static int ssl_write_supported_did_methods_ext(...)`
It fills the ClientHello data buffer with the data of the SupportedDidMethods extension. In this extension the client inserts the list of supported DID Methods.
Input:

- `mbedtls_ssl_context *ssl`, reference to `SSL/TLS` context structure
- `unsigned char *buf`, pointer to the current position in the buffer containing the information to be sent in the ClientHello
- `const unsigned char *end`, pointer to the end of the buffer
- `size_t *olen`, length of this extension

Output: Return `0` if successful, or a `MBEDTLS_ERR_XXX_XXX` error code

`static int ssl_parse_supported_did_methods_ext(...)`
This function parses the information in the SupportedDidMethods extension. If this extension is present, it indicates that the client wants to use DID-based authentication instead of certificate-based authentication. The server verifies if it has a valid DID created with a DID method that is compatible with one of the ones listed by the client.
Input:

- `mbedtls_ssl_context *ssl`, reference to `SSL/TLS` context structure
- `const unsigned char *buf`, this is the pointer to the current position in the ClientHello buffer, now pointing to the SupportedDidMethods extension data.
- `size_t len`, SupportedDidMethods extension length

Output: Return `0` if successful, or a `MBEDTLS_ERR_XXX_XXX` error code

`static int ssl_pick_did(...)`
This function enables the server to check if the chosen ciphersuite, among those offered by the client, requires the server to have a specific type of DID and private key.
Input:

- `mbedtls_ssl_context *ssl`, reference to `SSL/TLS` context structure
- `const mbedtls_ssl_ciphersuite_t * ciphersuite_info`, current ciphersuite information

Output: Return `0` if successful, `-1` otherwise

`static int ssl_write_did_request(...)`
This function serves to create and send the DidRequest message to request client authentication.
Input:

- `mbedtls_ssl_context *ssl`, reference to `SSL/TLS` context structure

Output: Return `0` if successful, or a `MBEDTLS_ERR_XXX_XXX` error code

`static int ssl_parse_did_request(...)`
This function parse the the DidRequest message to check if the Server has requested the Client's authentication. This function is based on the function already present in Mbed TLS for parsing the CertificateRequest message called `ssl_parse_certificate_request`. The latter was created so that the information in the CertificateRequest message, which is meant to assist the client in selecting a certificate to send to the server, is ignored and not processed on the client side. The Mbed TLS developers opted for a minimal parsing of the message to verify its basic validity, allowing the client to send its certificate, whatever it may be, and let the server decide if it is suitable or not. It has been decided to adopt the same approach in the case of DIDs.
Input:

- `mbedtls_ssl_context *ssl`, reference to `SSL/TLS` context structure

Output: Return `0` if successfull, or a `MBEDTLS_ERR_XXX_XXX` error code

`static int ssl_write_did_verify(...)`
This function serves to create and send the DidVerify message if client authentication has been requested and the client holds a valid DID.
Input:

- `mbedtls_ssl_context *ssl`, reference to `SSL/TLS` context structure

Output: Return `0` if successful, or a `MBEDTLS_ERR_XXX_XXX` error code

`int mbedtls_ssl_write_did(...)`
This function serves to create and send the Did message containing the client/server

DID.
Input:

- `mbedtls_ssl_context *ssl`, reference to `SSL/TLS` context structure

Output: Return `0` if successful, or a `MBEDTLS_ERR_XXX_XXX` error code

`static int ssl_check_peer_did_unchanged(...)`
This function ensures that the peer's DID has not changed during session renegotiation, avoiding a attack called the Triple Handshake Attack.
Input:

- `mbedtls_ssl_context *ssl`, reference to `SSL/TLS` context structure
- `unsigned char *did_buf`,
- `size_t did_buf_len`,
- `did_methods did_method`,

Output: Return `0` if successful, or any other value in case of failure

`int mbedtls_ssl_parse_did(...)`
This function acts as a wrapper for function `ssl_parse_did`, and additionally checks that it is correct to expect and parse the message "Did".
Input:

- `mbedtls_ssl_context *ssl`, reference to `SSL/TLS` context structure

Output: Return `0` if successful, or a `MBEDTLS_ERR_XXX_XXX` error code

`static int ssl_parse_did(...)`
This is the core function responsible for parsing the "Did" message, retrieving the DID value and its corresponding DID Method, and invoking the function called `resolve_did`.
Input:

- `mbedtls_ssl_context *ssl`, reference to `SSL/TLS` context structure

Output: Return `0` if successful, or a `MBEDTLS_ERR_XXX_XXX` error code

`int mbedtls_ssl_conf_own_did(...)`
This function sets the client/server's DID Document and private key within the SSL/TLS configuration data structure. Must be called during the configuration phase inside a client or server application, before the start of the actual handshake.
Input:

- `mbedtls_ssl_config *conf`, SSL/TLS configuration to be shared between mbedtls_ssl_context structures
- `mbedtls_ssl_did_document *own_did_doc`, structure containing the information about a DID Document
- `mbedtls_pk_context *pk_key`, private key paired with the public key contained in the DID Document

Output: Return `0` if successful, or a `MBEDTLS_ERR_XXX_XXX` error code

void mbedtls_ssl_conf_did(...)
This function enables the use of DIDs during the handshake process.
Input:

- `mbedtls_ssl_config *conf`, SSL/TLS configuration to be shared between mbedtls_ssl_context structures
- `char did_conf`, this parameter holds either the value `MBEDTLS_SSL_DID_ENABLED` or `MBEDTLS_SSL_DID_DISABLED`

B.1.1 Relevant files

Some relevant files present in this project are explained below. These files contain definitions for new data structures and functions used throughout the library.

include/mbedtls/did.h and library/did.c

These files define the data structure that holds the pair of DID Document and private key, which together represents the peer's identity, along with a set of functions used internally during the handshake or exposed to users for interacting with DIDs. These functions can be seen more like wrapper functions that utilize the underlying "bridge" functions, discussed below, that interact with different DID Methods.

```
typedef struct mbedtls_ssl_key_did_doc{
    struct mbedtls_ssl_did_document *did_doc; /** did document **/
    mbedtls_pk_context *key; /** private key **/
} mbedtls_ssl_key_did_doc;
```

int resolve_did(...)
This function is invoked during the handshake when a peer receives the DID message from the other peer. This triggers a chain of function calls until the "resolve" function of the corresponding DID Method is called. The function internally invoked by this function is the `resolve_did_core`.
Input:

- `mbedtls_ssl_context *ssl`, reference to SSL/TLS context structure
- `const unsigned char *buf`, this buffer contains the value of the other peer's DID
- `size_t len`, this is the length of the DID
- `did_methods did_method`, this identifies the DID Method needed to "resolve" the DID

Output: Return `0` if successful, or a `MBEDTLS_ERR_XXX_XXX` error code

`mbedtls_resolve_own_did_document(...)`
This function performs a similar role as the `resolve_did` function, but it is designed for external usage, unlike the latter which is solely for internal library use. It has been utilized in both the client (`ssl_client2`) and server (`ssl_server2`) applications to fetch the client/server identity and subsequently configure the application with that identity.
Input:

- `mbedtls_ssl_did_document *did_doc`, peer's DID Document structure
- `const char *did_value`, peer's DID value

Output: Return `0` if successful, or a `MBEDTLS_ERR_XXX_XXX` error code

`int resolve_did_core(...)`
This function simply acts as a wrapper for the `resolve_did_mapping` function with some additional check.
Input:

- `mbedtls_ssl_did_document *did_doc`, peer's DID Document structure
- `const unsigned char *did_value`, peer's DID value
- `size_t len`, DID length
- `did_methods method`, corresponding DID Method

Output: Return `0` if successful, or a `MBEDTLS_ERR_XXX_XXX` error code

include/mbedtls/did_bridge.h and library/did_bridge.c

These files define the data structure used internally by the library for managing DID Documents, called `mbedtls_ssl_did_document`, along with the functions that serve as an interface between Mbed TLS and DID Methods. Currently, the only supported method is SnD.

```
typedef struct mbedtls_ssl_did_document{
    unsigned char *did;
    size_t did_len;
    did_methods method;
    mbedtls_pk_context pk; /** public key **/
} mbedtls_ssl_did_document;
```

`int resolve_did_mapping(...)`
This function is used to call the correct "resolve" function based on the DID Method related to the peer's DID. The function also includes checks for the SOV and BRCT Methods, to show how future developments could be, although for the moment these two methods are not handled within the library.

```
int resolve_did_mapping(mbedtls_ssl_did_document *did_doc, const
   unsigned char *did_value, size_t len, did_methods method){
    int ret = -1;
    if(method == SnD){
        ret = resolve_snd(did_doc, did_value, len);
    }
    if(method == SOV){
        /** Not yet implemented **/
        ret = -1;
    }
    if(method == BTCR){
        /** Not yet implemented **/
        ret = -1;
    }
    /** More could be added in the future **/
    return ret;
}
```

`static int resolve_snd(...)`
This function acts as a wrapper for the `resolve_` function defined within the SnD Method. Internally, it allocates a data structure of type `did_document_`, which is the data structure to hold DID Document information defined in SnD, and passes it along with the DID to the "resolve" function `resolve_`, which is responsible for retrieving the related DID Document and populating the data structure with the information contained in it. Subsequently, the function `map_from_snd` is called, which is responsible for mapping the information contained in the data structure `did_document_` to the data structure `mbedtls_ssl_did_document`, defined in Mbed TLS.
Input:

- `mbedtls_ssl_did_document *did_doc`, peer's DID Document structure
- `const unsigned char *did_value`, peer's DID value
- `size_t len`, DID length

Output: Return `0` if successful, or a `MBEDTLS_ERR_XXX_XXX` error code

`static int map_from_snd(...)`
This function is used to map the `did_document_` data structure defined in SnD to the data structure specific to Mbed TLS called `mbedtls_ssl_did_document`. In the future, there may be other functions like this for various other DID Methods, which internally manage DID Documents with a data structure defined by them, but Mbed TLS will always interact with this common data structure defined internally.
Input:

- `mbedtls_ssl_did_document *mbedtls_did_doc`, DID Document data structure defined in Mbed TLS

- `did_document_ *did_doc`, DID Document data structure defined in the SnD Method

Output: Return `0` if successful, or a `MBEDTLS_ERR_XXX_XXX` error code

Bibliography

[1] "Security mechanisms and technologies for constrained iot devices", Internet of Things A to Z: Technologies and Applications (Q. F. Hassan, ed.), pp. 219–254, Wiley-IEEE Press, 2018, DOI 10.1002/9781119456735.ch8

[2] Mbed TLS, https://github.com/Mbed-TLS

[3] What is the Trust Triangle?, https://academy.affinidi.com/what-is-the-trust-triangle-9a9caf36b321

[4] Decentralized Identifiers (DIDs) v1.0, https://www.w3.org/TR/did-core

[5] What Is A DLT?, https://www.iota.org/foundation/our-research

[6] Verifiable Credentials Data Model v1.1, https://www.w3.org/TR/vc-data-model/

[7] IOTA foundation website, https://www.iota.org/

[8] The Coordinator, https://wiki.iota.org/learn/about-iota/coordinator

[9] IOTA 2.0 DEVNET, https://v2.iota.org/

[10] LINKS foundation website, https://linksfoundation.com/

[11] P. Rogaway, "Authenticated-encryption with associated-data", Proceedings of the 9th ACM Conference on Computer and Communications Security, New York, NY, USA, 2002, pp. 98–107, DOI 10.1145/586110.586125

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

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

[14] IANA, https://www.iana.org/

[15] IANA ExtensionType Registry, https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml

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

[17] STMicroelectronics, https://www.st.com/content/st_com/en.html

[18] Intel® Core™ i7-6700HQ Processor, https://www.intel.com/content/www/us/en/products/sku/88967/intel-core-i76700hq-processor-6m-cache-up-to-3-50-ghz/specifications.html

[19] Raspberry Pi 4 Model B, https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/

[20] STM32L4+ Series - STM32L4S5VIT6 MCU, https://www.st.com/en/microcontrollers-microprocessors/stm32l4-plus-series.html#overview

[21] X-CUBE-IOTA1 Firmware Package, https://github.com/STMicroelectronics/x-cube-iota1

[22] Intel® Core™ i7-12700H Processor, https://www.intel.com/content/www/us/en/products/sku/132228/intel-core-i712700h-processor-24m-cache-up-to-4-70-ghz/specifications.html

[23] Explaining the IOTA Congestion Control Algorithm, https://blog.iota.org/explaining-the-iota-congestion-control-algorithm/

[24] cJSON, https://github.com/DaveGamble/cJSON

[25] Mbed TLS 2.28 repository, https://github.com/Mbed-TLS/mbedtls/tree/v2.28.0

[26] STM32 discovery board, https://www.st.com/en/evaluation-tools/stm32-discovery-kits.html

[27] STM32CubeIDE, https://www.st.com/en/development-tools/stm32cubeide.html

[28] Tera Term, https://ttssh2.osdn.jp/index.html.en