POLITECNICO DI TORINO

MASTER's Degree in COMPUTER ENGINEERING

MASTER's Degree Thesis

Anchoring self-sovereign identity to a hardware TPM 2.0

Supervisors

Prof. DANILO BAZZANELLA

Ph.D ANDREA GUIDO ANTONIO VESCO

Candidate

MICHELE FESTA

APRIL 2025

Summary

Self-sovereign identity (SSI) is a decentralized identity model. When applied in the Internet of Things domain, this model enables digital trust independently of the protocol agreed among peers for data exchange. Each peer has full control on its own identity and it must publish a decentralized identifier on a trusted public utility. Such identifiers are the basic building block to provide authentication in a decentralized fashion. However, SSI framework requires secure management of the cryptographic material required for building the digital identity. Each device is the true owner of its own identity, therefore a proper key management system needs to be integrated for autonomously generating identity keys and produce digital signatures.

Today, trusted platform modules (TPMs) are used in IoT. TPMs have key management capabilities: they can generate asymmetric key pair in its isolated environment from the operating system and securely produce digital signatures. This thesis describes the usage of an hardware TPM 2.0 as a key management system for the IOTA Identity framework, an SSI library that relies on the IOTA Tangle, a largely used Distributed Ledger Technology (DLT). Both projects are created by IOTA Foundation. The *de facto* standard key management system implementation for IOTA Identity is Stronghold. This solution is a software component that can execute the required cryptographic procedures and implements a secure storage of the cryptographic material, both at rest and in memory. The IOTA Identity framework contains generic interfaces to ease the implementation of custom key management systems to support SSI operations. The aforementioned interfaces have been implemented to offload key management operations to the hardware TPM 2.0 device. Specifically, the TPM 2.0 creates keypair with signing capability and exposes the public part when a new decentralized identifier needs to be published. On the other hand, the sensitive part of the key pair is never exposed outside of the TPM 2.0 context. Similarly, if a verifier requires to verify the credential, the TPM 2.0 loads a previously generated signing key and produces a digital signature.

In addition, the usage of hardware TPM 2.0 can be propagated to the higher level of the SSI framework. A credential issuer that is also a Privacy Certification

Authority can verify the TPM 2.0 device identity and the policy of the key used for decentralized identity. The issuer releases credentials that enforce a strong policy about the usage of identity keys. A peer that requires a stronger authentication may require for such credentials and trust only digital signatures performed by keys loaded in the hardware TPM 2.0 that can neither be exported or generated externally.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 IoT digital identity

The *National Institute of Standards and Technology* (NIST) gives a definition of digital identity:

> "For these guidelines, digital identity is the unique representation of a subject engaged in an online transaction. A digital identity is always unique in the context of a digital service, but does not necessarily need to uniquely identify the subject in all contexts. In other words, accessing a digital service may not mean that the subject's real-life identity is known" [1]

The NIST definition of digital identity is a representation of an online persona. A digital identity does not need to completely describe a subject; rather, it is sufficient for it to possess certain attributes that serve to uniquely identify the subject within a specific context.

The subject of a digital identity does not necessarily refer to an individual; any type of abstract or physical entity that needs to be identified in a digital context requires an identity. IoT devices are a significant example of non-individual subjects associated with digital identities. Those devices are usually constantly available. Some devices are associated with sensors that collect data to transmit to a controller, while others receive actions that they execute with actuators.

The unique characteristics of IoT devices are also cause of security concerns. Data collected by these devices must be authentic, so that controllers can decide to take action on reliable data [2]. In addition, leakage of data collected from IoT devices compromises privacy. In conclusion, malfunctioning of IoT devices has the potential to result in physical harm [3].

Providing strong digital identities can be a solution to some of the security issues of IoT devices; subjects can be *authenticated* providing evidence of the digital

identity they claim to be. Different types of digital identity may be also *authorized* to perform specific operations rather than others.

Identity models can be categorized according to the actor that owns a digital identity. Three models can be distinguished: *centralized*, *federated* and *decentralized*.[4]

1.1.1 Centralized identity model

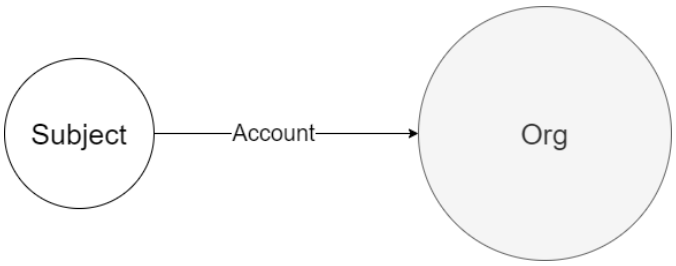

Figure 1.1: Centralized identity model

It represents a common *account-based identity*. Subjects can register to a certain organization, usually by providing a username and a password. The digital identity is owned by the organization rather than subjects. If the identity is deleted, the subject will be unable to access to the application.[4]

Centralized identity characteristics

- Centralized identities are not reusable nor portable. Subjects are required to create a new digital identity for each new registration.
- Identities are fully controlled by organizations. They control all the registered accounts of subjects, which generates a honeypot that serves to incentivize attackers. A successful attack may lead to a data breach, with a significant quantity of identities compromised.

1.1.2 Federated identity model

The *federated identity model* mitigates some of the issues of the centralized identity model. Subjects can now register to an *identity provider* (IdP). The digital identity is an account owned by the IdP. This digital identity can be used by subjects to access *relying parties* (RP) that support the identity provider where the account is registered. This mechanism is widely used and enables subjects to connect to different RPs, using the single sign-on (SSO) feature.

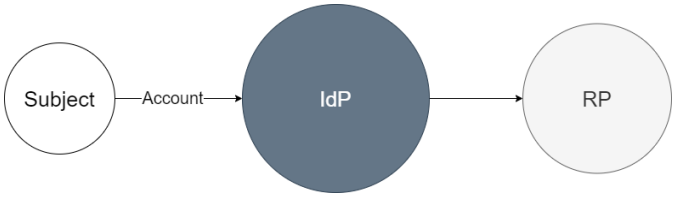

Figure 1.2: Federated identity model

Federated identity characteristics

- SSO partially allows to reuse digital identities
- It still does not exists a single IdP that enables access to any RP
- IdP is an even bigger centralized storage of accounts
- IdP can log subject activities online

1.1.3 Decentralized identity model

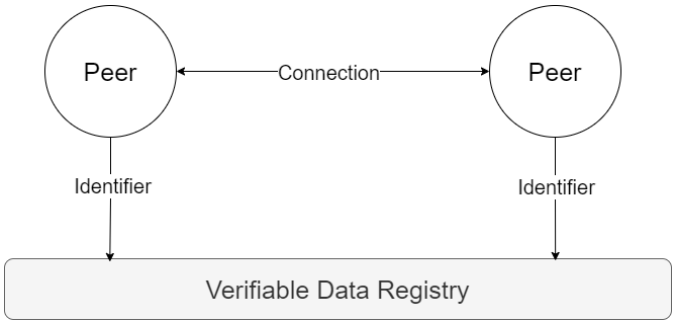

Figure 1.3: Decentralized identity model

The *decentralized identity model* proposes a different approach from centralization and federation. The main difference is that this model is no longer account-based. Instead, the parties involved contribute equally to share a peer-to-peer connection. Each subject publishes its own identifier on a *verifiable data registry* (VDR), visible to any other peer that wants to establish a connection. The peers can mutually authenticate using *asymmetric cryptography* [4].

Decentralized identity characteristic

- Subjects control their own decentralized identifiers

- Identifiers are published on a verifiable data registry
- Identifiers can always be reused
- Identifiers are published on a decentralized VDR.

1.1.4 Decentralized key management

A decentralized identity model solves weaknesses of the centralized and federated identity model. However, it also needs to overcome its own challenges. In fact, the biggest challenge of the decentralized identity model is *decentralized key management* [5].

It is recommended to delegate key management to hardware security devices [3]. In the context of *trusted computing*, Trusted Platform Modules 2.0 (TPMs) are the solution provided by the *Trusted Computing Group* (TCG) [6]. TPM devices are low power, relatively inexpensive but capable of storing keys and performing cryptographic operations in a shielded environment.

Furthermore, TPM 2.0 has its own identity based on a PKI. It uses a custom protocol that also extends the identity to transient keys generated by the TPM. This protocol enables to uniquely authenticate a TPM device while also being privacy preserving.

1.2 Thesis purpose

The purpose of this thesis is to analyze an existing decentralized identity model and to try to offload cryptographic capabilities to a secure hardware device. Specifically, the TPM is used as a key management system (KMS) for *IOTA Identity Framework* [7], a concrete example of decentralized identity.

Secondly, this thesis also proposes a solution to exploit TPM 2.0 capabilities to generate credentials that can be used in the decentralized identity model, ensuring that a decentralized digital identity is bound to a physical TPM 2.0 device.

Chapter 2

Background and related work

2.1 Self-sovereign identity

'*Self-sovereign identity*' has gradually become the most common term to identify the decentralized identity model. In fact, a self-sovereign identity is «neither dependent nor subjected to any other power of state» [4]. Indeed, the paradigm of self-sovereign identity represents a shift in identity management. As opposed to account-based models, each subject autonomously controls their own digital identities without the need to request to a centralized entity to operate on subject's behalf.

As illustrated in Figure 1.3, the self-sovereign identity model can be used to establish a peer-to-peer connection. Initially, each peer has to generate a new asymmetric key pair. The next step involves the publication of the public key on a trusted verifiable data registry, with a decentralized identifier (DID) being uniquely assigned to the published data [8]. Decentralized identifiers are described in detail in Section 2.2.1. Finally, peers can be mutually authenticated to establish a connection; each peer sends its own DID and the proof of possession of the private key. The proof can be verified by querying the VDR to retrieve the public key and verify the proof of possession.

2.1.1 ToIP Stack

The connection of peers using decentralized identifiers has already been described. However, the fact that a subject can autonomously manage its own identity does not imply that an SSI is *self-asserted* [4]. In particular, the *Trust over IP Foundation* project is interested in simplifying and standardizing how trust can be established

between digital identities [9].

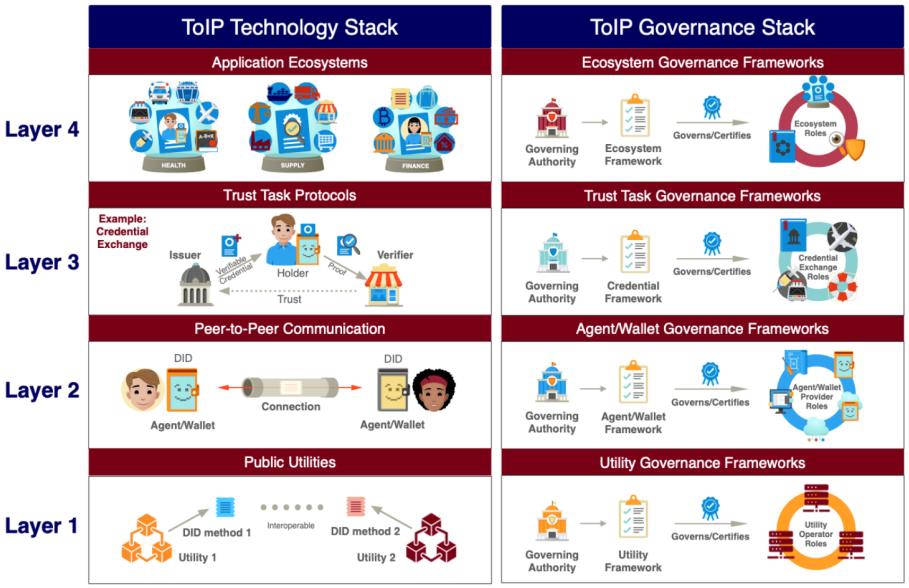

Figure 2.1: ToIP Stack [9]

The ToIP stack, shown in Figure 2.1, is a framework presented by the ToIP Foundation to provide the components required to obtain digital trust. Each level of the stack is separated in two halves: the first one delineates the technology, while the second one refers to governance frameworks applied for each level. Indeed, it is insufficient to ensure trust solely through the implementation of a technological stack; although digital trust is valid from a technological perspective, it is worthless in the real world if it isn't supported by a governance authority. Consequently, it is necessary for the governance authority to describe, at each layer, the rules that it considers necessary to respect in order to guarantee the validity of digital trust.

The layers defined in the ToIP stack are as follows:

- *Layer 1* is for verifiable data registries, defining the root of trusts for decentralized identifiers.
- *Layer 2* is for the establishment of peer-to-peer connection protocols.
- *Layer 3* is for the establishment of transitive trust relationship through the Trust Triangle.
- *Layer 4* is for application built on top of the decentralized digital trust infrastructure.

Trust Triangle

The credential Trust Triangle «enables establishment of transitive trust relationships between any three parties anywhere in the world using the data exchange formats and protocols for verifiable credentials» [9]. Transitive trust is supported by the utilization of digital credentials.

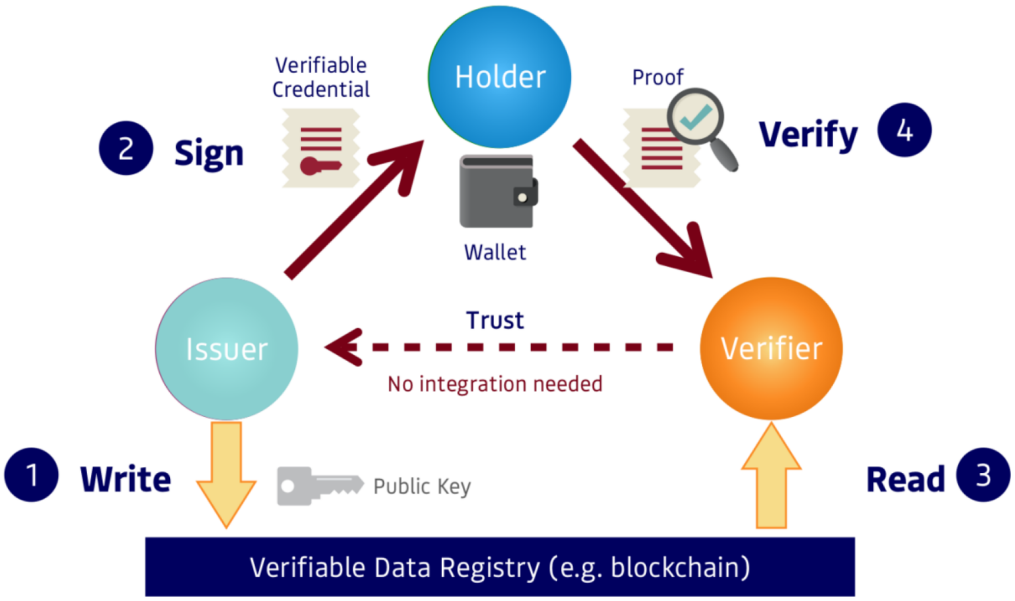

Figure 2.2: Verifiable Credential Trust Triangle [9]

Figure 2.2 describes how transitive trust can be established in the context of decentralized digital identity. The actors involved in the trust triangle are distinguished by their roles, which are categorized as follows.

- The *issuer* is the provider of digital credentials.
- The *holder* is the subject of a digital credential. It directly trusts the issuer and stores issued digital credentials into its own digital wallet.
- The *verifier* is any entity that requires some form of assurance about the holder. Next, the holder can present the credential provided by a trusted issuer.

The establishment of trust, as illustrated in Figure 2.2 is structured into four phases:

1. The issuer publishes a decentralized identifier on a trusted VDR

2. The issuer digitally signs a *Verifiable Credential* (described in Section 2.2.2) and send it to the credential holder.

3. The holder presents the credential to the verifier, which reads the DID of the issuer to recover its public key from the VDR

4. Finally, the verifier can verify the issuer's digital signature. If the verification is successful, it trusts the assertions that the issuer made about the holder

2.2 SSI building blocks

It has been previously described the self-sovereign identity model and trust relationships among SSI. This section focuses on the building blocks that are used to implement the aforementioned concepts. For this purpose, the following elements are used:

- An asymmetric key pair
- A *Decentralized Identifier* (DID) and its associated DID Document
- At least a *Verifiable Credential* (VC) provided by a trusted issuer
- A *Verifiable Presentation* (VP) provided to a verifier

2.2.1 Decentralized identifier

Decentralized identifier (DID) is a type of globally unique identifier that can be generated and autonomously controlled by subjects. Moreover, subjects can provide cryptographic proofs to demonstrate to control a DID. Decentralized identifiers are defined as a W3C Recommendation [8].

Figure 2.3: An example of DID [8]

Decentralized identifiers are represented as a URI (Figure 2.3). The URI scheme for a decentralized identifier corresponds to the value "*did*". In addition, the URI

also includes a DID method, which uniquely identifies a specification that describes how to create, read, update or delete DIDs. Often a DID method is associated to a specific verifiable data registry. Finally, the URI is completed with a *method-specific identifier*, which is a unique identifier that depends on the DID method.

The complete URI is a unique identifier that is resolved into a DID document.

DID Document

A DID document is a JSON object that contains data referred to a DID subject (e.g., public keys, trusted service endpoints).

DID Document


```
{
    "id": "did:iota:tst:0x44fb8cd620852460df1020f657842174e7a085d92c3fe3467ab4f1562468deda",
    "verificationMethod": [
        {
            "id": "did:iota:tst:0x44fb8cd620852460df1020f657842174e7a085d92c3fe3467ab4f1562468deda#AAuWHM9fcR6kgMlx_oMdc3Yot93CsZQyB-EfOGcy7Szh5Q",
            "controller": "did:iota:tst:0x44fb8cd620852460df1020f657842174e7a085d92c3fe3467ab4f1562468deda",
            "type": "JsonWebKey2020",
            "publicKeyJwk": {
                "kty": "EC",
                "alg": "ES256",
                "kid": "AAuWHM9fcR6kgMlx_oMdc3Yot93CsZQyB-EfOGcy7Szh5Q",
                "crv": "P-256",
                "x": "Z_iU9kPwIy4OPTj-PbhUck35pdUeu2S_xblcxgg9K2c",
                "y": "1gZ8z5COCxKTNqBwpa9ltzYerOAlLXsWmZRConEUD04"
            }
        }
    ]
}
```


Figure 2.4: An example of DID Document

At the very least, a DID document requires that the `id` property contains the DID of the subject. However, it would not be enough to provide proof of control of the DID in the SSI model. Therefore, a DID document should also have at least a verification method in the `verificationMethod` property. A verification method contains a representation of an asymmetric public key. A subject proves to control a DID providing a digital signature that can be verified with the public key defined in the verification method. Figure 2.4 illustrates a valid DID document that contains a verification method that wraps a public key represented as a JSON Web Key (JWK) [10].

2.2.2 Verifiable Credential Data Model

The Layer 3 of the ToIP stack (Section 2.1.1) implements transitive trust relationships on top of the decentralized identity model, through the management of digital credentials. Digital credentials are represented in compliance with the Verifiable Credential Data Model W3C Recommendation [11]. The specification provides a common format for representing any type of credential. Furthermore, it facilitates

the use of cryptographic proofs to detect and prevent tampering, as well as to verify the authorship of credential data.

Verifiable Credential

In accordance with the Verifiable Credential Data Model, digital credentials are represented by *Verifiable Credentials* (VCs). The information contained within these credentials is represented in JSON-LD format.

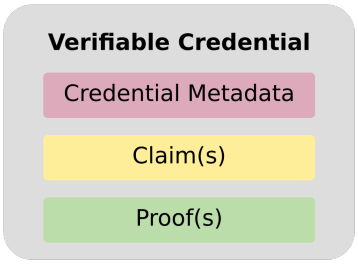

Figure 2.5: Verifiable Credential sections

As illustrated in Figure 2.5, a Verifiable Credentials contains three types of information:

- *Credential Metadata*: it describes properties about the credential (e.g., credential issuer identifier, expiration date, credential revocation mechanism).
- *Claims*: it represents the set of assertion of the issuer about the credential subject.
- *Proofs*: a credential contains one or more cryptographic proofs provided by the issuer.

Verifiable Presentation

The trust triangle is completed when the verifier verifies one or more credentials provided by the credential holder. The standard format provided by the Verifiable Credential Data Model to display VCs is the *Verifiable Presentation* (VP).

Analogously to the Verifiable Credential structure, a Verifiable Presentation provides three types of information:

- *Presentation Metadata*: it describes specific properties about the presentation (e.g., identifier of the holder, expiration date).
- *Verifiable Credentials*: a presentation contains one or more VCs that the holder wants to send to the verifier.

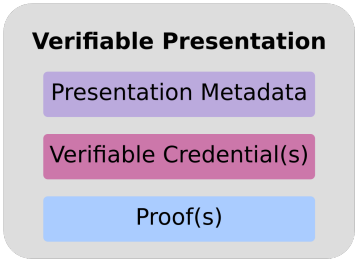

Figure 2.6: Verifiable Presentation sections

- *Proofs*: the holder can provide one or more cryptographic proofs.

When a verifier receives a VP, it verifies that it is an authentic message received from the holder. In addition, it can cryptographically verify that included Verifiable Credentials are authentic messages generated by trusted issuers. Furthermore, the verifier is able to validate the claims provided by the issuer and determine the trust to be placed in the holder.

2.3 IOTA Tangle

Distributed ledger technologies (DLTs) are commonly used as verifiable data registries in the SSI model [4, 9, 8]. In the Decentralized Identifiers specification, a definition of distributed ledger technology is provided as follows:

> 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.[8]

The IOTA Foundation has developed an entire ecosystem around its core, known as the IOTA Tangle. It is a DLT that supports a range of services for the development of decentralized applications for the IoT industry [12]. The current version of the Tangle, known as *Stardust*, enables the usage of the Tangle as a native token exchange, smart contract platform and verifiable data registry for SSI.

The Tangle structure is a directed acyclic graph (DAG), where transactions are represented as vertices in the graph. A new insertion approves other "parents" transactions. Consequently, the approval time of a transaction is determined by the traffic conditions on the DLT nodes. In conditions of low traffic, an inserted transaction will experience a longer waiting period before being approved from a new transaction. Under conditions of high traffic, it is easier to find a new transaction that verifies a previous one. On the other hand, new transactions also

need to provide a proof of work to protect nodes against distributed denial of service (DDoS) attacks [13].

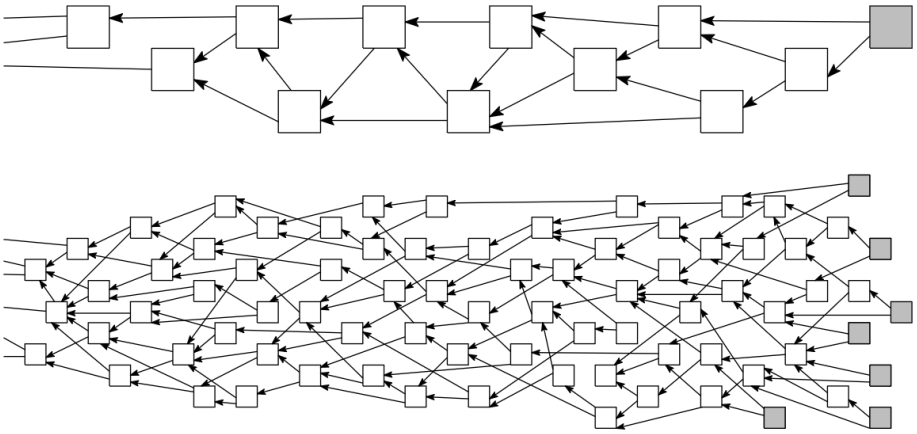

Figure 2.7: Example of the DAG generated by insertion of transaction in IOTA Tangle [12]

Transaction Block

The specifications of the Stardust protocol have been published as Tangle Improvement Proposals (TIPs) by IOTA Foundation. Specifically, TIP-0024 describes the structure of a transaction block. Each block has an identifier, named *block ID*, which is derived as the digest of the BLAKE2b-256 hash of the transaction block. The following fields, according to implementation proposals, are part of a transaction block [14]:

- *Protocol version*: version number that identifies the protocol.
- *Parents count*: number of directly approved blocks.
- *Parents*: a list of block identifiers that are directly approved by the block.
- *Payload length*: size of the transaction payload.
- *Payload*: generic content of the transaction.
- *Nonce*: the nonce found that completes the *proof of work*.

2.3.1 Consensus

Despite the goal of achieving true decentralized consensus [15], the Stardust version of the Tangle requires a node plugin, the *coordinator*, which is used by nodes to confirm blocks and reach consensus [16]. The current implementation of the coordinator is based on the *Tendermint Core BFT consensus* [17] which enable a set of validators to operate as a distributed coordinator. The coordinator scans blocks in the DAG for conflicts. Conflicting transactions cannot both be verified, as this could lead to a double spending attack [12].

The coordinator can insert in the ledger a single type of transaction, named *milestones*. The insertion of a milestone results in each block that is either directly or indirectly referred to the milestone adopting a definitive state. Legit transactions are included in the ledger state, while conflicts are excluded.

White-flag consensus

White flag consensus, as described in [18], defines the behavior of the coordinator when conflicting transactions are found. In TIP-0002, a deterministic order for transactions, starting from a milestone, is delineated. The order is generated using the Depth-First Search algorithm. In order to avoid conflicts, only the first valid message is added to the ledger state. The coordinator then sets the valid transactions as '*applied*' and adds them to the ledger state, while a conflicting transaction is marked as '*ignored*'.

Milestone

Subsequent to the process of transaction validation, the coordinator adds to the Tangle a specific transaction type, named *milestone*. Starting from the last inserted milestone, the coordinator executes the white-flag consensus on unconfirmed transactions. Conflicting transactions are excluded from the ledger state, marking them as ignored.

The coordinator also computes a *Merkle tree hash* (MTH) of transactions included in the validation, provided as a proof of inclusion. Indeed, the milestone payload includes the hash root of all blocks IDs referenced by the milestone and the hash root of all block IDs applied to the ledger state [19]. Finally, the coordinator digitally signs the content of the milestone, using the Ed25519 signature scheme. Tangle nodes that are in possession of the coordinator's public key in advance verify the authenticity of the milestone.

2.3.2 UTXO

Two distinct implementation proposals, respectively TIP-0018 and TIP-0020, define the content of the transactions supported in the Stardust version of the Tangle. The proposed model of transaction is based on *unspent transaction output* (UTXO).

In general, the account balances are not directly bound to an address (as in the account-based model). Instead, they are associated with the output of transactions. These outputs are then consumed as inputs of other transactions, and the unspent balance is stored in a new output. According to [20], the utilization of this model within the Tangle offers certain advantages over the prior account-based model:

- Parallel validation of transactions.
- Easier double-spending detection.
- Replay protection when using the same address.

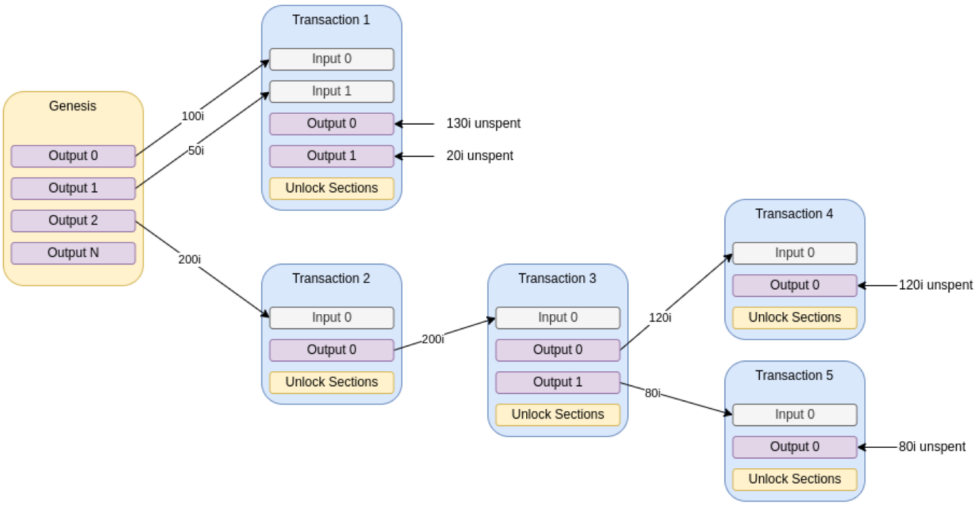

Figure 2.8: The UTXO model [20]

The figure 2.8 represent the high-level concept of UTXO, while TIP-0020 describes the structure of a transaction payload. The structure of a transaction payload is distinguished by three main components: *inputs*, *outputs* and *unlocks*.

Inputs

Inputs of a UTXO transaction are outputs resulted from previous confirmed transactions that are going to be consumed. Each transaction contains a counter of the inputs consumed, the list of inputs and the *input commitment*, which is the

concatenation of the hash of serialized outputs selected by the inputs. The list of inputs needs to refer to specific outputs:

- *Input Type*: always set to 0.
- *Transaction ID*: Block ID referring to the transaction that contains the output.
- *Transaction Output Index*: the index of the referenced output.

Outputs

The UTXO transaction produces a series of outputs. Therefore, the count of the outputs produced is added in the transaction together with the list of outputs. IOTA defines the Stardust version of the Tangle as a *multi asset ledger* [21]. The tangle supports different types of outputs that can be produced in UTXO transactions:

- Basic Output
- Alias Output
- Foundry Output
- NFT Output

2.3.3 Ledger programmability

The Stardust protocol exploits the functionalities of the UTXO model. In fact, this version of the protocol enables the execution of UTXO scripts on the ledger. Some UTXO output can be evaluated as states of a UTXO state machine. Stardust specifications provides a set of supported operations that can be used during the change of a state [21].

Chain constraint

The process of updating the state of a UTXO state machine is only possible according to a well-defined set of rules called chain constraint. It enables to «carry the UTXO state machine state encoded in outputs across transactions» [21]. The permitted operations for changing state depends on the output type and the current state of the transaction. Therefore, inside the transaction graph it is possible to identify *chains* generated by UTXO state machine. Each chain is uniquely identified by an identifier.

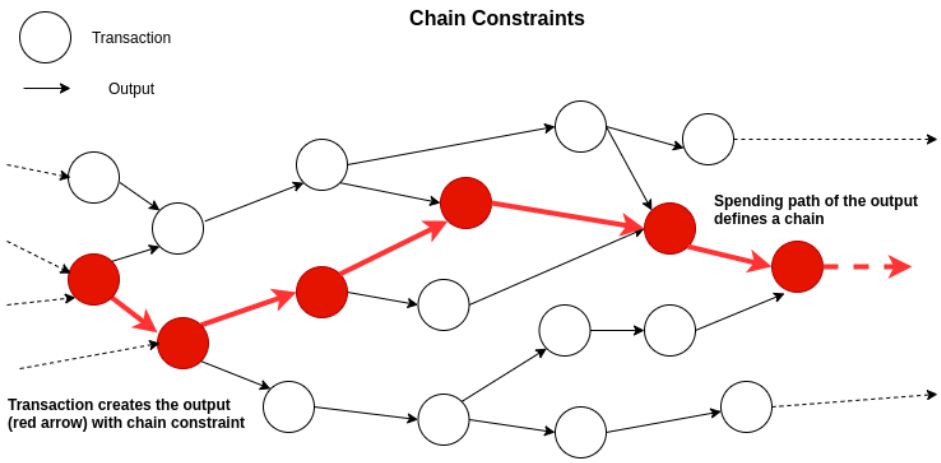

Figure 2.9: Representation of a chain generated by the UTXO state machine [21]

2.3.4 Alias Output

Alias output is a specific implementation of the UTXO state machine. Each state machine is uniquely identified by an alias identifier that remains unchanged for all the states of the UTXO state machine. The next clauses are describing some of the properties that characterize an Alias Output. The complete specification is available in TIP-0018 [21].

Alias ID

It is the unique identifier of the state machine. It is a 32 bytes long identifier generated as the BLAKE2b-256 hash of the specific output identifier that created the first state. The Alias Address is derived from the Alias ID.

State Index

It is a counter that is increased each time a new transaction is generated and the state machine moves to a newer output state.

State Metadata

This property is a byte array that can be updated exclusively by the state controller. A state controller is an address that is defined as the owner of an alias state machine. The byte array starts with a 16-bits integer that denotes the size of the array, succeeded by the raw bytes.

Unlock Conditions

UTXO outputs are characterized by specific unlock conditions. In order to consume outputs, transactions are required to unlock them. Each output type defined in Stardust has specific unlock conditions. In particular, an alias output can have at most one of each of the following:

- *State Controller Address Unlock Condition*: when an output is unlocked by the state controller, the state index is increased, and only some properties can be updated
- *Governor Controller Address Unlock Condition*: when an output is unlocked by the governor, the state index is not increased. The governor can update features, metadata, and unlock conditions.

Both unlock types are simply a digital signature of the hash of the transaction payload is executed, using the key associated with the correspondent address set in the unlock conditions.

Features

The characterization of outputs is also performed through features. While unlock constraint are adding constraints to be checked in order to consume a block, features enable new functionalities of outputs. For instance, an alias output can have at most one of each of the following features:

- *Sender Feature*: certain types of UTXO outputs, such as the alias output, necessitate a form of authorization, associating each output to exactly one specific address. Such address is specified in the sender feature property of an Alias Output.
- *Issuer Feature*: it is an immutable feature for the Alias Output. It contains the address of the creator of the UTXO state machine, named the issuer. It cannot be changed during state changes and it is validated to have always the same value in the UTXO chain.
- *Metadata Feature*: this features does not influence any validation of the output. It is an additional field where arbitrary binary data can be added. It is used to serve higher layer applications built on the Tangle. It starts with 2 byte representing the length, succeeded by a byte array containing the data.

2.4 IOTA Identity Framework

The preceding section provided a concise synopsis of main feature of the Stardust protocol version of the IOTA Tangle as a multi-asset ledger. The *IOTA Identity*

Framework [7], also known as IOTA Identity, is a library written in Rust that facilitates the utilization of the Tangle as a VDR and provides support for self-sovereign identity operations. The library is also compliant with the W3C recommendations for decentralized identifiers and Verifiable Credential Data Model [8, 11]. The operations available on IOTA Identity can be categorized as follows:

- DID operations: provides CRUD operations of DID documents, interacting with the IOTA Tangle.
- VC operations: provides a set of operation related to Verifiable Credentials creation, revocation and validation.
- VP operation: provides a set of operation related to Verifiable Presentations creation and validation.
- Key management: provides generic interfaces that can be implemented for binding a key management system as a provider of cryptographic operations for SSI. IOTA Identity also includes default implementations of such interfaces.

2.4.1 DID operations

IOTA Identity provides utility functions in order to use the Tangle as a verifiable data registry.

Publish

The publishing operation of a DID document consists in the creation of a new UTXO state machine of alias outputs.

As a requirement, the DID controller possesses a funded Ed25519 address. Indeed, an alias output that stores some data requires a deposit of IOTA coins, known as storage deposit [22]. This mechanism limits the maximum size of data that can be stored in the Tangle, avoiding an uncontrolled expansion of snapshot size on the DLT nodes. The following steps are to be taken for the publication:

1. Create a DID document compliant with the W3C Recommendation [8]
2. Create a new alias output. The state controller and the governor are set with the same funded Ed25519 address. The DID document is encoded in the *state metadata* property of the alias output.
3. Publish the new output as a transaction on the DLT.
4. When the block is published, the DID can be assigned to the output. In fact, the method-specific identifier is set to the hexadecimal encoding of the Alias ID.

It is important to note that the security of DID document data is also dependent on the management of the key pair associated with the Ed25519 address of the DID controller. In the event of a leakage of the sensitive part of the key, a malicious actor may gain control of the published DID document. For instance, the malicious actor could add a new verification method that contains a new key, allowing them to prove possession of the DID.

Resolve

IOTA Identity provides code examples for resolving DIDs [7]. Initially, the user provides a configuration of the Tangle node with which it whishes to interact. Subsequently, a resolver recovers the last alias output of the UTXO state machine associated to the alias ID contained in the DID. The DID document can be read from the *state metadata* property of the retrieved alias output.

Update

Updating a DID document consists in generating a new state of the UTXO state machine which contains the updated DID document in the *state metadata* property. The user should execute the following instructions, using IOTA Identity:

1. Resolve the DID in order to receive the correspondent DID document.
2. Update the DID document.
3. Encode the DID document in a new alias output.
4. Publish the output with a new transaction.

If the size of the DID document is increased, then it is also required to add more funds as storage deposit.

Delete

Delete operation of a DID document from the Tangle can be both reversible and an irreversible:

- *deactivation*: the state controller sets DID document metadata as '*deactivated*'. This change is an update of the state metadata, therefore it can be reversed. The DID documents can still be resolved but validation may fail.
- *destroy*: the governor address consume the alias output without generating a new one. The storage deposit is once again available to the governor for spending.

2.4.2 Verifiable Credential Data Model implementation

IOTA Identity Framework is compliant with the *Verifiable Credential Data Model v1.1* [11]. The library can only manage the JWT representation of Verifiable Credentials and Verifiable Presentation, described in Section 6.3.1 of the specification [11]. Additional encoding and decoding rules are provided in order to avoid data duplication and compatibility with JWT decoders.


```
{
    "iss": "did:iota:tst:0x6a226dac5200cc71f04f8b791899b8bb8e4bdcf03c59ae56e03299cd68daa1cb",
    "nbf": 1741987605,
    "jti": "https://example.edu/credentials/3732",
    "sub": "did:iota:tst:0x2a46fabfff11b3e2d4d1dd2e7e5538546e01b78d05e5aff5f226ef11b17d8b27",
    "vc": {
        "@context": "https://www.w3.org/2018/credentials/v1",
        "type": [
            "VerifiableCredential",
            "UniversityDegreeCredential"
        ],
        "credentialSubject": {
            "GPA": "4.0",
            "degree": {
                "type": "BachelorDegree",
                "name": "Bachelor of Science and Arts"
            },
            "name": "Alice"
        }
    }
}
```


Figure 2.10: Verifiable credential enveloped in a decoded JWT payload

As can be verified in figure 2.10:

- The `proof` field is absent and replaced by the JWT header and the JWT signature.
- The `iss` property corresponds to the issuer identifier.
- The `nbf` represents issuance date of the credential as a UNIX timestamp, in place of the `issuanceDate` property.
- The `jti` property replaces the `id` property.
- The `sub` corresponds to the identifier of the credential subject.
- The `exp` replaces the `expirationDate` property, formatted as a UNIX timestamp.
- The `vc` property (or `vp` for a Verifiable Presentation) contains the additional credential properties that are not replaced in JWT representations.

2.4.3 Key Storage

The cryptographic operations in the SSI model are based on asymmetric key pairs; each public key added into a DID document has a corresponding private key. DID subjects need to manage the key pair and be able to provide a proof of possession of the private key. IOTA Identity Framework provides two Rust traits, `JwkStorage` and `KeyIdStorage`, that can be used to connect an external key provider to the SSI operations implemented in the library.

Architecture

The Figure 2.11 illustrates the high-level architecture of the key storage interfaces. Specifically, the relationships between the interfaces have been highlighted in order to describe the retrieval of a private key, providing a verification method as input.

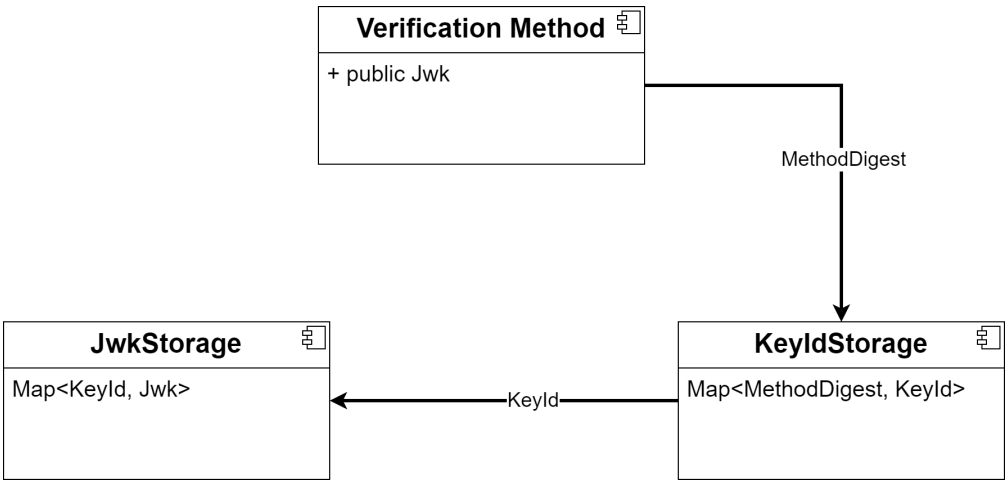

Figure 2.11: High level architecture of the Key Storage of IOTA Identity Framework

The following process is employed by IOTA Identity to address private keys, starting with a verification method:

1. Compute the method digest; for JWKs corresponds to the SHA256 thumbprint [23] of the public JWK.
2. Query the `KeyIdStorage` to retrieve, if exists, the key identifier associated to the verification method
3. Use the private key managed by `JwkStorage`, providing the key identifier as an input. The interface can retrieve the private key and complete the operation requested.

JwkStorage interface

The interface is designed to expose all the operations necessary to support the life cycle of key pairs in the SSI model. The keys are represented as JSON Web Keys. Each key pair is associated with a `keyID`, a generic identifier that is used to locally address the private key. In order to implement the `JWkStorage` interface, the following operations must be implemented:

- Generate a new key pair and return the public JWK, associated with a unique key identifier.
- Insert an existing JWK.
- Digitally sign messages with a supported signature scheme.
- Delete the JWK from the storage.
- Check if the JWK associated to a key identifier exists.

KeyIdStorage interface

The `KeyIdStorage` interface provides a mapping of a verification method to a KeyID. It allows to translate a public key representation (e.g., the digest of the public key) to the local identifier of the correspondent private part available in `JwkStorage`.

The implementation of the `KeyStorage` interface requires four operations:

- Insert a new couple of key identifier and verification method reference.
- Read the correspondent key identifier of the provided verification method.
- Delete a couple of key identifier and verification method reference.

Default implementations

IOTA Identity also provides default implementations of key storage interfaces. A first one, called `Memstore`, is an insecure in-memory implementation of key storage. Its intended use is exclusively for testing.

The default implementation intended for real use cases is the *StrongholdStorage*. The implementation of key storage interfaces are build upon *Stronghold*, the *de facto* standard key management system in IOTA Identity.

2.5 Stronghold

Stronghold is an open-source project, written in Rust, provided by the IOTA Foundation [24]. The core idea of Stronghold is to provide secret manager functionalities without exposing secrets outside of a protected environment generated by the software. In order to utilize sensitive data, Stronghold exposes special functions, termed 'procedures', which facilitate the utilization of data without disclosing the sensitive cryptographic material. The software is frequently employed as a library, yet it is equally capable of functioning as a command-line interface.

Stronghold is generally applied to three different use cases:

- It is used as a `SecretManager` in the IOTA SDK, supporting capability of a deterministic wallet [25].
- As previously mentioned, it supports the implementation of key storage interfaces in the IOTA Identity Framework [26].
- It can be used as a vault to protect arbitrary data provided by the user.

2.5.1 Data protection at rest

Stronghold securely stores secrets in files, using its own serialization structure, named *snapshot*. A snapshot contains all the information required by Stronghold when it is running, but they are protected using symmetric encryption. The encryption key is derived from a passphrase provided by the user. Therefore, Stronghold requests to provide a passphrase for unlocking the snapshot.

Snapshot encryption key generation

Each time Stronghold generate a snapshot, a new encryption key is generated as follows:

1. A secret key is derived from the passphrase

$$BLAKE256_2b(\text{passphrase})$$

2. Generate an ephemeral key pair for x25519 key exchange
3. Complete the key exchange among the secret key and the ephemeral key

Finally, the obtained shared secret can be used to encrypt the Stronghold instance with *xChaCha20-Poly1305* algorithm.

2.5.2 Data protection at runtime

When running, Stronghold is a collection of vaults. Each vault contains a set of sensitive data, named records. The content of records cannot be accessed directly; rather, they're loaded as crypto boxes. Crypto boxes are enclosures that protect records during runtime. Each vault has a master key that is used to retrieve plain text of records from the crypto boxes. The protection of data in crypto boxes is achieved using the *xChaCha20-Poly1305* algorithm.

However, this mechanism is not enough to protect sensitive data during runtime; vault keys are still available in plain text in the volatile memory. In the event of a memory dump, it would still be possible to use master keys to decipher vault records. In order to solve this issue, Stronghold implements a mechanism of mitigation based on *non-contiguous data types* [24].

This proposed mitigation mechanism involves the division of information (in this case, vaults' master keys) into two distinct shards of data. Consequently, the information is stored in two different shards and in different memory locations. In circumstances where a vault key is required to be used, shards are reunited to reconstruct the information.

In Stronghold, there are three possible types of non-contiguous memory implementation,categorized according to the location of the shards:

- *Full RAM*: both shards are stored in different pages of volatile memory.
- *Full Files* : both shards are stored on different files in non-volatile memory.
- *Ram and File*: it is a hybrid solution that stores the information both in file and in volatile memory.

The splitting phase of information is inspired by the work of Bruce Shneier *et al.* in "Cryptography Engineering" with the *Boojum scheme* [27]. Considering a random R and a key k to protect, we can define:

$$x = R$$

$$y = h(R) \oplus k$$

where h is specifically the $BLAKE256_{2b}$ hash function. With full control of memory, k can be deleted, leaving only x and y when data is not actually in use.

In the event of a memory dump being conducted at a specific moment in time, it is unlikely that the master key will be available in contiguous pages of memory.

It is also critical that the two shards are constantly refreshed. Considering a

new random R', then:

$$x' = x \oplus R' \tag{2.1}$$
$$y' = y \oplus h(x) \oplus h(x') \tag{2.2}$$
$$= k \oplus h(x) \oplus h(x) \oplus h(x') \tag{2.3}$$
$$= k \oplus h(x') \tag{2.4}$$

Therefore, it is always true that:

$$y \oplus h(x) = k \oplus h(x) \oplus h(x) = k \tag{2.5}$$

2.5.3 Procedures

Stronghold exposes a set of functions, named *procedures*, that can be used to interact with Stronghold environment. For instance, procedures are the only possible instrument that a user can use for operating with a key stored in Stronghold.

Procedures pipeline

It is possible to categorise stronghold procedures into three distinct types:

1. *Generators*: procedures that generates new secrets without receiving any input (e.g, BIP-39 mnemonic)

2. *Processors*: procedures that generates new secrets, receiving secret seeds as input. This is common for deterministic wallets (e.g, SLIP-0010 key derivation)

3. *Receivers*: procedures that produce a result that can be exported outside the Stronghold environment (e.g, public key derivation or digital signature)

The different types of procedures can be combine to produce a result that can be exported outside of the Stronghold environment. In figure 2.12 it can be observed a pipeline pattern generated by the categories of procedures. Initially, a generator creates a new secret and stores it inside a vault. The secret itself is stored in a crypto-box and can be addressed by an internal addressing mechanism called *locations*. Each new record in a vault has a location. When the generator procedure is completed, the location of the result is returned, instead of the secret itself. Next, if needed, a processor procedure can be executed, providing as input the location of the secret generated by the generator. The processor procedure can access the vault and operate with secrets, generating a new one in a new location. Finally, a receiver procedure receives as an input the location of a secret and return public data to the caller.

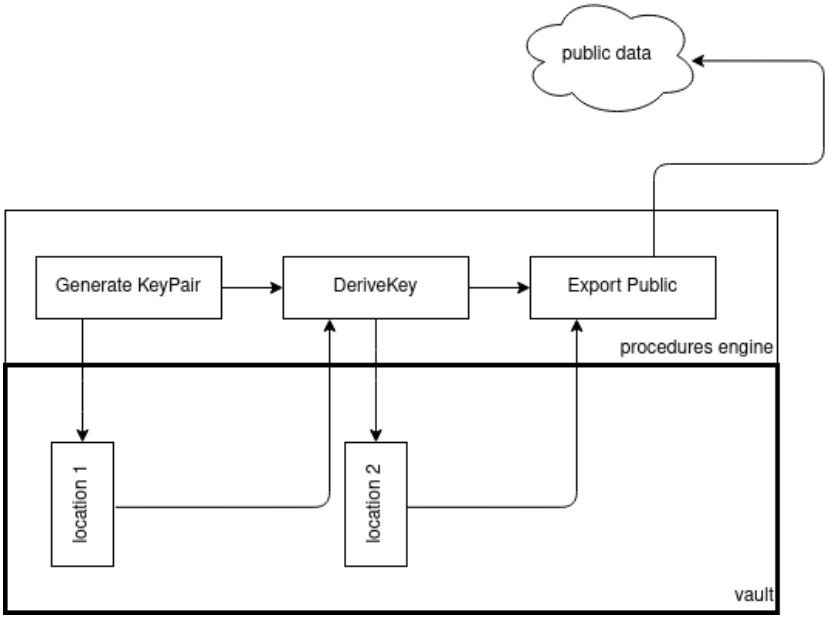

Figure 2.12: Stronghold procedure pipeline design [24]

2.5.4 Supported algorithms

Having considered how Stronghold works and the interfaces that interact with users, it is essential to analyze the list of available procedures to define the capabilities of Stronghold.

Stronghold procedures

The following list details the Stronghold procedures that are currently available:

- "*WriteVault*" stores arbitrary data.
- "*RevokeData*" sets data of a location as revoked. This operation prevents a secret from being accessed.
- "*GarbageCollect*" deletes from vaults all records marked as revoked.
- "*CopyRecord*" copies the content of a location into a new one.
- "*Slip10Generate*" generates a new seed according to SLIP-0010.
- "*Slip10Derive*" derives a SLIP-0010 child key, given a seed or a parent key. The chain code is returned.

- "*BIP39Generate*" generates a new BIP-39 seed and store it in a location. The correspondent BIP-39 mnemonic is returned.
- "*BIP39Derive*" derives the seed from a BIP-39 mnemonic and stores it in a new location.
- "*GenerateKey*" generates a *Curve25519* secret key and store it in a location.
- "*Ed25519Sign*" creates a digital signature using a private key stored in a provided location using the *EdDsa* signature scheme.
- "*X25519DiffieHellman*" generates a x25519 shared key in a new location.
- "*Hmac*": computes the *HMAC*, given the location of a secret key and a message.
- "*Hkdf*" computes a new hashed key derivation function and store it in a new location. The new location is return as a result
- "*ConcatKdf*" executes the concat KDF as defined in Section 5.8.1 of NIST.800-56A and store the new output in a new location.
- "*AesKeyWrapEncrypt*" uses a key stored in a record as a key encryption key (KEK). The cipher text of the secret key encrypted by the wrap key is returned.
- "*AesKeyWrapDecrypt*" decrypts the wrapped key and store it in a new location.
- "*Pbkdf2Hmac*" generates a key from the provided password and store it in a new location.
- "*AeadEncrypt*" encrypts data using AEAD encryption scheme with a key stored in a vault. The cipher text is returned.
- "*AeadDecrypt*" decrypts data using AEAD encryption scheme. The plaintext is returned.
- "*ConcatSecret*" concatenates two secrets and store the result in a new location.

IOTA Identity Framework compatibility

As stated in the preceding section, the IOTA Identity Framework incorporates a key storage implementation based upon Stronghold, named `StrongholdStorage`. The compatibility of algorithms for the key storage is provided by the intersection of Stronghold procedures and JWS compatible digital signature algorithms. Therefore, only HMAC [28] and EdDSA [29] can be used to generate a JWS with Stronghold. However, since HMAC is not based on asymmetric cryptography, `StrongholdStorage` implements only the EdDSA signature scheme.

2.6 Trusted Platform Module 2.0

Trusted Platform Modules are components that operate in the context of trusted computing. The trusted computing group (TCG) defines trust as «expectation of behavior» [6]. Therefore, trusted computing can be defined as *components that are behaving as expected*, so that such components are considered trusted.

Trusted computing requires the concept of Root of Trust, which refers to a component that must be trusted and for which any incorrect action is undetectable. The TCG identifies three distinct Root of Trust:

- *Root of Trust for Measurement* (RTM) performs integrity checks (measurement) and sends it to the RTS
- *Root of Trust for Storage* (RTS) collects and store measurements in a shielded memory, isolated from the measured platform.
- *Root of Trust for Reporting* (RTR) reports the content of the RTS, typically with a digital signature of the hash of the contents of the RTS.

The Trusted Platform Module is a system component that has a separated state from the system on which it reports. There are different implementations of a TPM. *Discrete* TPMs are isolated in a tamper resistant silicon, while *integrated* TPMs included in another chip, losing tamper resistance properties. Instead, *firmware* TPMs are provided on the host CPU in a special execution mode. In the context of trusted computing, the TPM acts as a Root Trust for Storage and Root of Trust for Reporting.

The following list enumerates the primary functionalities of a TPM 2.0:

- *Key management capabilities*: the TPM has its own random number generator, which can be used to generate keys. Moreover, keys can be used inside the volatile memory of the TPM and can be stored inside TPM non-volatile memory or externally, but protected through *binding* and *sealing*[6].
- *Identity management*: the TPM architecture provide an identity layer to authenticate the platform. This identity layer can be used during the process of remote attestation.
- *Non-volatile storage*: non-volatile memory of the TPM can be used to store arbitrary data (e.g. x509 certificate or secret that must be retrieved before the operating system is loaded)
- device attestation: being a RTR, the TPM can securely report the state of the platform, digitally signed.

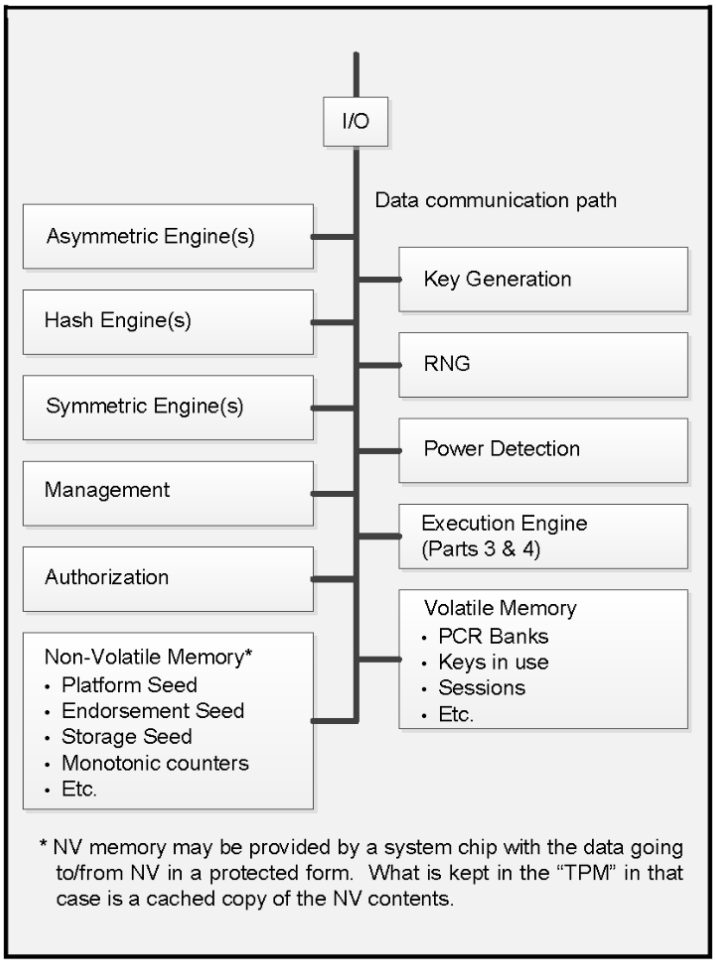

Figure 2.13: TPM 2.0 architecture [6]

2.6.1 Architecture

The figure 2.13 represents the architecture overview of the TPM. For discrete implementation, the TPM is connected to the host system through an external interface and communicate via an I/O buffer.

Cryptographic subsystem

The TPM 2.0 has a cryptographic subsystem that can be used by the device itself or requested by the host.

The following list enumerates the capabilities of the cryptographic subsystem:

- Key derivation functions

- Hashing functions
- Symmetric encryption
- HMAC
- Asymmetric encryption
- Sigital signature creation and verification
- Random number generator (it has its own source of randomness)

Authorization subsection

When a command is sent to the TPM, the authorization layer checks that resources are accessed only if the caller has an adequate authorization.

Volatile memory

Volatile memory contains all the information that the TPM uses in its running state. Obviously, when the TPM device is powered off the volatile memory is lost. It contains:

- *Platform Configuration Registers* (PCR). They're shielded locations that contain the content of measurements.
- *Object Store*: TPM keys and data are called objects. In order to be used, TPM objects must be loaded on the volatile memory. Objects can be loaded externally or load data from the non volatile memory of the TPM
- *Session Store*: TPM commands, particularly authorization, are provided through session. Sessions can be negotiated in order to execute actions on the TPM.

Generally, TPM resources are very limited. In particular, the minimum requirement for volatile memory is to be able to contain at the same time the following:

- Two loaded TPM objects
- Three authorization sessions
- The I/O buffer

Non-volatile (NV) memory

The TPM has a non-volatile memory as well to store persistent data. Similarly to volatile memory, resources are limited.

Power detection module

TPM has only an ON/OFF power state. However, the system host may have several states. The power detection modules tracks the power state of the host system and reset volatile memory data if needed.

2.6.2 TPM Software Stack 2.0

The TPM Software Stack 2.0, maintained by the TCG, is a set of APIs that facilitate communication between the system host and the TPM device. Figure 2.14 illustrates the API layers of the software stack.

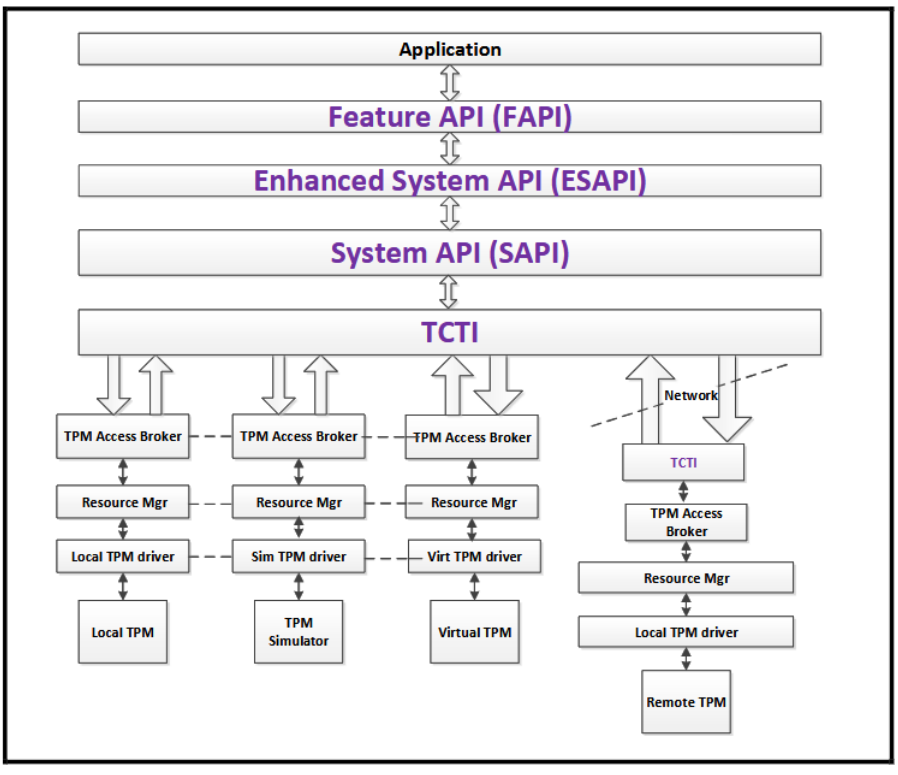

Figure 2.14: TPM Software Stack 2.0 structure [30]

TPM Device Driver

The TPM device driver is an operating system-specific software that handles communication between the hardware TPM and the operating system host.

TPM Access Broker and Resource Manager (TABRM)

This component is optional, but it relieves the user of the need to manually manage the TPM's resources. The volatile memory of TPM is very limited; therefore, the TABRM layer takes care of a series of operations.

The *Resource Manager* swaps objects, sessions and sequences from TPM memory. This is particularly useful when it is required to manage more than an object at the same time.

The *TPM Access Broker* takes manages the synchronization of interactions with the TPM. It allows different processes to interact with the TPM without collisions.

TPM Command Transmission Interface (TCTI)

The TPM command transmission interface (TCTI) is responsible for interfacing with different types of TPMs. It is the first common layer for each TPM implementation.

Marshalling/Unmarshalling API

This is not a proper layer of the stack. It is a common library that contains serialization and deserialization instruction for TPM's commands and responses.

System API (SAPI)

The system API provides access to TPM 2.0 functionalities. It is designed to operate with low level calls.

Enhanced System API (ESAPI)

The layer's functionality is contingent upon SAPI, providing a set of simplified abstractions in the form of system calls. It comprises functions for cryptographic operations, session management, and commands necessary for attestation.

Feature API (FAPI)

This is the highest level of abstraction of the TSS. This provides a simpler high-level abstraction for application developers. However, it is important to note that, given its purpose is to function within software applications, it does not offer all the capabilities provided by ESAPI.

2.6.3 TPM Hierarchical Object Structure

The TPM 2.0 specifications introduce the concept of hierarchies. The initial composition of each hierarchy is characterized by a seed, which is situated within the TPM device. The creation of objects within the TPM always requires the specification of a parent object; this can be either the hierarchy seed or other derived TPM objects. It is important to note that every TPM object is directly or indirectly associated with a hierarchy seed.

The TPM 2.0 has four different hierarchies [31]:

- The *Platform Hierarchy* is intended for use by the platform owner. Usually contains objects utilized during the initial boot sequence.
- The *Endorsement Hierarchy* is used by the privacy administrator. It is used for operations that require the TPM identity to be presented.
- The *Storage Hierarchy* is intended for use by the platform owner. Its utilization is recommended for non-privacy-sensitive operations.
- The *Null Hierarchy* is an ephemeral hierarchy with; each time the TPM is powered off the seed is dropped. A new one is generated at each new power cycle.

It is important to note that seeds of persistent hierarchies can be cleared and recreated independently.

Primary keys

The TPM incorporates cryptographic keys into the TPM object structure [32]. Consequently, keys are also part of a hierarchy.

Primary keys are TPM objects that have the hierarchy seed as parent object. The cryptographic material is derived directly from the hierarchy seed using TPM key derivation functions. The `TPM2_CreatePrimary()` ESAPI function from the TSS 2.0 is used to generate primary keys. The sensitive part of a primary key is never exported outside the TPM memory.

Despite the limited resources of the TPM, primary objects enable the possibility to generate an unlimited amount of keys. Indeed, keys are the result of a function that takes as input the seed of the hierarchy (already persistent) and a *public template* [32]. Public templates are defined as structures that contain the configuration of the TPM object to be created, including attributes, authorization values, key usage and key algorithms. In addition, a public template may contain the *unique data*, that uniquely characterize the public template. Consequently, primary key are generated in a deterministic manner; if the same public template is provided and the hierarchy seed has not changed, the TPM 2.0 can recreate the

exact key. This approach is advantageous in circumstances where it is necessary to have keys that can survive a power cycle without consuming the scarce resources of non-volatile memory.

Child keys

Specific types of primary objects can be used as parents for *child keys*. Child keys can be created with `TPM2_Create()` ESAPI function [33]. The generation of the object is analogous to primary keys. However, the parent object cannot be the hierarchy seed but a different TPM object. The sensitive part of the key is exported outside the the TPM memory, but it is wrapped by the parent key. Child keys with *restricted decrypt* attributes can also be parent for other keys.

2.6.4 Credential Protocol

The TPM architecture specification describes the credential protocol, which is

> a privacy preserving protocol for distributing credentials for keys on a TPM. The process allows a credential provider to assign a credential to a TPM object, such that the credential provider cannot prove that the object is resident on a particular TPM, but the credential is not available unless the object is resident on a device that the credential provider believes is an authentic TPM.[6]

Endorsement Key Credential Profile

Endorsement keys are asymmetric key pair generated by the TPM in the Endorsement hierarchy. They are primary objects directly derived from the hierarchy seed. The *TCG EK Credential Profile* specification describes the usage of endorsement keys and the creation of Endorsement Key Certificates [34]. Endorsement keys involved in the PKI define the digital identity of the TPM device.

Consequently, the creation of digital signatures with the EK may result in privacy concerns due to correlation. Indeed, the EK should never directly perform digital signatures. The solution provided by the TCG is the Credential Protocol [6]. A TPM can create an ephemeral signing key and present it with the EK to a credential provider that supports the protocol. The provider can verify the possession of the EK by the TPM and return a valid credential associated with the ephemeral signing key. The credential provider is trusted to not disclose any association between the EK and the signing key.

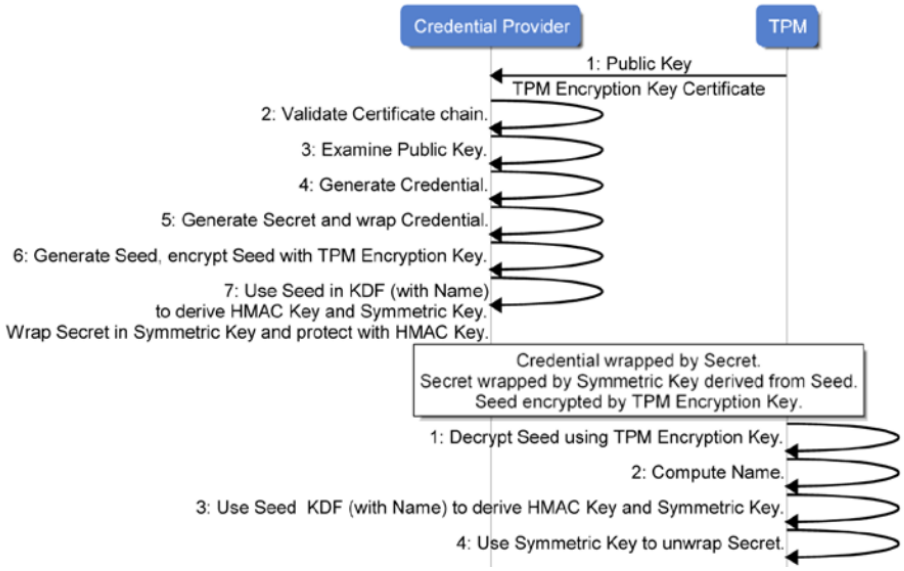

Figure 2.15: TCG Credential protocol example [31]

Credential generation

The Figure 2.15 illustrates a possible workflow proposed in [31] to issue a credential to an ephemeral signing key.

The protocol is composed of these steps:

1. The client TPM sends the public area of a TPM object corresponding to a new signing key and a valid EK certificate to the credential provider.
2. The credential provider validate the certificate chain of the EK certificate.
3. The credential provider examines key parameters and decides whether to issue a new credential or not.
4. The credential provider generates the new credential.
5. The credential provider generates a new symmetric key and uses it to wrap the credential.
6. The credential provider generates a seed and encrypt it with the public part of the EK.
7. The credential provider generates a new symmetric key derived from the seed and the TPM object name (the hash of the public part of the TPM object) of the credentialed object. Finally, the secret is encrypted by the generated symmetric key.

Steps 6 and 7 are performed by the `TPM2_MakeCredential()` function. It's worth noting that the make credential operation is stateless; it does not necessarily requires a TPM device to be executed. It should also be noted that the EK is used to encrypt a seed and not to sign data. This prevents the credential provider to have an evidence that correlates the EK with the ephemeral key.

The result of the make credential operation and the encrypted credential is sent back to the TPM client. Therefore, the TPM solves the challenge sent by the credential provider:

1. Decrypt the seed with the EK.
2. Compute the name of the signing key object.
3. Use the seed and the object name to derive the encryption key.
4. Unwrap the secret.
5. Finally the secret can be used to decrypt the encrypted credential.

The operation that the client TPM executes is the `TPM2_ActivateCredential()` ESAPI function. When the EK and the signing key are loaded in the TPM memory, it solves the challenge received from the provider and returns the secret key.

A successful challenge demonstrates that the TPM device is the owner of the EK and that an object with the name corresponding to the one computed during the generation of the challenge is loaded in the TPM memory. The credential provider can add validation steps or require additional proofs in order to issue a credential. For instance, the TPM can also load externally generated keys, the protocol itself does not guarantee that the key is created and managed by the TPM itself. On the other hand, the credential provider can accept only signing keys with specific attributes that guarantees that the key is generated by the TPM device itself.

Chapter 3

Solution Design

3.1 Rationale

SSI model provides a decentralized identity layer for humans, organizations or things. The model relies on verifiable data registries, used as a distributed public key infrastructure (DPKI).

In order to provide secure distributed key management, IOTA Identity Framework relies on the Stronghold project, which provide key management capabilities and data protection both at rest and at runtime. However, the runtime protection of keys relies exclusively on the mitigation mechanism of non-contiguous data types.

The first section of the chapter provides a possible implementation of key storage for IOTA Identity Framework, using a TPM 2.0 as a key management system.

The primary rationales for the selection of the TPM are as follows:

- It is relatively inexpensive and low power. Therefore, it is suitable for being integrated with IoT devices.
- Discrete and integrated TPMs are more resistant to software attacks. The cryptographic material is always loaded in the isolated TPM environment and managed by the cryptographic subsystem.
- The TPM is already capable of managing a digital identity. It relies on a PKI based on X.509 certificates that aim to provide a digital identity to the system host.

TPM capabilities can be integrated in the SSI model, especially for IOTA Identity, which provides key storage interfaces that can be easily implemented. The following section describes the development of a library that uses TSS 2.0 ESAPI functions to implement key storage interfaces of the IOTA Identity Framework. This library enables a TPM device to support SSI operations (e.g. creating a DID

document or digitally sign a Credential), managing the required cryptographic operations.

Section 2.6 describes in detail the capabilities of the cryptographic subsystem of the TPM and how the cryptographic material is represented in the TPM context. A subset of TPM keys can express a stronger form of authentication. For instance, a primary key depends directly on the hierarchy seed resident in the TPM device. Consequently, that primary key can be used exclusively on the specific TPM device that includes the hierarchy.

The final section of the chapter describes possible solutions for anchoring a self-sovereign identity to a TPM 2.0. The solution consists in a revision of the TPM credential protocol [6] for requesting and issuing a verifiable credential. Then, the credential issuer is able to generate a new verifiable credential attesting that the key pair used by the holder uniquely corresponds to a TPM device.

3.2 Tpm storage

This section describes the design of the implementation of a library, named `TpmStorage`, that implements IOTA identity key storage interfaces using a TPM device and the TPM Software Stack 2.0.

3.2.1 Overview

As observed in the IOTA Identity section of Chapter 2, two generic interfaces are provided to extend functionalities of a key management system, `JwkStorage` and `KeyIdStorage`. The IOTA Identity Framework is written using the Rust programming language.

Analogously, the implementation has been performed inside a fork of the IOTA Identity Framework [26]. Therefore, `TpmStorage` is provided as a library inside the IOTA Identity Framework, similarly to the other existing implementation of `MemStore` and `StrongholdStorage`.

The `TpmStorage` library interacts with the TSS through Rust bindings of TSS 2.0 ESAPI layer [35].

3.2.2 Architecture

The Figure 3.1 illustrates at a high level of abstraction the interaction of different entities involved in the IOTA Identity Framework: as already mentioned, the verification methods contained into the DID document are identified by a method digest and the `KeyIdStorage` associates the method digest to a unique key identifier. The `TpmStorage` provides an implementation of the `JwkStorage` interface: it takes as input the key identifier to uniquely address to the JWK. However, the key

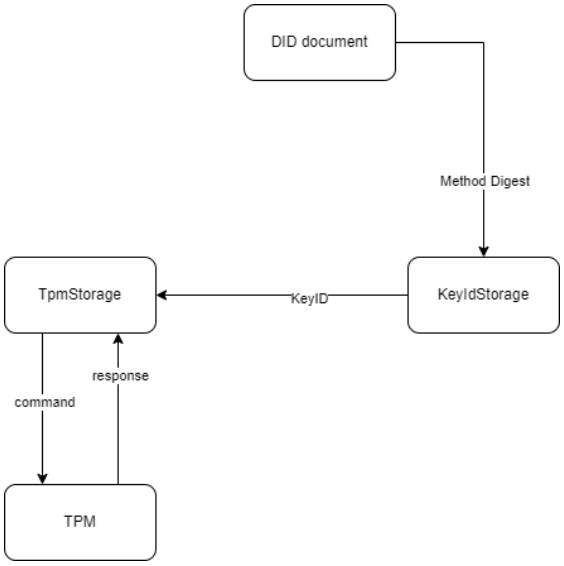

Figure 3.1: High level design of key storage implementation

identifier is also involved in the key generation process; the identifier is used as unique data for the generation of primary key in the TPM. This approach enables the generation of key pairs that are independent of the limited resources of the TPM device. Considering the hierarchy seed as unchanged, if a TPM object is not loaded in the volatile memory, it can be recreated providing the same template as input.

3.2.3 Key generation

In the event of the generation of a new verification method. Key storage interfaces are involved as follows to generate a new key pair and return a new public JWK:

1. The DID document Rust abstraction request for a new verification method to the `TpmStorage`
2. A public template with a new unique data is generated. Then, the `TPM2_CreatePrimary()` ESAPI function is called to generate a new primary key.
3. If the command is completed successfully, an opaque address, a transient *handle*, and the new public TPM object data are returned.
4. The transient handle is mapped with the corresponding key identifier. It is highly probable that a generated object will be utilized. Therefore, the internal cache tracks the objects loaded in the volatile memory and retrieves the handle instead of recreating them when they need to be used.

5. the TPM public area is encoded as a public key JWK according to the section 6 of RFC-7518 [28].

6. `TpmStorage` returns the key identifier and the corresponding public key in JWK format.

Output format

The result is a Json Web Key compliant with the format specified in RFC-7518 like the one present in 2.4.

The `kid` properties in the JWK contains the object name of the TPM. As suggested by [31] in Chapter 8, the entity name is stored combined with the public key at the creation time. Considering the fact that TPM handles are opaque pointers, it could be possible by a malicious actor to swap the object pointed by the handle so that an operation is performed with an unintended key. However, in this case it is possible to check that the object name inserted in the JWK corresponds to the name of the object pointed by the transient handle.

TPM public template

The key generation process in `TpmStorage` currently support only one template. The following object attributes are set for each object:

- `fixedTPM`: the key cannot be duplicated in a different TPM
- `fixedParent`: the key cannot change its parent object
- `sign`: the key can be used for signatures

The single supported signature scheme is ECDSA with SHA-256 hashing algorithm. Key pair is generated on *secp256r1* curve.

Finally, the unique data field contains the provided key identifier, which is a 32-byte random generated with the TRNG of the TPM device.

3.2.4 Signature

`JwkStorage` provide a `sign()` method that generate a digital signature to be included in a JWS. The implementation of `TpmStorage` requires two prerequisites to perform sign operation:

- A suitable key has been created and loaded in the TPM volatile memory.
- The internal cache contains the mapping of the transient handle corresponding to the loaded TPM object and the matching key identifier.

It should be noted that this prerequisite corresponds to the state of the `TpmStorage` after the key generation phase. Consequently, after the key generation phase is completed it is possible to execute the sign operation.

The sign operation from `JwkStorage` requires three input parameters:

- A key identifier
- The corresponding public JWK
- the message to be signed

The sign operation proceeds as follows:

1. The internal cache is queried to retrieve the transient TPM handle, given the corresponding key identifier.
2. The public part of the TPM object is read and the object name is computed to be compared with the `kid` field of the public JWK. This verifies that the loaded object in the TPM memory corresponds to the intended signing key.
3. Compute the digest of the message with the SHA-256 hash function.
4. Sign the digest with the TPM using TSS ESAPI API.
5. Signature bytes are encoded and returned to the caller.

3.2.5 Key deletion

`JwkStorage` requires the implementation of the delete operation. It is necessary that the key is deleted and purged so that it cannot be recovered anymore. Considering the key generation process, it is evident that the generated primary key cannot be properly purged; it is a specific derivation of a key derivation function that takes as input the hierarchy seed and the public template with unique data. As long as both information are available for the TPM, the same key can be regenerated also after the delete operation. Consequently, the TPM object key can be recreated as long as the value can be read. When the delete operations is executed:

1. The corresponding object handle is deleted from the internal cache.
2. The object is removed from the volatile memory of the TPM.

However, it should be noted that it does not guarantee that a key has been deleted and cannot be recovered. The irreversibility of the operation can be obtained by removing any reference to key identifiers. All the references are eliminated, but not from the mapping of the `KeyIdStorage`. Therefore, even though the key identifier references are removed from the `TpmStorage`, the irreversibility of the operation is obtained when some `KeyIdStorage` implementation drops the value as well.

3.2.6 Data Persistence

The architecture of the `TpmStorage` described above does not keep data in non-volatile memory. As a consequence, if the system host goes through a power cycle or the TPM device is reset. In such cases, volatile memory is lost. However, key generation of this implementation is deterministic and the same keys can be recomputed if are both available the hierarchy seed and the key identifier.

Hierarchy management

In order to regenerate keys after power cycles, TPM objects keys must be created under one of the three persistent hierarchy of the TPM; if keys are created using the seed of the Null hierarchy, the seed is regenerated at each power cycle, making impossible for identity keys to be recovered. TPM keys for used in `TpmStorage` are created childs of the *Storage Hierarchy Primary Seed*. It is a persistent hierarchy, therefore the seed is stored in the TPM non-volatile memory and is not dropped after a power cycle.

The choice of generating keys in the Storage hierarchy instead of the Endorsement hierarchy has been determined by the intention of keep an independent management of the keys for SSI and keys that are used for the digital identity of the TPM in trusted computing.

Key identifier management

The generation of a key also depends on the public template provided to the TPM as input of the `TPM2_CreatePrimary()` ESAPI function. Currently, `TpmStorage` supports the generation of a single key type. Therefore, public templates are distinguished only by the *unique data* property. Indeed, it has been established by design to use the key identifier as a value for *unique data*. Therefore, if the hierarchy seed unchanged, the generation of different keys exclusively depends on the value of the key identifier.

As already discussed, the key identifier is a sufficiently long random sequenc of bytes generated by the TPM, ensuring a sufficient entropy to the primary key generation. Even though the IOTA Identity Framework uses the key identifier as a form of local addressing, it still determines the life cycle of the identity keys; if the mapping of the `KeyIdStorage` is lost, it would be impossible to retrieve a key starting from the public JWK of the verification method.

`TpmStorage` directly binds the key generation process to key identifiers. If the hierarchy seed does not change, the persistence of the key identifiers determines the persistence of the TPM keys. Consequenly, keys can be easily recreated as long as the implementation of the `KeyIdStorage` exists and contains key identifiers. For instance, if the implementation stores the mapping in a persistent storage (e.g.,

a JSON file), then the correspondent keys can be recreated; it is possible to read the saved key identifiers and provide them as input to the key generation function, satisfying the prerequisites required for the usage of TPM keys as described in Section 3.2.4. On the other hand, a volatile implementation, like `MemStore`, loses all data when a power cycle happen. A volatile implementation could still be acceptable for some use cases; when the system starts, it is possible to interact with the DLT to update the DID document and replace it with a new verification method or publish a new one obtaining a new DID. This approach forces to apply a key rotation strategy.

Generally, `TpmStorage` can work with any implementation of `KeyIdStorage`. IOTA Identity Framework has only two implementations available:

- `KeyIdMemstore`: the in-memory implementation of `KeyIdStorage`. It is not a persistent storage, but the access to the mapping is efficient.
- `StrongholdStorage`: key identifiers mapping is stored as caching data in the stronghold snapshot. It provides data persistence and integrity with a loss in performance.

3.3 TPM Verifiable Credentials

The first part of the chapter described a solution that uses a TPM 2.0 device as a key provider for supporting operations to manage a self-sovereign identities. Identity keys are handled in the isolated environment on the TPM device. The provided implementation of `JwkStorage` generates keys that are directly dependent on the storage hierarchy seed and the sensitive part is always resident inside the TPM volatile memory. As a consequence, the keys used to operate with an SSI are tied directly to the TPM; these keys cannot exist if the device that created them is not available.

This section provides a solution to express the aforementioned result in the SSI context. An holder of an identity key generated with the TPM 2.0 should be able to provide evidence that the key in use can be used exclusively by a specific TPM device.

The TPM 2.0 already support identities for the RTR with the credential protocol. In fact, TPM vendors usually generate one or more X.509 certificates for primary keys in the endorsement hierarchy. Endorsement keys are not used for signing content of applications; the ability to correlate signatures may be privacy concerning. The credential protocol provides a solution for the issuance of credentials; a TPM can present the EK certificate and a signing key to a credential provider, which is trusted to not leak any information about correlation between the EK and the new signing key. Subsequently, the credential provider issues a credential for the signing key.

3.3.1 TPM Credential Protocol for Verifiable Credentials

The credential protocol does not specify the format of the credential generated by the credential provider. Therefore, the same protocol can be used to generate a Verifiable Credential. Considering the trust triangle, the holder can request to the issuer a Verifiable Credential that asserts that a verification method in a DID document is derived by a TPM key that is compliant with a fixed policy for key creation.

Key policy

Initially, identity keys must come from allowed TPM devices; the issuer trusts root certificates of the Certification Authority that issued the EK certificate of TPMs. In a credential request, the issuer must receive the EK certificate, the public area of the TPM object key and the DID of the holder. In addition, it validates the certificate chain up to the root certificate. The issuer reads the TPM public area object and computes the name of the object. It also resolve the DID and verifies the public JWK in the verification method, checking that the computed name corresponds to the value stored in the `kid` property. At this point, both keys are associated to a verifiable identity; the EK is provided as a valid X.509 certificate and the public TPM object corresponds to a valid verification method for the provided DID.

The issuer validates the key object and decide whether to issue a VC. It checks that:

- `fixedTPM` is SET.
- `fixedParent` is SET.
- `decrypt` is CLEAR.
- `sign` is SET.

This attributes ensures that the TPM key is a signing key that can be used exclusively by a specific TPM device.

VC model

Following the successful validation of the key by the issuer, it adds a claim in the Verifiable Credential to assert that the key that has been provided satisfies the policy. Then, the issuer computes the digest of the public key in the verification method. In particular, the digest computation for a *secp256r1* public key as been done as follows:

```
SHA-256(x||y)
```

where x and y are the coordinates on the curve.

Figure 3.2 illustrates an the payload of the Verifiable Credential that contains the digest reference to the validated key.


```
{
  "@context": "https://www.w3.org/2018/credentials/v1",
  "id": "https://example.com/credentials/3732",
  "type": [
    "VerifiableCredential",
    "TpmCredential"
  ],
  "credentialSubject": {
    "id": "did:iota:tst:0xbb5fef06b3bd8bac1284861ecb2885a73eef8ecbfa1a195571e4754a536adce7",
    "sha256": "fXQHrD8_6dFa1Ynz66D3b0k90tjpP5GtIQs5ZJqGfbY"
  },
  "issuer": "did:iota:tst:0x2badaa70d6b27aabe8d1883cec880f12954b8018af6a3ea2c0c6e17d05cec2ed",
  "issuanceDate": "2025-03-23T21:12:03Z"
}
```


Figure 3.2: Custom verifiable credential for `TpmStorage`

3.3.2 Proposed solutions

In the event of a successful validation of parameters, the issuer sends a challenge to the holder. The issuer creates a challenge with ESAPI `TPM2_MakeCredential()` function and the holder solves the challenge with ESAPI `TPM2_ActivateCredential()` function. The holder can solve the challenge only if both the EK and the identity key are loaded in the TPM memory.

The validation of the key performed by the issuer, determines that the holder can solve the challenge only if it is the owner of both the Endorsement Key and the identity key.

The following clauses describe two different approaches implemented, using IOTA Identity Framework and `TpmStorage`.

Actors

In order to present the proposed solutions, the involved actors are defined in order to properly describe the protocol steps:

- *TpmStorage*.
- *Holder*: it is a running application that uses IOTA Identity Framework.
- *IOTA Tangle*: it is the verifiable data registry used for SSI
- *Issuer*: it is an application that can issue a Verifiable Credential to the holder.

It is supposed that Holder and Issuer have already published their DID documents on IOTA Tangle and that the parties communicate over a secure protocol (e.g., HTTPS).

1-RTT solution

The first solution is inspired by the example of credential provisioning presented in [31].

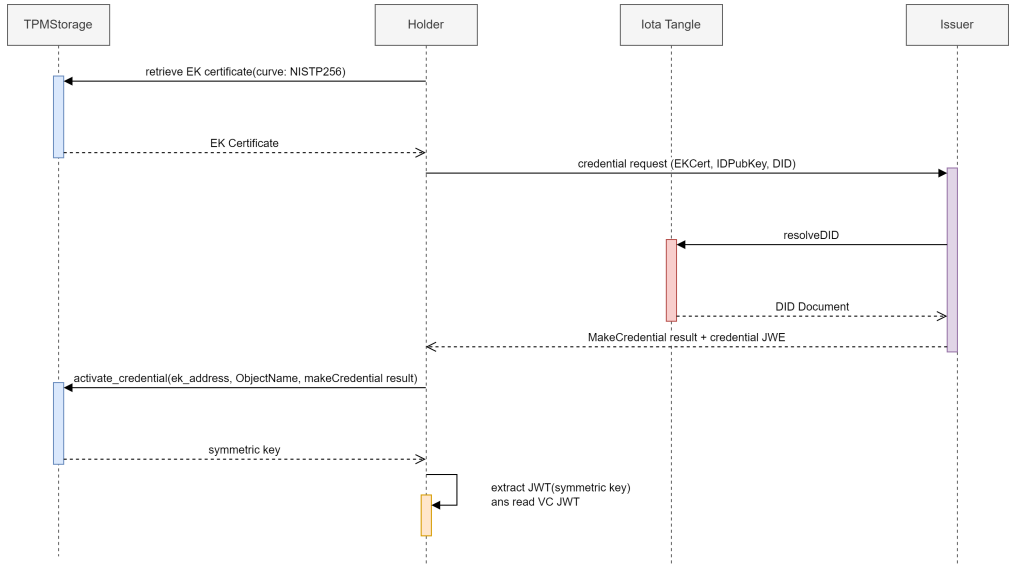

Figure 3.3: Sequence diagram of 1-RTT solution for TPM credential issuance

In detail, the protocol is executed as follows:

1. Holder asks the `TpmStorage` to retrieve the EK certificate. The certificate is available in the TPM non-volatile memory in a specific location according to TCG specifications [34].

2. Holder sends to the Issuer a request for a new credential, sending the EK certificate, the TPM key object and a DID.

3. The issuer validate the parameters received from the holder.

   - The holder DID is resolved into the corresponding DID document.
   - The name of the TPM key object is computed and compared with the `kid` property of the public JWK contained in the DID document.
   - The TPM key object is validated to ensure that satisfies the key policy.

4. The issuer prepares the challenge to be sent to the holder:

   - The EK public key is extracted from the EK certificate, according to default templates defined in [34].
   - A 32-bytes symmetric key is generated.

- The `TPM2_MakeCredential()` operation is executed to protect the symmetric key.

5. The issuer creates a new Verifiable Credential for the holder.
   - The digest of the public key is computed and included as a claim.
   - A cryptographic proof is added to the Verifiable Credential.
6. The issuer wraps the credential into a JWE. The payload is encrypted with the generated symmetric key.
7. The issuer sends to the holder the challenge and the wrapped VC.
8. The holder request to the `TpmStorage` to solve the challenge with `TPM2_ActivateCredential` and return the symmetric key.
9. The holder unwraps the VC using the symmetric key.

2-RTT solution

The second proposed solution completes the credential issuance in two Round Trip Time instead of a the single one needed by the first solution. On the other hand the encryption of the Verifiable Credential as a JWE is not required in this case.

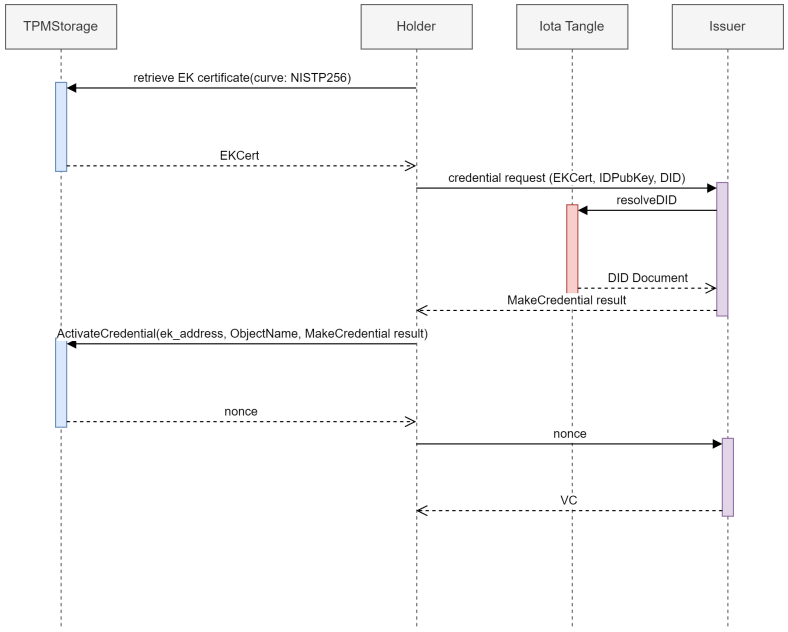

Figure 3.4: Sequence diagram of 2-RTT solution for TPM credential issuance

In detail, the protocol is executed as follows:

1. Holder asks the `TpmStorage` to retrieve the EK certificate. The certificate is available in the TPM non-volatile memory in a specific location according to TCG specifications [34].
2. Holder sends to the Issuer a request for a new credential, sending the EK certificate, the TPM key object and a DID.
3. The issuer validates the parameters received from the holder (see 3.3.2).
4. In the event of a successful validation, the issuer prepares a challenge for the holder:
   - The EK public key is extracted from the EK certificate, according to default templates defined in [34].
   - A 32-bytes nonce.
   - The `TPM2_MakeCredential()` operation is executed to protect nonce.
5. The issuer sends to the holder the challenge.
6. The holder request to the `TpmStorage` to solve the challenge with `TPM2_ActivateCredential` and return the symmetric key.
7. The holder sends the nonce back to the issuer, proving that the challenge has been solved.
8. The issuer generates a new VC for the holder (see 3.3.2).
9. The issuer sends the Verifiable Credential to the holder.

Verifiable Credential Verification

The preceding clauses described the procedure with which a credential holder proves to a trusted issuer to possess a verification method that it is anchored to a TPM device. In the trust triangle, a verifier accepts the claims that the issuer made about the holder. Therefore, the verifier can trust the holder to be an SSI anchored to a TPM device.

As illustrated in Figure 3.5, the protocol is executed as follows:

1. The holder request a nonce to the verifier. It will be included in the Verifiable Presentation to prevent replay attacks.
2. The holder creates a Verifiable Presentation containing the issued Verifiable Credential. The Verifiable Presentation must be signed with the key verified by the issuer.

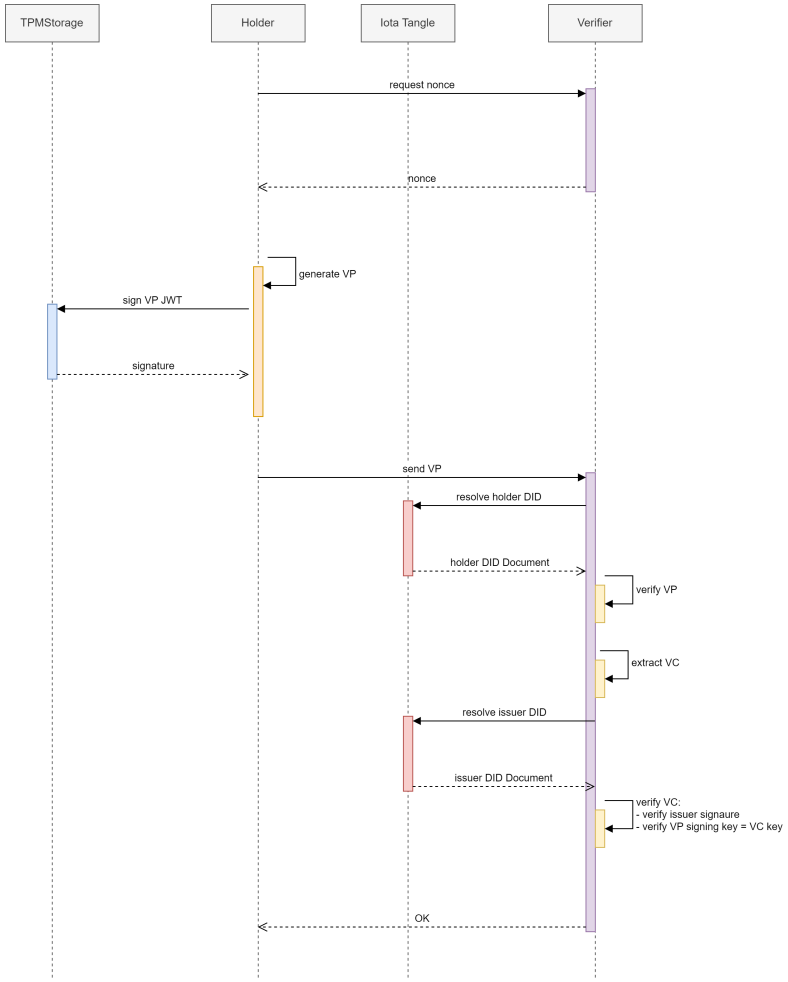

Figure 3.5: Sequence diagram of VP verification

3. The holder sends the VP to the verifier.
4. The verifier validates the VP received from the holder:
   - Holder's DID is extracted and resolved into a the corresponding DID document
   - The proof of the presentation is verified
   - Issuer's DID is extracted and resolved into the corresponding DID document
   - The proof of the Verifiable Credential is verified
   - The digest of the holder's verification method is computed and compared with the digest verified by the issuer

Chapter 4

Testing and results

This chapter describes the testing phase and the results obtained for the solutions provided in the preceding chapter. Each of the following tests are executed with three different key storage implementations: `MemStore`, `StrongholdStorage` and `TpmStorage`. However, it has not been possible to run tests with the same key type and signature algorithm; key storage implementations in IOTA Identity and the TPM 2.0 do not share a common signature scheme. Therefore, `TpmStorage` uses *secp256r1* key pairs and *ECDSA* signature scheme. On the other hand, `MemStore` and `StrongholdStorage` use *curve25519* key pairs and *EdDSA* signature scheme.

4.1 Test environment

The tests needs intends to simulate the behavior of an IoT device that uses its own SSI to interact with other actors.

4.1.1 Testbed

All tests are executed on a *Raspberry Pi 4 Model B*, having *Raspberry Pi OS* installed via the *Raspberry Pi Imager*. Additionally, it has been installed a discrete TPM 2.0, model *OPTIGA™ TPM SLB 9670 TPM2.0* manufactured by *Infineon*.

4.1.2 Actors

Testing includes the establishment of trust relationships according to the trust triangle. Therefore, three different actors have been used during the testing phase

- *Issuer*: it is an application capable of both issuing traditional credentials and credentials for `TpmStorage`.

- *Verifier*: it is an application that verifies VPs received from the credential holder. It supports the validation of credentials for `TpmStorage`
- *Holder*: it is a set of binary executables that contains the test cases.

4.2 Test cases

Each different implementation of key storage is used in four different tests:

1. Generation of a new key pair, returning the correspondent public JWK.
2. Creation of a new DID document that includes a verification method, provided by the key storage.
3. Holder connects with the issuer to obtain a Verifiable Credential.
4. Holder sends a Verifiable Presentation and waits for a successful verification from the Verifier.

A single test contains 100 measured iterations, from which the arithmetic mean, median and standard deviation have been computed.

4.2.1 Key generation

This test measures the key generation operation defined by the `JwkStorage` interface.

	Mean (ms)	Median (ms)	Std. deviation
`MemStore`	0.148	0.144	0.015
`StrongholdStorage`	3,050.004	3,040.229	27.338
`TpmStorage`	285.925	285.787	2.380

Table 4.1: Key generation results

The outcomes of the tests differ significantly for each implementation. `MemStore` has obviously the best performance, because it does not implement any protection for the generated JWKs. In contrast, `StrongholdStorage` is severely affected by the security features implemented. The generation of a new key results in its immediate inclusion within a snapshot file. However, this requires to serialize a new snapshot that is encrypted with a new symmetric key. This operation is fully dependent on the processing power of the host system. Consequently, the less processing power a device has, the longer it takes to generate a key. `TpmStorage` obtained a better result than the `StrongholdStorage`, but it is significantly distant

from the results obtained by `MemStore` . In order to generate a key, `TpmStorage` sends several commands to the TPM to receive the primary key, compute the name, and generate the random for the key identifier. Communication with the device is the slowest part of the process and determines the result of the test.

4.2.2 DID document creation

The test consists in the generation of a simple DID document that contains a single verification method. The key of the verification method is provided by the distinct implementation of key storage.

	Mean (ms)	Median (ms)	Std. deviation
`MemStore`	0.175	0.169	0.028
`StrongholdStorage`	6,161.352	6,135.867	59.737
`TpmStorage`	282.766	285.425	8.28

Table 4.2: DID document creation results

`MemStore` and `TpmStorage` obtained similar result respect to the test on key generation. On the other hand, `StrongholdStorage` doubled the execution time. Indeed, `StrongholdStorage` also implements the `KeyIdStorage` interface. Therefore, it saves in the snapshot file the key identifier, which triggers a new commit that updates the snapshot.

4.2.3 Verifiable Credential issuance

In this test, the holder requests a credential to the issuer and receives a valid Verifiable Credential. The test measures the time elapsed from the credential request to the reception of the plain text VC by the issuer. In addition, `TpmStorage` requests the credential using the custom version of the credential protocol. The test also measures the amount of data transmitted and received, expressed in bytes.

	Mean (ms)	Median (ms)	Std. deviation	TX (bytes)	RX (bytes)
`MemStore`	113.3568	108.178	15.072	649	791
`StrongholdStorage`	159.612	154.491	33.218	649	791
`TpmStorage - 2RTT`	1,617.841	1,610.206	25.12	1133	1051
`TpmStorage - 1RTT`	1,541.677	1,539.091	19.98	990	1441

Table 4.3: VC issuance results

It is worth noting that `MemStore` and `StrongholdStorage` obtained similar results in this test. The overhead generated by the networking operations is sufficient to remove the performance gap between the two implementations. Conversely, the

`TpmStorage` took more than a second to complete an iteration. It has to interact with the TPM device even more than previous tests to recover the endorsement key and its certificate, recover the identity key and solve the challenge provided by the credential issuer. Moreover, the issuer has to interact with its own TPM device to generate the challenge for the holder.

4.2.4 Verifiable Presentation verification

The last test consists in the last phase of transitive trust establishment. A credential holder tries to authenticate to the verifier, providing its own Verifiable Presentation. The holder measurements include the generation of a Verifiable Presentation and a successful verification provided by the verifier.

	Mean (ms)	Median (ms)	Std. deviation	TX (bytes)	RX (bytes)
`MemStore`	324.127	315.829	38.717	1727	48
`StrongholdStorage`	324.025	318.670	30.24	1727	48
`TpmStorage`	394.422	393.299	26.81	1859	48

Table 4.4: VP verification results

In addition, it has been also extracted the creation time of a Verifiable Presentation for each key storage.

	Mean (ms)	Median (ms)	Std. deviation
`MemStore`	21.542	20.438	8.115
`StrongholdStorage`	23.861	22.352	8.800
`TpmStorage`	189.300	191.188	12.740

Table 4.5: VP creation results

The test confirms the considerations formulated for the issuance of Verifiable Credential. Both `MemStore` and `StrongholdStorage` takes similar time to generate digital signature. Instead, `TpmStorage` has a consistent overhead provided by the communication with the TPM device.

Chapter 5

Conclusions and future works

The expected results for the thesis work has been fully achieved. Initially, the TPM has been identified as a suitable key management system for self-sovereign identity. Indeed, the implementation of the `TpmStorage` demonstrated that it is possible to complete operations in the SSI model with the support of a TPM device, while managing the cryptographic material in an environment isolated from the system host. The storage implementation enabled the generation of primary keys encoded as public JWK in DID documents, the completion of challenges to prove the ownership of the DID, as well as to provide proofs included in Verifiable Presentations, without being limited by the scarce resources of the TPM device.

In addition, the characteristics of the TPM have been exploited to provide a stronger concept of decentralized digital identity in the IoT domain. The revision of the credential protocol for the issuance of Verifiable Credential generate credentials that can be used specifically by a unique TPM device. Therefore, the proposed solution actually bounds a self-sovereign identity to a physical TPM device.

The integration of the TPM capabilities and SSI can be further investigated in two main aspects: performance and digital identity integration. The current implementation of `TpmStorage` requires to access several times to the TPM in order to complete stateless operations. For instance, during the test phase it has been used the TPM to generate the challenge for the issuance of the Verifiable Credential. Software implementation of stateless commands reduce the number of commands executed by the TPM, resulting in an improvement in performance.

The result of this thesis provided the anchoring of an SSI to a single TPM device. However, future developments can extend the binding not only to a physical device but also to the state of the system host. Indeed, the usage of Extended Authorization Policies for identity keys enable to require that certain conditions

about the state of the system host are verified before allowing the device to use a key. For instance, such policies may prevent the usage of identity keys if the state of the system host is potentially compromised.

Appendix A

User Manual

This chapter will provide a guide to replicate the same environment used for the development and testing of `TpmStorage` on a *Raspberri Pi 4*.

A.1 Requirements

The requirements section provide a guide to install all the software required to run the `TpmStorage`.

A.1.1 TPM Software Stack

The TPM Software Stack is provided by the Trusted Computing Group. It contains API to interact with a TPM. The version *4.1.3* has been used to successfully run the software.

The following instructions are required to install the version 4.1.3 of the TPM Stoftware Stack 2.0:

```
$ git clone https://github.com/tpm2-software/tpm2-tss
$ cd tpm2-tss
$ git checkout 4.1.3
$ ./bootstrap
$ ./configure
$ make -j$(nproc)
$ sudo make install
$ sudo ldconfig
```

A.1.2 TPM Access Broker and Resource Manager

The execution of some examples of the developed software required different processes to access the same TPM device. The resource manager offloads the user to manually export and reload the context of a TPM.

These are the instructions to install version 3.0.0 of the resource manager:

```
$ git clone https://github.com/tpm2-software/tpm2-abrmd
$ git checkout 3.0.0
$ ./bootstrap
$ ./configure
$ make -j$(nproc)
$ sudo make install
$ sudo ldconfig
```

A.1.3 OpenSSL

The OpenSSL library is used by the code examples of the issuer to decode and verify X.509 certificate chains.

OpenSSL can be installed as follows:

```
$ sudo apt install openssl
```

A.1.4 Rust

The code examples of `TpmStorage` are written in Rust. Therefore, it is required to install it with the following command:

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

A.2 TPM Storage

The `TPMStorage` is available on GitHub as a fork repository of IOTA Identity: [26]

A.2.1 Install

The software can be downloaded as a git repository:

```
$ git clone https://github.com/Cybersecurity-LINKS/tpmstorage-
    identity.rs.git
$ git checkout tags/thesis
$ cd tpmstorage-identity.rs
```

It possible to run all the benchmarks executed for the testing phase through the command:

```
$ cargo run --release --example <example_name>
```

Bibliography

[1] National Institute of Standards and Technology. *NIST Special Publication 800-63-3: Digital Identity Guidelines*. Tech. rep. SP 800-63-3. Accessed: 2025-03-24. National Institute of Standards and Technology, 2017. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63-3.pdf (cit. on p. 1).

[2] A. Preukschat and D. Reed. *Self-Sovereign Identity*. Manning, 2021. Chap. 19 (cit. on p. 1).

[3] Eryk Schiller, Andy Aidoo, Jara Fuhrer, Jonathan Stahl, Michael Ziörjen, and Burkhard Stiller. «Landscape of IoT security». In: *Computer Science Review* 44 (2022), p. 100467. ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2022.100467. URL: https://www.sciencedirect.com/science/article/pii/S1574013722000120 (cit. on pp. 1, 4).

[4] A. Preukschat and D. Reed. *Self-Sovereign Identity*. Manning, 2021. Chap. 1 (cit. on pp. 2, 3, 5, 11).

[5] A. Preukschat and D. Reed. *Self-Sovereign Identity*. Manning, 2021. Chap. 10 (cit. on p. 4).

[6] Trusted Computing Group (TCG). URL: https://trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-1.83-Part-1-Architecture.pdf (cit. on pp. 4, 28, 29, 34, 38).

[7] IOTA Foundation. URL: https://wiki.iota.org/identity.rs/welcome/ (cit. on pp. 4, 18, 19).

[8] *Decentralized Identifiers (DIDs) v1.0*. World Wide Web Consortium (W3C), 2022. URL: https://www.w3.org/TR/2022/REC-did-core-20220719/ (cit. on pp. 5, 8, 11, 18).

[9] Trust Over IP Foundation. *Introduction to Trust Over IP*. Tech. rep. Trust Over IP Foundation, 2021. URL: https://trustoverip.org/wp-content/uploads/Introduction-to-ToIP-V2.0-2021-11-17.pdf (cit. on pp. 6, 7, 11).

[10] Internet Engineering Task Force (IETF). URL: https://datatracker.ietf.org/doc/html/rfc7517 (cit. on p. 9).

[11] *Verifiable Credentials Data Model v1.1*. World Wide Web Consortium (W3C), 2022. URL: https://www.w3.org/TR/2022/REC-vc-data-model-20220303/ (cit. on pp. 9, 18, 20).

[12] Serguei Popov. *The Tangle*. White Paper. 2018. URL: http://cryptoverze.s3.us-east-2.amazonaws.com/wp-content/uploads/2018/11/10012054/IOTA-MIOTA-Whitepaper.pdf (cit. on pp. 11–13).

[13] IOTA Foundation. URL: https://wiki.iota.org/tips/tips/TIP-0012/ (cit. on p. 12).

[14] IOTA Foundation. URL: https://wiki.iota.org/tips/tips/TIP-0024/ (cit. on p. 12).

[15] Serguei Popov et al. *The coordicide*. 2020. URL: https://files.iota.org/papers/20200120_Coordicide_WP.pdf (cit. on p. 13).

[16] IOTA Foundation. URL: https://wiki.iota.org/learn/protocols/coordinator/ (cit. on p. 13).

[17] Ethan Buchman, Jae Kwon, and Zarko Milosevic. *The latest gossip on BFT consensus*. 2019. arXiv: 1807.04938 [cs.DC]. URL: https://arxiv.org/abs/1807.04938 (cit. on p. 13).

[18] IOTA Foundation. URL: https://wiki.iota.org/tips/tips/TIP-0002/ (cit. on p. 13).

[19] IOTA Foundation. URL: https://wiki.iota.org/tips/tips/TIP-0029/ (cit. on p. 13).

[20] IOTA Foundation. URL: https://wiki.iota.org/tips/tips/TIP-0020/ (cit. on p. 14).

[21] IOTA Foundation. URL: https://wiki.iota.org/tips/tips/TIP-0018/ (cit. on pp. 15, 16).

[22] IOTA Foundation. URL: https://wiki.iota.org/tips/tips/TIP-0019/ (cit. on p. 18).

[23] Internet Engineering Task Force (IETF). URL: https://datatracker.ietf.org/doc/html/rfc7638 (cit. on p. 21).

[24] IOTA Foundation. URL: https://wiki.iota.org/stronghold.rs/welcome/ (cit. on pp. 23, 24, 26).

[25] IOTA Foundation. URL: https://github.com/iotaledger/iota-sdk (cit. on p. 23).

[26] IOTA Foundation. URL: https://github.com/iotaledger/identity.rs (cit. on pp. 23, 38, 57).

[27] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography engineering: design principles and practical applications*. John Wiley & Sons, 2011 (cit. on p. 24).

[28] Internet Engineering Task Force (IETF). URL: https://datatracker.ietf.org/doc/html/rfc7518 (cit. on pp. 27, 40).

[29] Internet Engineering Task Force (IETF). URL: https://datatracker.ietf.org/doc/html/rfc8037 (cit. on p. 27).

[30] Trusted Computing Group (TCG). URL: https://trustedcomputinggroup.org/wp-content/uploads/TSS_Overview_Common_v1_r10_pub09232021.pdf (cit. on p. 31).

[31] Will Arthur, David Challener, and Kenneth Goldman. *A Practical Guide to TPM 2.0*. Apress, 2015. ISBN: 978-1-4302-6583-2. DOI: https://doi.org/10.1007/978-1-4302-6584-9 (cit. on pp. 33, 35, 40, 46).

[32] Trusted Computing Group (TCG). URL: https://trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-1.83-Part-2-Structures.pdf (cit. on p. 33).

[33] Trusted Computing Group (TCG). URL: https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part3_Commands_pub.pdf (cit. on p. 34).

[34] Trusted Computing Group (TCG). URL: https://trustedcomputinggroup.org/wp-content/uploads/TCG-EK-Credential-Profile-for-TPM-Family-2.0-Level-0-Version-2.6_pub.pdf (cit. on pp. 34, 46, 48).

[35] Cloud Native Computing Foundation. URL: https://github.com/parallaxsecond/rust-tss-esapi (cit. on p. 38).