POLITECNICO DI TORINO

Master degree course in Computer engineering

Master Degree Thesis

Self-Sovereign Identity(SSI) integration in OpenSSL

Supervisor
prof. Antonio Lioy
Andrea Vesco, Ph.D

Candidate
Alessio Claudio

AA 2021-2022

To my family

Summary

Decentralized digital identity is a concept that is rapidly expanding due to new technologies such as Distributed Ledger Technology(DLT) which allows the secure storage of unalterable data. The purpose of this thesis is to implement the Self-Sovereign Identity(SSI) paradigm, a standard proposed by the W3C group of decentralized digital identity, within the most popular and widely used open-source OpenSSL cryptographic library. This thesis is proposed by the LINKS Foundation Cybersecurity team that is working on a larger research project on SSI. The first part of the work was done by studying and analyzing the two major current versions of OpenSSL, obtaining working models that were implemented in software. The next phase was carried out by designing and developing the actual integration of SSI into OpenSSL, side-by-side with a set of APIs and a DID Method to enable communication with IOTA's Tangle. The study concludes with a testing phase of CRUD operations, also evaluating latency times about the type of public keys used.

Acknowledgements

Thanks to the LINKS Foundation Cybersecurity team for their support in times of need.

Contents

Chapter 1

Introduction

1.1 Digital identity

The problem of reliable identity is a problem that humankind has had from ancient times and today, with the advent of digitalization, is even more relevant. In the European priorities 2019-2024 one of the priority areas defined by the European Commission in *"A Europe fit for the digital age"* [1] is *"European Digital Identity"* [2]. First, we must define what is a digital identity.

The digital identity consists of a set of personal information (name, surname, social security number, ...) and generated data during online activities (search history, electronic transactions, social media interactions, ...).

Because of the enormous amount of possible data, it opens possible privacy and security risks like:

- identity fraud.
- identity theft.
- manipulation.

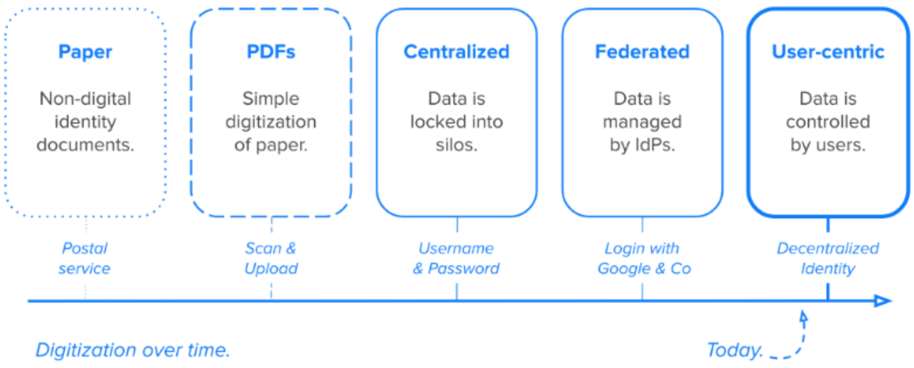


Figure 1.1. Evolution of identity over time). (Source [3])

Before the Internet era, all identity documents were on paper so was natural to translate them into PDF format. This solution doesn't provide any real digital identity, so it was soon replaced by another method, the centralized identity.

1.1.1 Centralized identity

The first iteration of digital identity is the centralized identity, the "one account per service" paradigm [3], where every service requires an account, and each platform maintains the user data. This leads to fragmented digital identities where each user has more than one account that represents his identity.

This form of identity is the simplest but has some major issues:

- **security issue** - each platform handles its way data protection and there is no guarantee that everyone follows the best practice and correct legal regulations, leading often to data loss or data breach;
- **lack of control over data** - the platforms are in charge of providing the identity, creating a situation where the user has no power over his data because, to access a certain service, must rely on that service providing his identity;
- **complex user experience** - the user has to remember various accounts with different methods of authentication

1.1.2 Federated identity

Over time some platforms grew bigger and bigger, gaining more money and technical capability, leading them to effectively control the market. They soon realized how important data and digital identity are, so they tried to overcome some of the limitations of the centralized approach by creating a federated identity.

The biggest service providers take care of users' personal information and provide it to other platforms, offloading them from the burden of managing it. This approach is called Single Sign On (SSO) [4]: the identity provider manages the log in process and the user with one account can access various platforms. This leads to a better user experience but creates some bigger issues:

- **monopoly of few platforms** - the identity providers gains a lot of power being the few that manage the user's identity, creating a monopoly situation;
- **lack of control over data** - same as the centralized approach

1.1.3 Decentralized identity

The future step of digital identity is a decentralized identity: the user gains full control over his data and the problems of centralized and federated approaches are gone. The user maintains a wallet used to store the private keys that replace the passwords and other verified identity details which contribute to proving the identity. [5] This approach is still in the early days and is being developed by the "Decentralized Identity Foundation" and the "Trust Over IP Foundation" and is known as **Self-Sovereign Identity**.

Chapter 2

Background and related work

2.1 Self-Sovereign Identity

SSI is a new concept of decentralized identity where the users create their own digital identity without being dependent on a central authority. Is based on the Decentralized identifier(DID) and the Verifiable Credentials (VC)

2.1.1 Decentralized identifiers

The Decentralized Identifiers (DIDs) are a W3C Recommendation [6] that define a global, unique, verifiable, and decentralized identifier. The DIDs are generated autonomously by the holder of the digital identity and are used to resolve the DID documents. The DIDs are in form of string called URI (Uniform Resource Identifiers) formed by three distinct parts

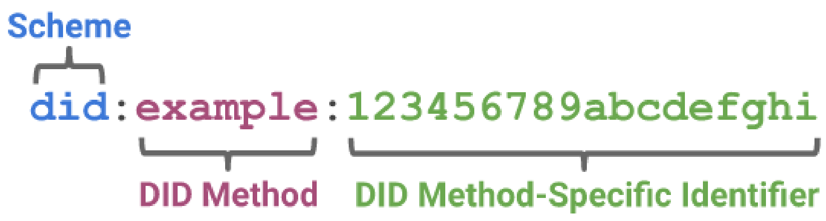

Scheme
did:example:123456789abcdefghi
DID Method
DID Method-Specific Identifier

Figure 2.1. Example of Decentralized identifier (DID). (Source [6])

Scheme

The prefix `did:` is the formal syntax for a generic DID.

DID Method

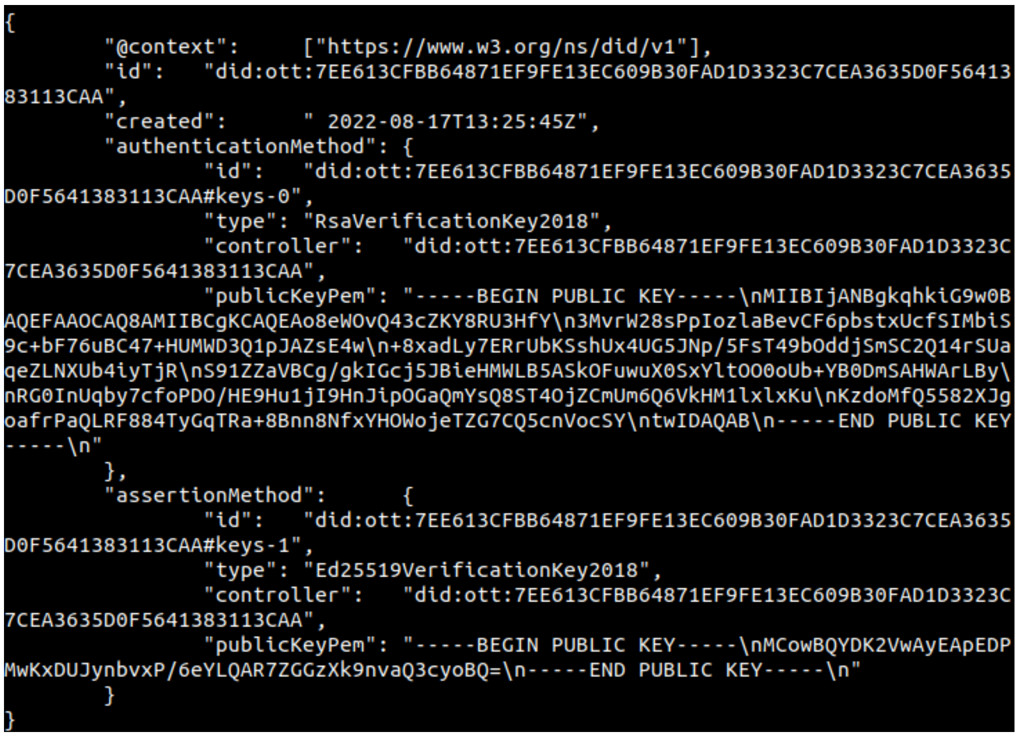
DID Method is the concrete implementation of the system. Usually, a DID Method defines: the Verifiable registry, the underlying technology in which the data is stored, for example, `did:ethr` uses the Ethereum blockchain [7]; the CRUD operation and the format of DID Document.

DID Method-specific identifier

Is a unique identifier inside of that specific DID Method

2.1.2 DID Document

DID Document is a document that contains all the information about the digital identity of the holder, associated with a DID. The usual representation is a JSON file[8] with variable fields because the core proprieties[9] are almost all optional, leaving the developer the option to choose which are useful for their platforms. For our project, we decided to use the following core proprieties.



```
{
        "@context":     ["https://www.w3.org/ns/did/v1"],
        "id":   "did:ott:7EE613CFBB64871EF9FE13EC609B30FAD1D3323C7CEA3635D0F56413
83113CAA",
        "created":      " 2022-08-17T13:25:45Z",
        "authenticationMethod": {
                "id":   "did:ott:7EE613CFBB64871EF9FE13EC609B30FAD1D3323C7CEA3635
D0F5641383113CAA#keys-0",
                "type": "RsaVerificationKey2018",
                "controller":   "did:ott:7EE613CFBB64871EF9FE13EC609B30FAD1D3323C
7CEA3635D0F5641383113CAA",
                "publicKeyPem": "-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0B
AQEFAAOCAQ8AMIIBCgKCAQEAo8eWOvQ43cZKY8RU3HfY\n3MvrW28sPpIozlaBevCF6pbstxUcfSIMbiS
9c+bF76uBC47+HUMWD3Q1pJAZsE4w\n+8xadLy7ERrUbKSshUx4UG5JNp/5FsT49bOddjSmSC2Q14rSUa
qeZLNXUb4iyTjR\nS91ZZaVBCg/gkIGcj5JBieHMWLB5ASkOFuwuX0SxYltOO0oUb+YB0DmSAHWArLBy\
nRG0InUqby7cfoPDO/HE9Hu1jI9HnJipOGaQmYsQ8ST4OjZCmUm6Q6VkHM1lxlxKu\nKzdoMfQ5582XJg
oafrPaQLRF884TyGqTRa+8Bnn8NfxYHOWojeTZG7CQ5cnVocSY\ntwIDAQAB\n-----END PUBLIC KEY
-----\n"
        },
        "assertionMethod":      {
                "id":   "did:ott:7EE613CFBB64871EF9FE13EC609B30FAD1D3323C7CEA3635
D0F5641383113CAA#keys-1",
                "type": "Ed25519VerificationKey2018",
                "controller":   "did:ott:7EE613CFBB64871EF9FE13EC609B30FAD1D3323C
7CEA3635D0F5641383113CAA",
                "publicKeyPem": "-----BEGIN PUBLIC KEY-----\nMCowBQYDK2VwAyEApEDP
MwKxDUJynbvxP/6eYLQAR7ZGGzXk9nvaQ3cyoBQ=\n-----END PUBLIC KEY-----\n"
        }
}
```

Figure 2.2. Example of DID document.

@context

This field is the formal syntax for a generic DID Document and contains the link to the DID W3C Recommendation.

id

This field associates the DID with the DID Document.

created

This field contains a timestamp of the creation of the DID document.

controller

This field indicates which entity is allowed to make changes in the document.

AuthenticationMethod

This field contains the public key of type `type` used during the process of authentication.

AssertionMethod

This field contains the public key of type `type` used during the process of verification of a verifiable credential, owned by the DID subject.

2.2 Trust over IP(ToIP)

Trust over IP[10] project aims to create a scalable, trusted architecture in digital networks which is usually difficult to establish, especially with a lack of regulatory framework. The trust is usually a single relation between an entity A that trusts another entity B. This directional relationship works only in small-scale scenarios but, in bigger communities or digital networks, it becomes difficult to have only direct trust among all entities. A new concept of trust is created, the transitive trust or also known as the trust triangle:

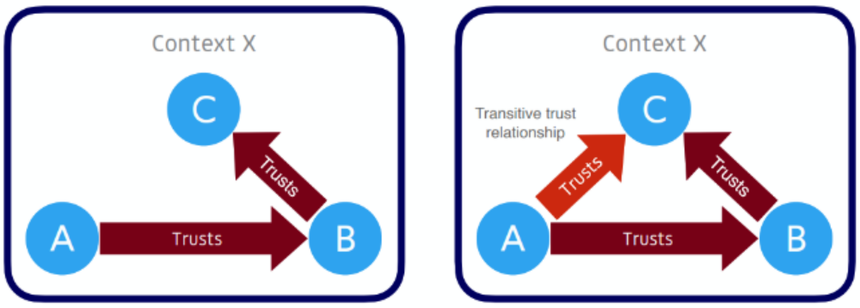


Figure 2.3. The trust triangle. (Source [10])

Entity A trusts entity B which trusts entity C, forming a transitive trust relationship between A and C, with A trusting C because is trusted by B. This model is well applied in the digital world with the problem of trust in digital identity, explained in Chapter1.1. For example, if A a platform and B is an identity provider trusted by A, if a user C successfully authenticates with the system C, receives the trust from B too. ToIP uses this concept of trust with the technological stack provided by Decentralized Identifier and Verifiable Credentials while adding a Governance stack.

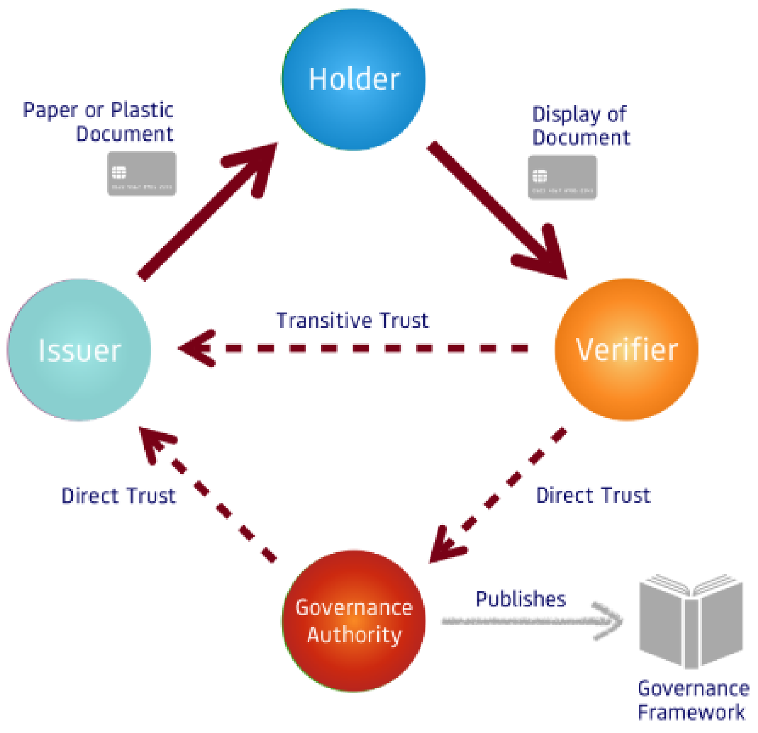


Figure 2.4. The governance trust diamond. (Source [10])

2.2.1 Governance and Technological layers

Trust over IP defines 4 layers among the Governance and Technological stack. The Technological stack aims to implement the architecture while the Governance stack aims to meet the regulatory requirements

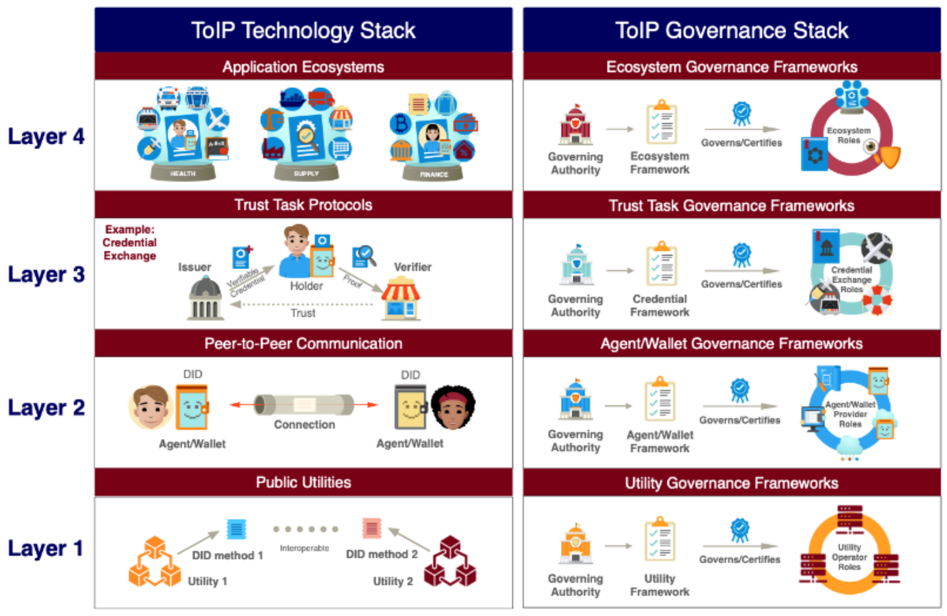


Figure 2.5. The four layers and two halves of the ToIP stack. (Source [10])

Layer 1: Public DID Utilities

Layer 1 is the foundation of the ToIP stack and is based on Distributed Ledger Technologies to create the DID Methods and the needed Governance Framework.

Layer 2: DIDcomm

Layer 2 is about DIDcomm secure messaging standards [11], a secure way to exchange messages between peers. Each Holder maintains a wallet containing his cryptographical material and secure channels are established via DID exchange.

Layer 3: Trust triangle

Layer 3 is the exchange and verification of Verifiable Credential, establishing the triangle of trust between Holder Issuer and Verifier.

Layer 4: Application ecosystem

Layer 3 is the Application ecosystem where the applications interact with humans to provide a service in a trusted way.

2.3 Distributed Ledger Technology (DLT)

As seen in the previous chapter, Layer 1 of the ToIP stack is defined by the foundational technology of the stack namely digital ledger technologies. As seen in the previous chapter, Layer 1 of the ToIP stack is defined by the foundational technology of the stack namely Distributed Ledger Technologies (DLT). DLTs are distributed digital ledgers i.e., the information they contain is replicated at each node and, data entry is done through internal consensus processes. The data entered are immutable and there is no central control coordinating operations or acting as an intermediary, leaving total freedom of access and transactions to users[12]. These features of DLTs are perfect for implementing a DID method.

The project developed by the LINKS Foundation works makes use of Internet of Things(IoT) devices, so DLT's choice to develop a DID Method naturally falls on IOTA.

2.4 IOTA

IOTA [13] is a feeless DLT designed to be lightweight enough to run on IoT devices as well. Since there is no reward for putting a block in the DLT, as IOTA's main purpose is not profit, miners are not present instead each user enters and validates transactions.To remove the figure of the miner, IOTA went beyond the block model of blockchain technology and designed the "Tangle", a mathematical model based on a Directed Acyclic Graph (DAG), where each block validates several blocks preceding it via "probabilistic consensus protocol that enables parallel validation of transactions without requiring total ordering"[14].

The advantage of the Tangle is that it overcomes the bottleneck problem of the blockchain. In the blockchain system, a transaction is validated only if it is placed in a block, and increasing the fees to the miner can increase its priority, creating delays or even making transactions with lower fees unfavorable. In the Tangle, on the other hand, you do not have to reward a miner with higher and higher fees and each user enters their transaction validating others in the process, in a parallel way working better with high transaction loads, which instead slows down the blockchain.

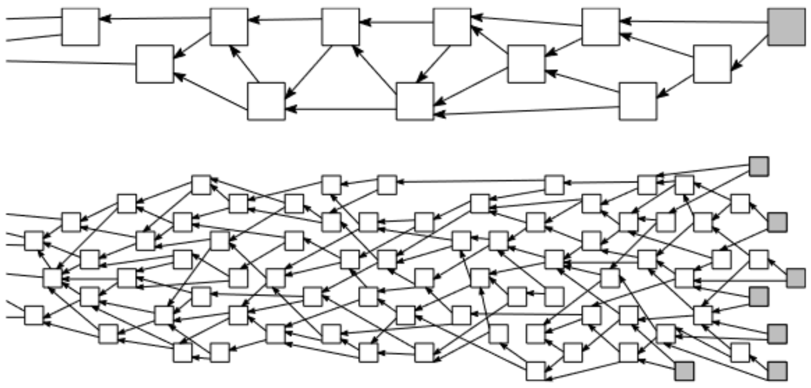


Figure 2.6. The Tangle. (Source [15])

The Tangle however has a problem, it is not entirely decentralized. Currently, there is a node under the control of the IOTA Foundation called "Coordinator" that is responsible for regularly publishing a zero-value transaction as a checkpoint called a "milestone" and definitively validates all functions that directly or indirectly are related to a "milestone" [16]. This sort of semi-centralization has triggered heavy criticism from the community that has not gone unheard. The IOTA Foundation has announced version 2.0 of IOTA that will remove the Coordinator, making the Tangle decentralized[17].

IOTA recently provided an L2 protocol to enable easy interaction of cryptographic messages with the Tangle, called STREAMS [18]. STREAMS is developed in the RUST language and is designed for desktop applications without taking into account the limited resources of IoT devices. To solve this problem, the Cybersecurity team at the LINKS Foundation collaborated on the development of an alternative L2 protocol, WAM.

2.5 WAM

WAM is an L2 cryptographic protocol for communicating with the IOTA Tangle, designed to send a set of logically linked data from a certain index, encrypted and authenticated via Authenticated Encryption with Associated Data (AEAD) [19], from IoT devices with limited computational and memory capacity.

The WAM packet consists of the following fields:

APPDATA and APPDATA_LEN

These fields contain, respectively, the data to be entered on the Tangle and their length. In case the message is too large to send, the data is segmented over multiple packets and linked to each other via the `NEXT_IDX` field, inserting the index of the next IOTA Chrysalis message.

PUBKEY

It contains the public key to be used in signature verification, contained in the `SIGN` field.

NEXT_IDX

Index of the next IOTA Chrysalis message that contains the next message in the chain. The indexes are generated with the following algorithm: from a random source the current seed and the seed of the next message in the chain are generated; with the seeds, the key pairs of the current message and the next message, are generated from the Curve25519 curve; index and next index are generated by making hashes of the public keys, using the hash function BLAKE2b[20].

SIGN

This field contains the signature of the packet digest with the private key corresponding to the public key present between the fields. The digest is computed on the fields `APPDATA_LEN`,`APPDATA`, `PUB_KEY` and `NEXT_IDX`. The verification process is twofold: the reader of the message verifies the integrity of the message by recalculating the hash on the parameters and verifying it thanks to the `SIGN` field; it also calculates the hash of the `PUB_KEY` field and verifies that it is equal to the index of the IOTA message. The combination of these checks ensures the integrity of the WAM packet and comes from the same source.

AUTHSIGN

This field contains a signature computed on the rest of the WAM packet and provides authentication of the packet creator because the private key that generates this signature is protected by a hardware secure element, while the signature of the `SIGN` field is generated by a private key created with each message. The corresponding public key is contained in a public certificate, verified by a trusted Certification Authority.

The tangle loads the data in the clear so, to ensure confidentiality, the entire packet is encrypted with the XSalsa20 cipher using a pre-shared key and a nonce generated from a random source.

2.6 OpenSSL

OpenSSL is an open-source project developed by The OpenSSL Project which aims to create *"a robust, commercial-grade, full-featured toolkit for general-purpose cryptography and secure communication"* [21]. At the time of the writing, there are two Long Term Support (LTS) versions, 1.1.1x and 3.0.x with a significant difference in the architecture.

2.6.1 Version 1.1.1x

In the 1.1.1x version there are four major components:

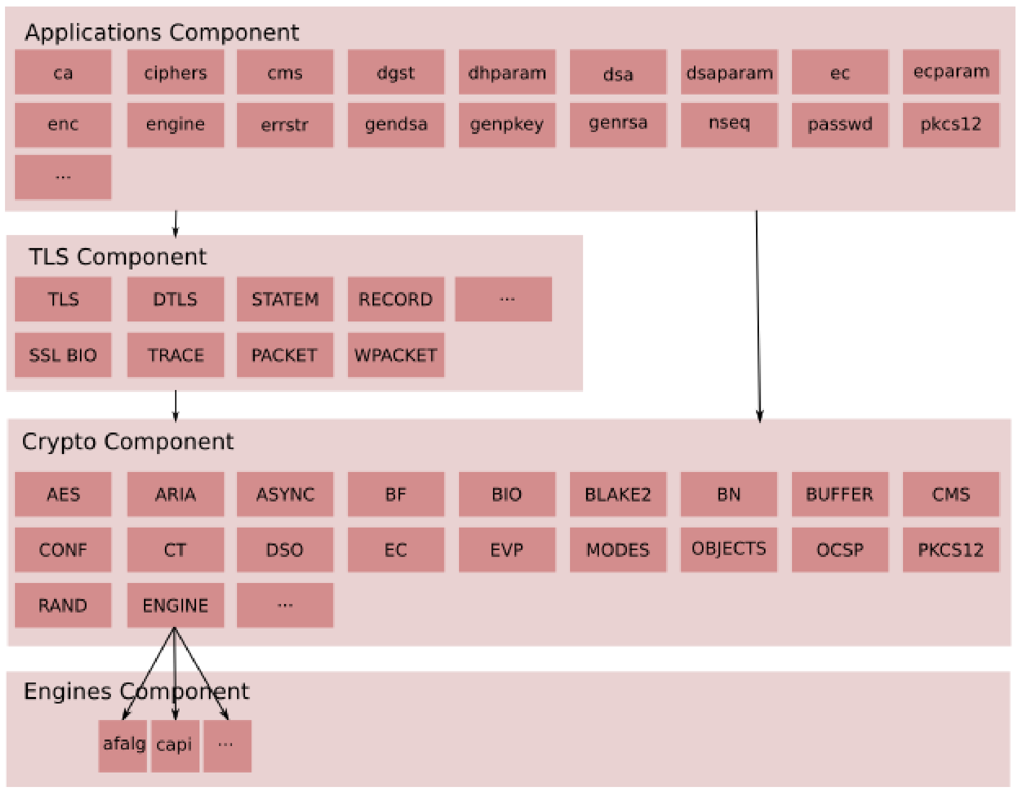


Figure 2.7. The conceptual components in the OpenSSL 1.1.1 architecture. (Source [22])

Application component

This component contains a set of pre-built OpenSSL applications that interact with the users via the command line. The applications implement all the features offered by OpenSSL like symmetric and asymmetric crypto, certificate generation, hashing, etc. using the libssl and libcrypto libraries.

TLS component(libssl)

This component is the implementation of TLS and DTLS protocols using the crypto primitives from libcrypto

Crypto component(libcrypto)

This component is the implementation of the crypto primitives of all supported algorithms which are used by all other components. These two levels of API: the EVP high level interface that splits the functionality from the algorithm implementation(ex `EVP_sign`) and the low-level implementation dependent commands(ex `RSA_sign`).

Engine component

The engine component allows us to extend the functionality of libcrypto. The engines are loadable modules used to provide an alternative implementation of cryptographic algorithms, even if not implemented. By default, OpenSSL contains some Engines but the user can add his own.

2.6.2 Version 3.0.x architecture

Version 3.0 is a major release [23] that changes the architecture of the library significantly. The major changes are:

- **Introduction of providers**;
- **New key concept in the libcrypto library**;
- **The deprecation of low-level APIs and Engines-** low-level APIs were only formally discouraged in the older versions but now are officially deprecated while Engines are replaced by Providers. In this version are still working but in future versions, they will be removed.
- **A new versioning scheme -** the patch level now is indicated by the last number of the version number, while before was a letter at the end of the release version number. For more info see the OpenSSL Migration Guide [24];
- **New cryptographic algorithms and deprecation of old ones -** the use of deprecated cryptographic algorithms is discouraged, but still present for compatibility in the legacy provider.

With the new changes, the architecture receives an important rearrangement.

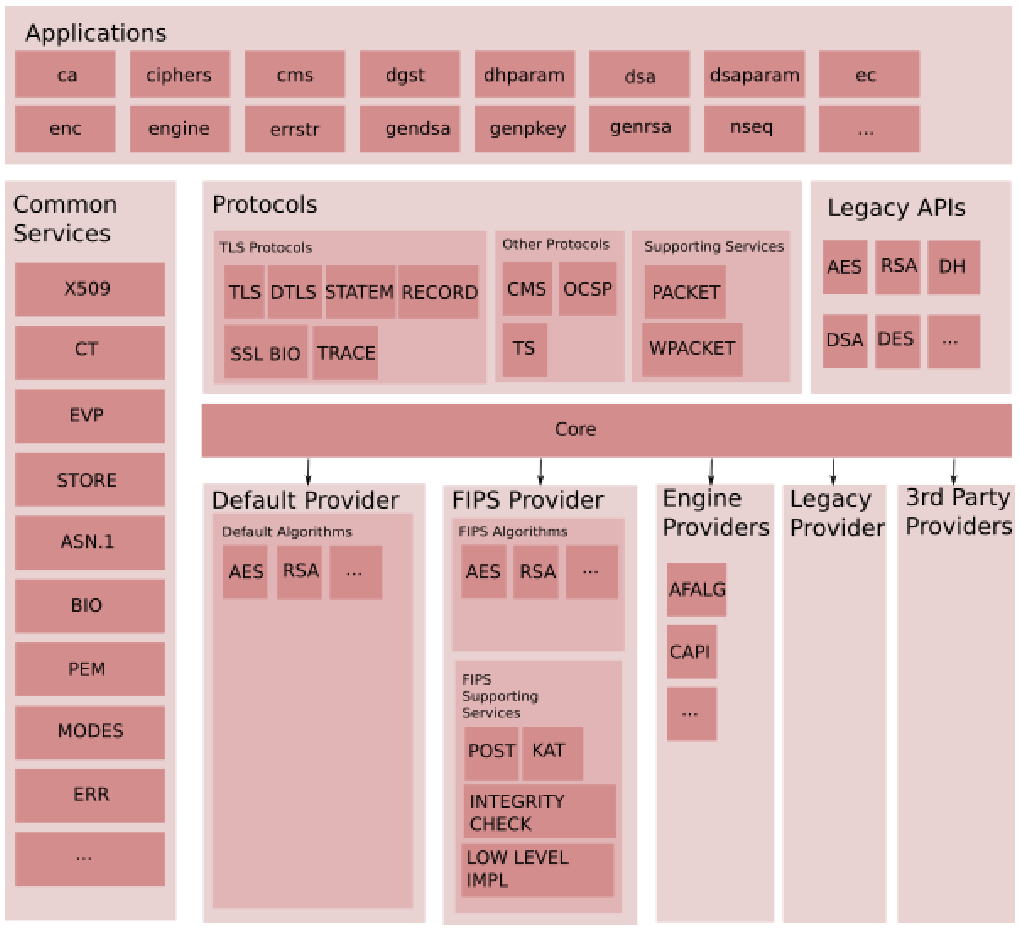


Figure 2.8. The conceptual components in the OpenSSL 3.0.X to-be architecture . (Source [22])

Application

the command line applications, same as version 1.1.1.

Common Services

these are the common blocks used in providers and applications that provide utilities like input and output handling, common format, etc.

Protocols

Is the block that implements the protocols. E.g., TLS, DTLS, OCSP(Online Certificate Status Protocol), TS(Timestamp Protocol)

Legacy APIs

The old low-level APIs are kept for backward compatibility.

Core

This component redirects requests for service to the most appropriate provider that supplies it, following a set of requested properties.

Providers

Providers are the component that collects algorithm implementations. By default, OpenSSL has 5 built-in providers:

- **Default provider** contains all default algorithms and it is loaded if no other provider is requested;
- **FIPS provider** implements the FIPS validated algorithms, algorithms that follow the minimum security requirements defined by the publication 140-2 of the Federal Information Processing Standard (FIPS);
- **Engines Provider** allows the use of old Engines with the new version;
- **Legacy provider** contains the older deprecated algorithms;
- **3rd Party Providers** allow users to develop and load their providers.

2.6.3 Providers and Crypto design

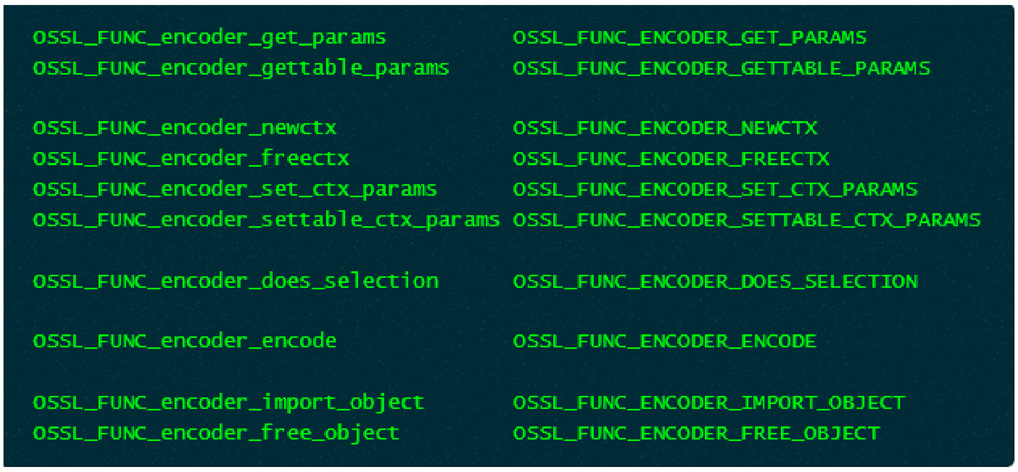
The major redesign of the libcrypto library is changing the focus from algorithms to operations: the algorithms are grouped based on their purpose creating macro categories called operations [25]. Available operations so far:

- **Digests (OSSL_OP_DIGEST)** - a collection of digest algorithms;
- **Symmetric ciphers (OSSL_OP_CIPHER)** - a collection of cipher algorithms;
- **Message Authentication Code (OSSL_OP_MAC)** - a collection of MAC algorithms;
- **Key Derivation Function (OSSL_OP_KDF)** - a collection of KDF algorithms;
- **Random Number Generation (OSSL_OP_RAND)** - a collection of random number generation algorithms and random number sources;
- **Key Management (OSSL_OP_KEYMGMT)** - a collection of functions for creating, holding and managing cryptographic keys;
- **Key Exchange (OSSL_OP_KEYEXCH)** - a collection of key exchange algorithms;

- **Signing and Verification (OSSL_OP_SIGNATURE)** - a collection of signing and verification algorithms;
- **Asymmetric Ciphers (OSSL_OP_ASYM_CIPHER)** - a collection of asymmetric cipher algorithms;
- **Asymmetric Key Encapsulation (OSSL_OP_KEM)** - a collection of asymmetric key encapsulation mechanism algorithms;
- **Encoding and Decoding (OSSL_OP_ENCODER/OSSL_OP_DECODER)** - collection of functions for encoding and decoding a generic function to a specific representation;
- **Store Management (OSSL_OP_STORE)** - collection of functions for loading a generic object from a given URI;

Each operation offers a set of predetermined function templates, with fixed input and output types, to standardize the things to do in that certain function.

There are two types: `OSSL_FUNC_operationname_name_fn` is the function type definition and `OSSL_FUNC_OPERATIONNAME_NAME` is a constant that uniquely identifies in the operation that function.



```
OSSL_FUNC_encoder_get_params           OSSL_FUNC_ENCODER_GET_PARAMS
OSSL_FUNC_encoder_gettable_params      OSSL_FUNC_ENCODER_GETTABLE_PARAMS

OSSL_FUNC_encoder_newctx               OSSL_FUNC_ENCODER_NEWCTX
OSSL_FUNC_encoder_freectx              OSSL_FUNC_ENCODER_FREECTX
OSSL_FUNC_encoder_set_ctx_params       OSSL_FUNC_ENCODER_SET_CTX_PARAMS
OSSL_FUNC_encoder_settable_ctx_params OSSL_FUNC_ENCODER_SETTABLE_CTX_PARAMS

OSSL_FUNC_encoder_does_selection       OSSL_FUNC_ENCODER_DOES_SELECTION

OSSL_FUNC_encoder_encode               OSSL_FUNC_ENCODER_ENCODE

OSSL_FUNC_encoder_import_object        OSSL_FUNC_ENCODER_IMPORT_OBJECT
OSSL_FUNC_encoder_free_object          OSSL_FUNC_ENCODER_FREE_OBJECT
```

Figure 2.9. Example of OSSL_FUNC. (Source [26])

The following image explains the interaction between the core, the provider, and the user application. The usual use case of OpenSSL is a user application that wants to perform some cryptographic operation using one of the supported algorithms.

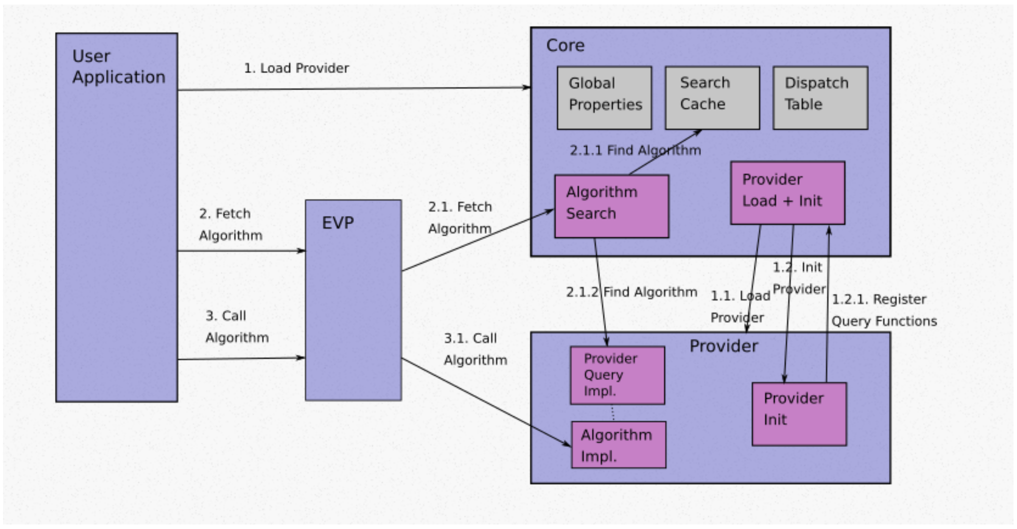


Figure 2.10. The interaction between core and provider. (Source [27])

Load provider

The user calls the `OSSL_PROVIDER_load(ctx, providerid)` or is called the *default provider* implicitly. The core searches in the file system the shared object .so with the corresponding name and, after allocating the correct amount of memory if not the standard provider, calls the provider's entry point to start his initialization. The provider's initialization function is `OSSL_provider_init` and has the task to initialize some provider variables with parameters passed by the structure `OSSL_DISPATCH` to handle the library context and set the dispatch table, a static function that maps the basic functions of the provider with a unique constant, used by the library to correctly identifies the function independently by the name given by the developer.

```
static const OSSL_DISPATCH deflt_dispatch_table[] = {
    { OSSL_FUNC_PROVIDER_TEARDOWN, (void (*)(void))deflt_teardown },
    { OSSL_FUNC_PROVIDER_GETTABLE_PARAMS, (void
        (*)(void))deflt_gettable_params },
    { OSSL_FUNC_PROVIDER_GET_PARAMS, (void (*)(void))deflt_get_params },
    { OSSL_FUNC_PROVIDER_QUERY_OPERATION, (void (*)(void))deflt_query },
    { OSSL_FUNC_PROVIDER_GET_CAPABILITIES,
      (void (*)(void))ossl_prov_get_capabilities },
    { 0, NULL }
};
```

Figure 2.11. Example of dispatch table. (Source [28]).

```
int ossl_default_provider_init(const OSSL_CORE_HANDLE *handle,
                               const OSSL_DISPATCH *in,
                               const OSSL_DISPATCH **out,
                               void **provctx)
{
    OSSL_FUNC_core_get_libctx_fn *c_get_libctx = NULL;
    BIO_METHOD *corebiometh;

    if (!ossl_prov_bio_from_dispatch(in)
            || !ossl_prov_seeding_from_dispatch(in))
        return 0;
    for (; in->function_id != 0; in++) {
        switch (in->function_id) {
        case OSSL_FUNC_CORE_GETTABLE_PARAMS:
            c_gettable_params = OSSL_FUNC_core_gettable_params(in);
            break;
        case OSSL_FUNC_CORE_GET_PARAMS:
            c_get_params = OSSL_FUNC_core_get_params(in);
            break;
        case OSSL_FUNC_CORE_GET_LIBCTX:
            c_get_libctx = OSSL_FUNC_core_get_libctx(in);
            break;
        default:
            /* Just ignore anything we don't understand */
            break;
        }
    }
...
    *out = deflt_dispatch_table;
    ossl_prov_cache_exported_algorithms(deflt_ciphers, exported_ciphers);

    return 1;
}
```

Figure 2.12. Example of provider init function. (Source [28]).

Fetch algorithm

The user application request to use a certain algorithm for a determined operation with some proprieties to the EVP interface. The EVP interface first searches it in the search cache, if previously cached. If not present calls the provider function `PROVIDER_QUERY`. This function receives in input an integer that represents the operation's id and returns, if present the vector `OSSL_ALGORITHM`, a collection of entries in the form of **{Algorithm name, list of proprieties, implementation function vector}**

```
static const OSSL_ALGORITHM deflt_digests[] = {
    /* Our primary name:NIST name[:our older names] */
    { PROV_NAMES_SHA1, "provider=default", ossl_sha1_functions },
    { PROV_NAMES_SHA2_224, "provider=default", ossl_sha224_functions },
    { PROV_NAMES_SHA2_256, "provider=default", ossl_sha256_functions },
...
}

static const OSSL_ALGORITHM *deflt_query(void *provctx, int operation_id,
                                         int *no_cache)
{
    *no_cache = 0;
    switch (operation_id) {
    case OSSL_OP_DIGEST:
        return deflt_digests;
    case OSSL_OP_CIPHER:
        return exported_ciphers;
    case OSSL_OP_MAC:
        return deflt_macs;
  ...
    return NULL;
}
```

Figure 2.13. Example of provider query function and one OSSL_ALGORITHM structure. (Source [28]).

After receiving the `OSSL_ALGORITHM` vector, the EVP interface first searches all rows with the chosen algorithm's name, then choose the row with the best matching proprieties. The user can omit the proprieties, resulting in the default search, or define a set of mandatory and optional properties. In that case, is chosen the row that matches all mandatory properties and has the major number of matches from the optional ones. From this row is taken the implementation function vector that consists of a set of tuples in the form of **{constant unique integer, pointer to function** . The integers are publicly defined by OpenSSL, so the function pointers are correctly identified and used to fill the search cache.

```
const OSSL_DISPATCH ossl_pem_to_der_decoder_functions[] = {
    { OSSL_FUNC_DECODER_NEWCTX, (void (*)(void))pem2der_newctx },
    { OSSL_FUNC_DECODER_FREECTX, (void (*)(void))pem2der_freectx },
    { OSSL_FUNC_DECODER_DECODE, (void (*)(void))pem2der_decode },
    { 0, NULL }
};
```

Figure 2.14. Example of decoder's implementation function vector. (Source [29]).

Call algorithm

The user application calls the requested EVP function, like `EVP_DigestUpdate()` and the function pointer previously fetched, is executed.

Chapter 3

Design and Implementation

In this chapter will be presented the work done for this thesis, from the initial study and design phase to the actual final implementation, and explained some choices made.

3.1 Case study: tpm2-tss-engine

After an initial phase of studying the work and the state of the art, it came naturally to wonder which version of OpenSSL to start with. Currently, there are two LTS versions, the 1.1.1x and the new 3.0.x version. As an initial choice, it was decided to start with version 1.1.1x because, being the longest-lived version, it has code stability, fewer bugs, and is not subject to change.

The initial goal is to analyze how this version would work, and how Self-Sovereign Identity could be integrated. To do this, the first step was to analyze and study an engine because it perfectly shows the interaction between OpenSSL, the engine, and a cryptographic implementation. It was decided to study the engine that allows the use of TPM2.0[30] with the OpenSSL applications, the tpm2-tss-engine[31]. This engine uses the TPM Software Stack (TSS)[32], the middle-level APIs Enhanced System API (ESAPI)[33], to communicate with the TPM. This engine supports:

- **Random number generation hardware**;
- **RSA and ECDSA sign and verification**;
- **TLS secure channel with TPM protected key**.

The engine does not support the creation of keys, but it is provided with an external program, `tpm2tss-genkey` [34], that generates RSA and ECDSA volatile TPM protected keys. To generate persistent keys, the tpm2-tools must be used instead of the program[35].

3.2 Models

In studying the operation of the tpm2-tss-engine, it was evident that its operations, net of specific differences required by the underlying cryptographic implementation, follow a pattern. For each operation, the workflow was studied, and the generic model explaining the pattern was extracted. Each model shows the interaction between the OpenSSL library, the engine, and the back-end implementation and they will be the basis for a better understanding of the library and for defining the integration requirements of Self-Sovereign Identity.

In the following sections, each model is analyzed in detail.

3.2.1 Loading a key

The first model made deals with the loading of a TPM key. The key generation model is not present because in the engine studied, key creation is not present but is done by an external program, and since the models study the interaction between OpenSSL, the engines, and the actual cryptographic implementation, it was chosen not to include it.

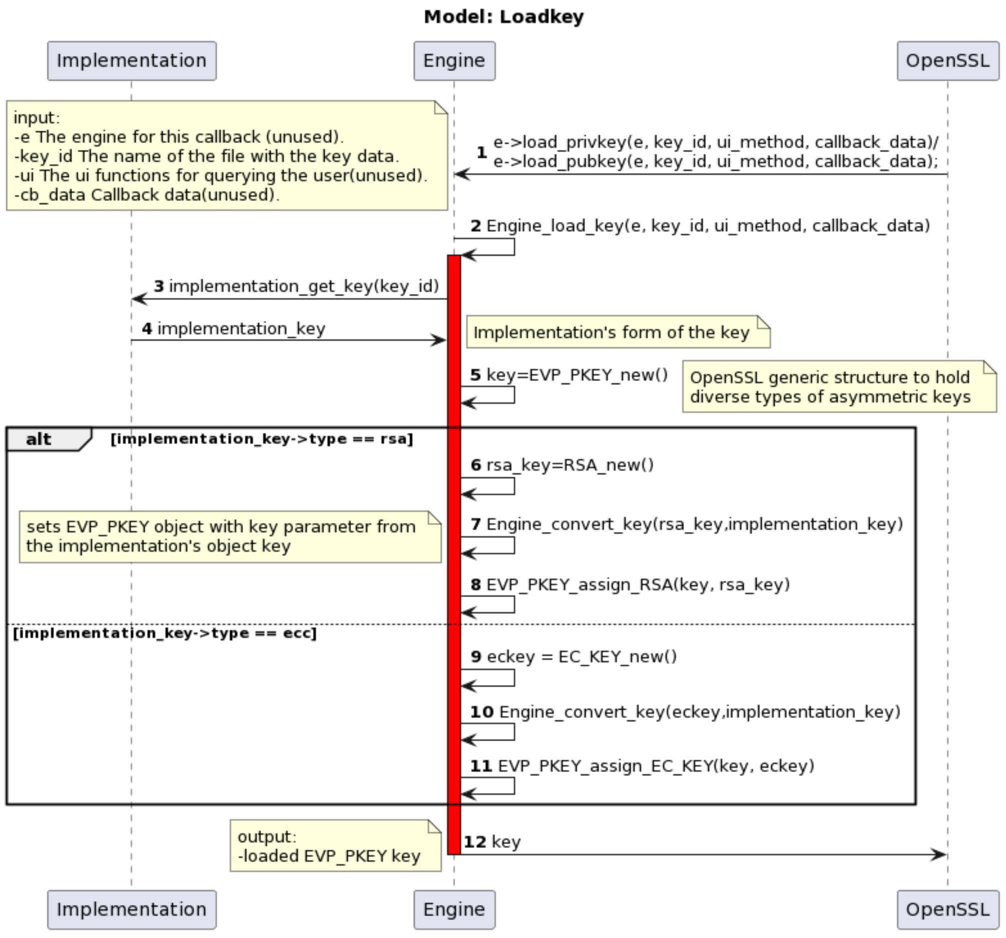

Model: Loadkey
Implementation
Engine
OpenSSL
input:
-e The engine for this callback (unused).
-key_id The name of the file with the key data.
-ui The ui functions for querying the user(unused).
-cb_data Callback data(unused).
1 e->load_privkey(e, key_id, ui_method, callback_data)/ e->load_pubkey(e, key_id, ui_method, callback_data);
2 Engine_load_key(e, key_id, ui_method, callback_data)
3 implementation_get_key(key_id)
4 implementation_key
Implementation's form of the key
5 key=EVP_PKEY_new()
OpenSSL generic structure to hold diverse types of asymmetric keys
alt [implementation_key->type == rsa]
6 rsa_key=RSA_new()
7 Engine_convert_key(rsa_key,implementation_key)
sets EVP_PKEY object with key parameter from the implementation's object key
8 EVP_PKEY_assign_RSA(key, rsa_key)
[implementation_key->type == ecc]
9 eckey = EC_KEY_new()
10 Engine_convert_key(eckey,implementation_key)
11 EVP_PKEY_assign_EC_KEY(key, eckey)
output:
-loaded EVP_PKEY key
12 key

Figure 3.1. OpenSSL1.1.1 model for loading a key

OpenSSL calls the engine's function passing the key id as arguments. The engine uses the id to retrieve the key encoded in the implementation form. After allocating a `EVP_PKEY`, an OpenSSL object that holds various types of asymmetric keys, depending on the key type of the appropriate object is allocated and the key is properly translated by filling this object. After filling the `EVP_PKEY` with the newly generated object, is returned to OpenSSL.

3.2.2 Random number generation

The model of random number generation is very simple. OpenSSL calls the engine's function passing a buffer to fill with a certain number of random bytes. If requested by the implementation, the engine calls the function that seeds the random number generation and then calls the actual implementation's random function that fills the buffer. At the end of the function, the buffer is returned to OpenSSL.

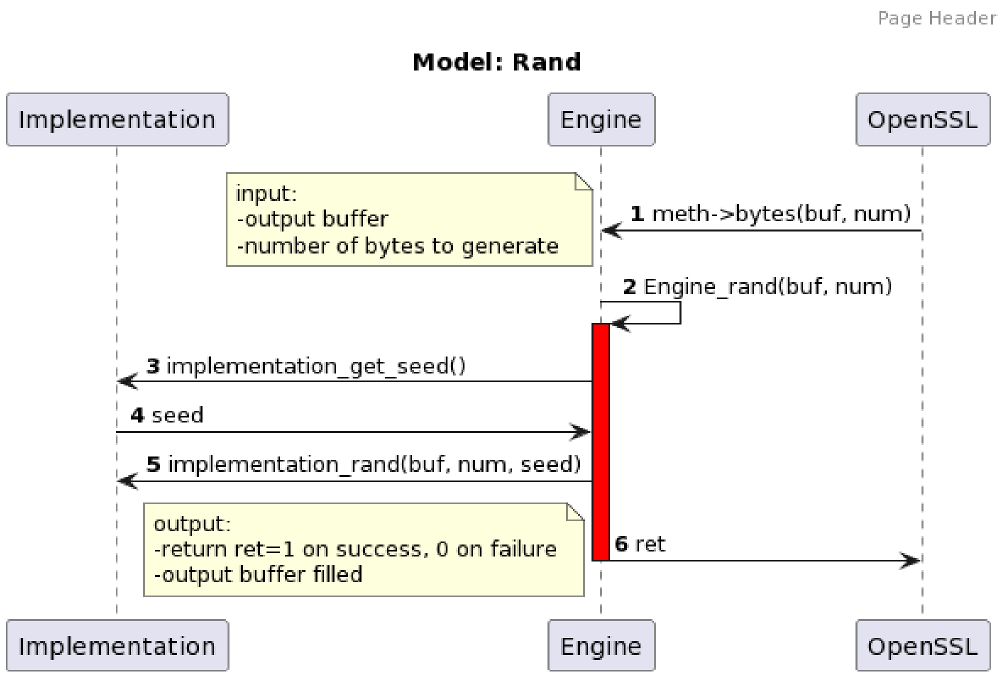


Figure 3.2. OpenSSL1.1.1 model for generating a random number

3.2.3 Signature creation

The signature creation in OpenSSL 1.1.1 is dependent on the algorithm and there are two levels, the high-level EVP calls the low-level algorithm-dependent implementation. In the following models, the low-level implementation is analyzed because is the most meaningful.

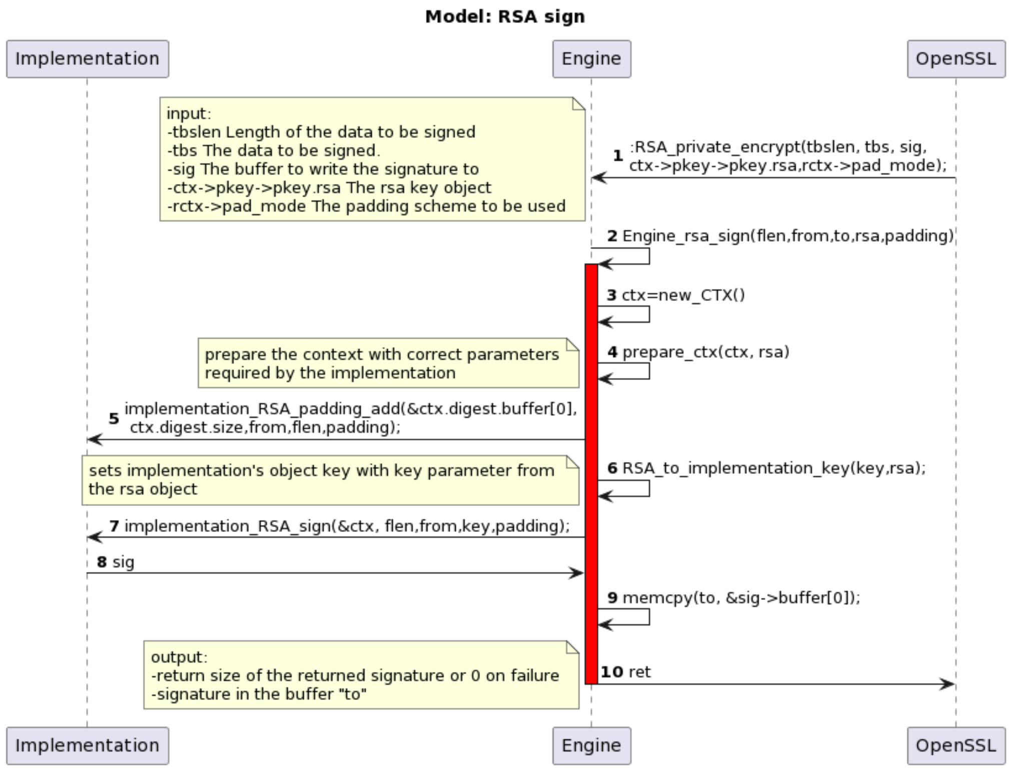

Model: RSA sign
Implementation
Engine
OpenSSL
input:
-tbslen Length of the data to be signed
-tbs The data to be signed.
-sig The buffer to write the signature to
-ctx->pkey->pkey.rsa The rsa key object
-rctx->pad_mode The padding scheme to be used
1 :RSA_private_encrypt(tbslen, tbs, sig, ctx->pkey->pkey.rsa,rctx->pad_mode);
2 Engine_rsa_sign(flen,from,to,rsa,padding)
3 ctx=new_CTX()
4 prepare_ctx(ctx, rsa)
prepare the context with correct parameters required by the implementation
5 implementation_RSA_padding_add(&ctx.digest.buffer[0], ctx.digest.size,from,flen,padding);
6 RSA_to_implementation_key(key,rsa);
sets implementation's object key with key parameter from the rsa object
7 implementation_RSA_sign(&ctx, flen,from,key,padding);
8 sig
9 memcpy(to, &sig->buffer[0]);
output:
-return size of the returned signature or 0 on failure
-signature in the buffer "to"
10 ret

Figure 3.3. OpenSSL1.1.1 model for generating a RSA signature

OpenSSL calls the engine's function passing as arguments the data to be signed, the buffer to fill with the computed signature, the RSA key object holding the private RSA key, and an integer that indicates which padding. The engine sets up the context, an object that holds the data needed by the implementation and fills it. Then calls the implementation function to pad the data, if needed, and finally calls the implementation function that returns the actual signature. After copying the signature in the buffer, returns it to OpenSSL.

The ECDSA signature generation is similar but the OpenSSL functions for ECDSA are different with different input parameters and return values.

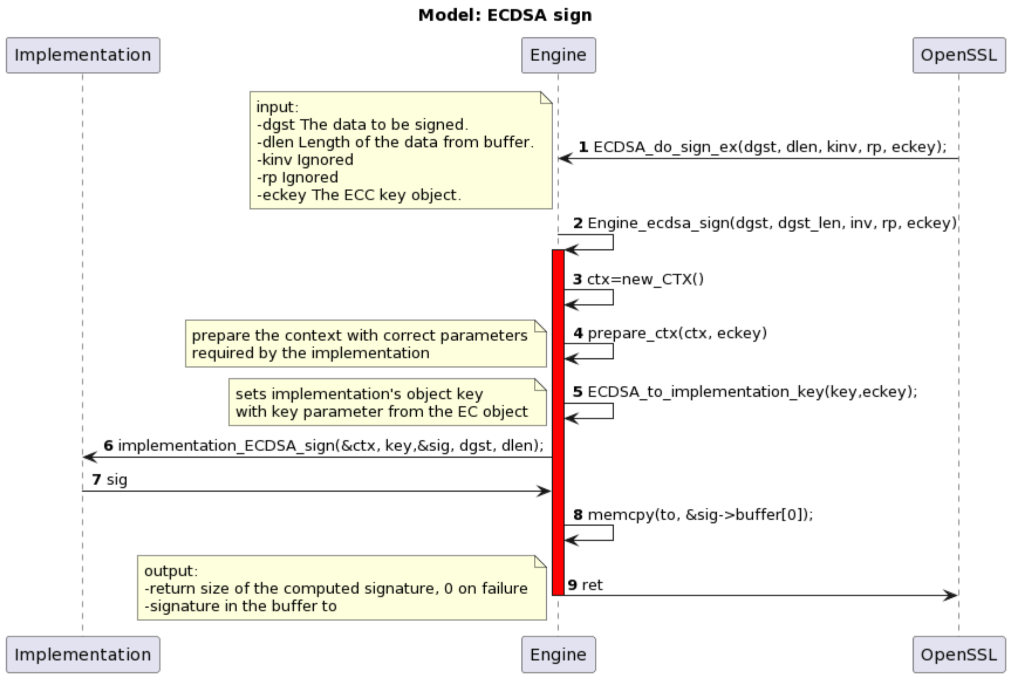

Model: ECDSA sign
Implementation
Engine
OpenSSL
input:
-dgst The data to be signed.
-dlen Length of the data from buffer.
-kinv Ignored
-rp Ignored
-eckey The ECC key object.
1 ECDSA_do_sign_ex(dgst, dlen, kinv, rp, eckey);
2 Engine_ecdsa_sign(dgst, dgst_len, inv, rp, eckey)
3 ctx=new_CTX()
prepare the context with correct parameters required by the implementation
4 prepare_ctx(ctx, eckey)
sets implementation's object key with key parameter from the EC object
5 ECDSA_to_implementation_key(key,eckey);
6 implementation_ECDSA_sign(&ctx, key,&sig, dgst, dlen);
7 sig
8 memcpy(to, &sig->buffer[0]);
output:
-return size of the computed signature, 0 on failure
-signature in the buffer to
9 ret

Figure 3.4. OpenSSL1.1.1 model for generating an ECDSA signature

OpenSSL calls the engine's function passing as arguments the data to be signed, the length of the passing data, and the ECDSA key object holding the private key. The engine set up the context, an object that holds the data needed by the implementation and fills it. The EC object key is translated into the implementation key, by setting up the curve and points. Then calls the implementation function that returns the actual signature and, after copying the signature in the buffer, returns it to OpenSSL the size of the computed signature.

3.2.4 Signature verification

The signature verification in OpenSSL 1.1.1, similarly to the signature creation, is dependent on the algorithm and there are two levels, the high-level EVP calls the low-level algorithm-dependent implementation. In the following models, is analyzed the low-level implementation because is the most meaningful.

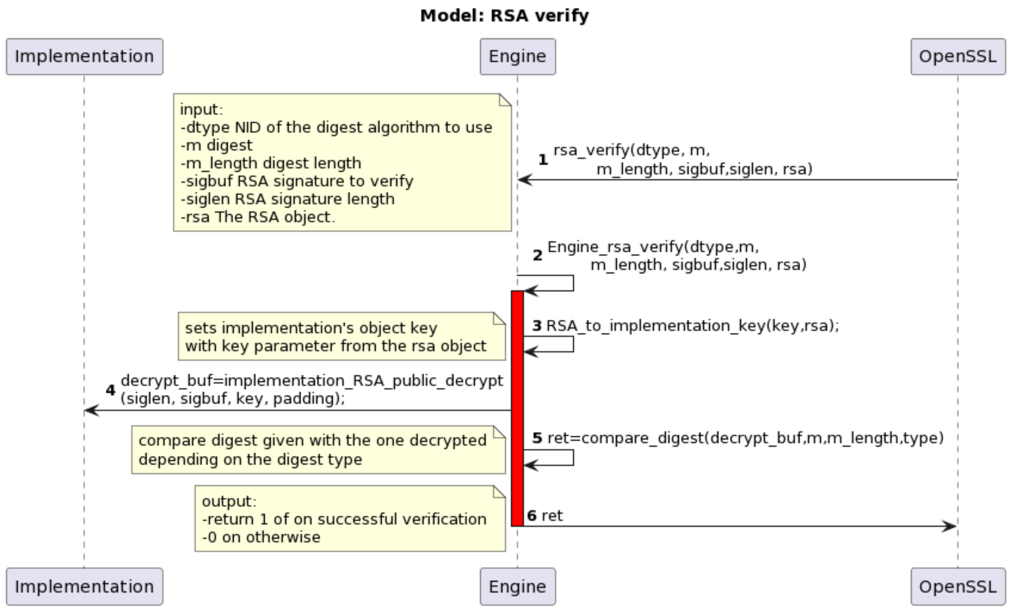

Model: RSA verify
Implementation
Engine
OpenSSL
input:
-dtype NID of the digest algorithm to use
-m digest
-m_length digest length
-sigbuf RSA signature to verify
-siglen RSA signature length
-rsa The RSA object.
1 rsa_verify(dtype, m, m_length, sigbuf,siglen, rsa)
2 Engine_rsa_verify(dtype,m, m_length, sigbuf,siglen, rsa)
sets implementation's object key with key parameter from the rsa object
3 RSA_to_implementation_key(key,rsa);
4 decrypt_buf=implementation_RSA_public_decrypt (siglen, sigbuf, key, padding);
compare digest given with the one decrypted depending on the digest type
5 ret=compare_digest(decrypt_buf,m,m_length,type)
output:
-return 1 of on successful verification
-0 on otherwise
6 ret

Figure 3.5. OpenSSL1.1.1 model for verifying a RSA signature

OpenSSL calls the engine's function passing as arguments: the indicator of which digest algorithm was used on data, the digest itself, the digest length, the signature data to verify, the length of the signature, and the RSA key object holding the public key. The RSA object key is translated into the implementation key form, by setting up the important parameters. Then the engine calls the implementation function that returns the decrypted digest from the signature data. The engine returns to OpenSSL the comparison value between the decrypted data and the passed digest data. If the return value is zero then the signature is correct and the verification is successful.

The ECDSA signature generation is similar but the OpenSSL functions for ECDSA are different with different input parameters.

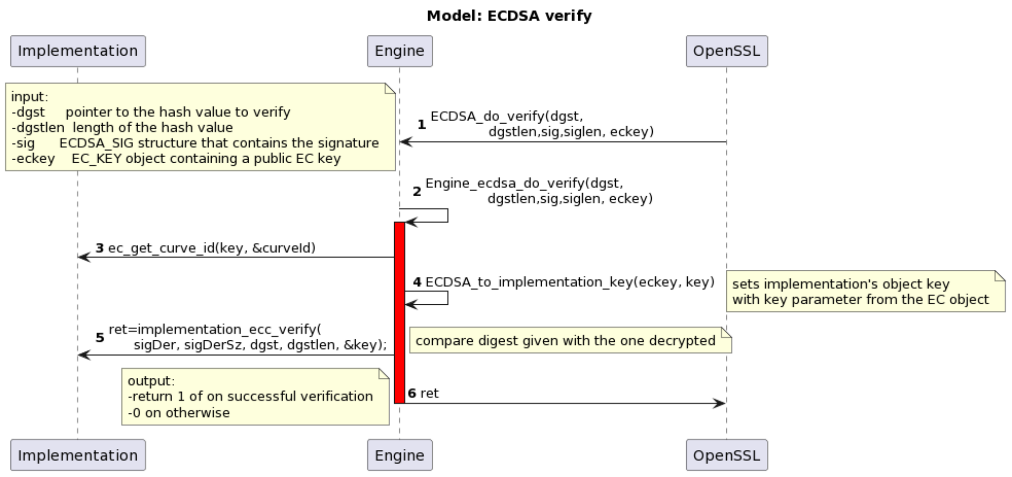

Model: ECDSA verify
Implementation
Engine
OpenSSL
input:
-dgst pointer to the hash value to verify
-dgstlen length of the hash value
-sig ECDSA_SIG structure that contains the signature
-eckey EC_KEY object containing a public EC key
1 ECDSA_do_verify(dgst, dgstlen,sig,siglen, eckey)
2 Engine_ecdsa_do_verify(dgst, dgstlen,sig,siglen, eckey)
3 ec_get_curve_id(key, &curveId)
4 ECDSA_to_implementation_key(eckey, key)
sets implementation's object key with key parameter from the EC object
5 ret=implementation_ecc_verify(sigDer, sigDerSz, dgst, dgstlen, &key);
compare digest given with the one decrypted
output:
-return 1 of on successful verification
-0 on otherwise
6 ret

Figure 3.6. OpenSSL1.1.1 model for verifying an ECDSA signature

OpenSSL calls the engine's function passing as arguments: the digest to verify, the digest length, the signature data to verify, and the EC key object holding the public key. From the implementation is retrieved the information about the curve used and then the EC object key is translated into the implementation key form, by setting up the important parameters. After that the engine calls the implementation function that verifies the passed signature and, if the signature verification is successful, the engine returns 1 to OpenSSL, and 0 otherwise.

3.3 Design of engine from models

After developing the models, the next step was to practice with the engines. Functions from the Mbed-TLS[36] library were chosen as cryptographic primitives because they are easy to use, and written in the C language as OpenSSL. To test the truthfulness of the models, it was decided to implement them. and a mock-up engine was produced with: RSA and ECDSA key loading capability, signing and verification with the same algorithms, and random number generation; More detailed information can be found in the developer manual.

3.4 Upgrade the models to 3.0 design

For the sake of completeness, the new version of OpenSSL was considered and studied as described in the chapters 2.5.2. It has emerged how this version was created with the idea in mind of crypto-agility and code cleanliness by trying to standardize the functions of all the algorithms dealing with the same operation. Individual functions have been divided into smaller functions with specific tasks, and context creation, previously analyzed and recognized as an operation often performed, is now officially standardized and present in most operations by dedicated function.

3.4.1 Loading and creation of a key

The first noticeable thing, compared to the model of the previous version, is its strong modularity because it goes from one macro function to multiple small and specialized functions. In addition, the creation of the key as well as its loading has also been analyzed.

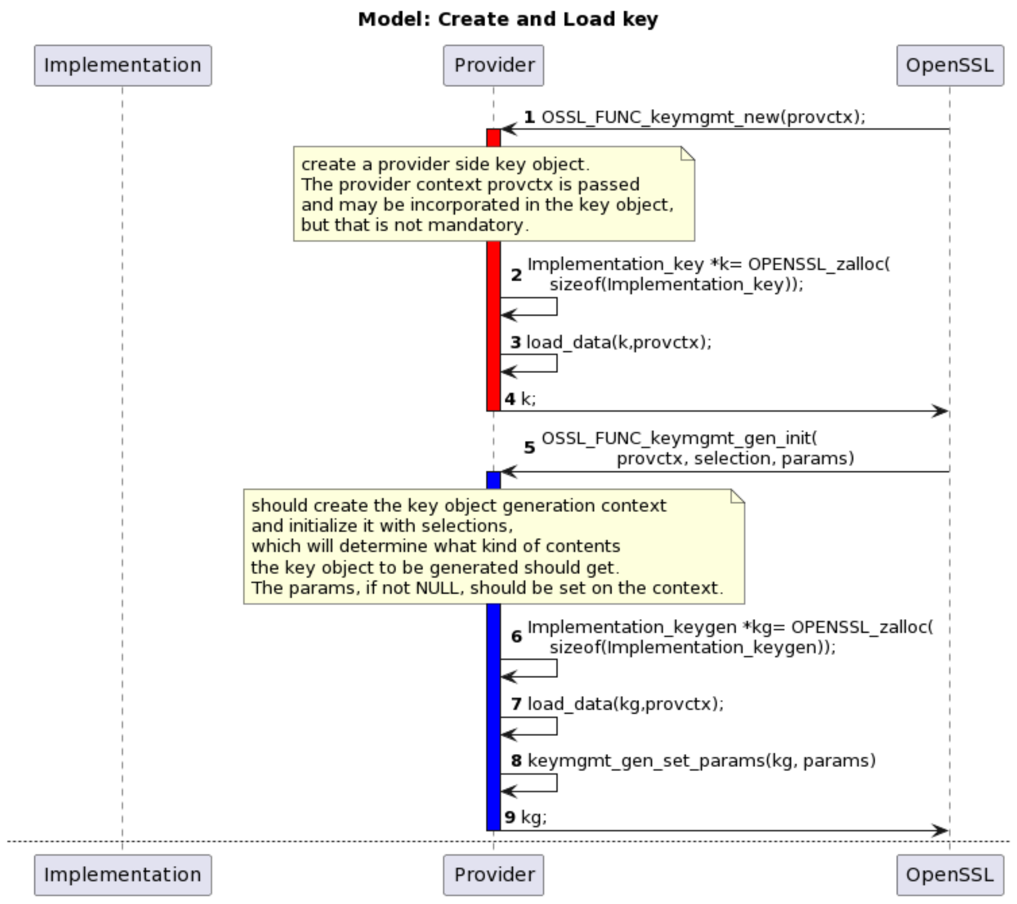

Model: Create and Load key
Implementation
Provider
OpenSSL
1 OSSL_FUNC_keymgmt_new(provctx);
create a provider side key object.
The provider context provctx is passed
and may be incorporated in the key object,
but that is not mandatory.
2 Implementation_key *k= OPENSSL_zalloc(
sizeof(Implementation_key));
3 load_data(k,provctx);
4 k;
5 OSSL_FUNC_keymgmt_gen_init(
provctx, selection, params)
should create the key object generation context
and initialize it with selections,
which will determine what kind of contents
the key object to be generated should get.
The params, if not NULL, should be set on the context.
6 Implementation_keygen *kg= OPENSSL_zalloc(
sizeof(Implementation_keygen));
7 load_data(kg,provctx);
8 keymgmt_gen_set_params(kg, params)
9 kg;
Implementation
Provider
OpenSSL

Figure 3.7. OpenSSL 3.0 model for loading a key 1/2

OSSL_FUNC_keymgmt_new and OSSL_FUNC_keymgmt_free

These functions are concerned with creating and destroying a context, the key object and eventually filling it.

OSSL_FUNC_keymgmt_gen_init

This function takes care of creating a context used during the key generation. Through the `params` vector, some optional generation information can also be specified to be added to the context.

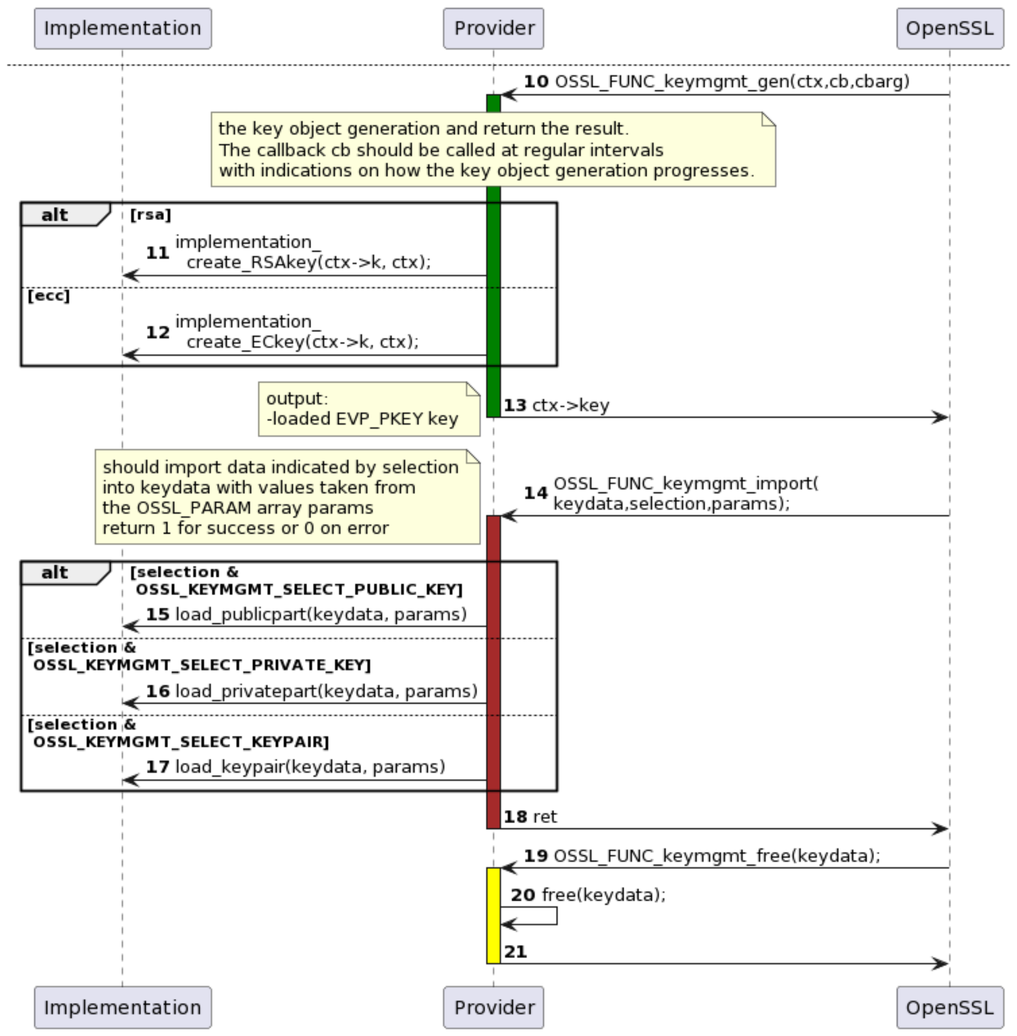

Implementation
Provider
OpenSSL
10 OSSL_FUNC_keymgmt_gen(ctx,cb,cbarg)
the key object generation and return the result.
The callback cb should be called at regular intervals
with indications on how the key object generation progresses.
alt
[rsa]
11 implementation_
create_RSAkey(ctx->k, ctx);
[ecc]
12 implementation_
create_ECkey(ctx->k, ctx);
output:
-loaded EVP_PKEY key
13 ctx->key
should import data indicated by selection
into keydata with values taken from
the OSSL_PARAM array params
return 1 for success or 0 on error
14 OSSL_FUNC_keymgmt_import(
keydata,selection,params);
alt
[selection &
OSSL_KEYMGMT_SELECT_PUBLIC_KEY]
15 load_publicpart(keydata, params)
[selection &
OSSL_KEYMGMT_SELECT_PRIVATE_KEY]
16 load_privatepart(keydata, params)
[selection &
OSSL_KEYMGMT_SELECT_KEYPAIR]
17 load_keypair(keydata, params)
18 ret
19 OSSL_FUNC_keymgmt_free(keydata);
20 free(keydata);
21
Implementation
Provider
OpenSSL

Figure 3.8. OpenSSL 3.0 model for loading a key 2/2

OSSL_FUNC_keymgmt_gen

This function takes care of generating the key by calling the implementation function based on the key type, filling the previously generated context

OSSL_FUNC_keymgmt_import

This function deals with importing a key. Using the `selection` flag, it is possible to select which part of the key to load. The provider calls the appropriate implementation function and returns the result to OpenSSL

3.4.2 Random number generation

This model is very similar to its previous model but a context creation and distraction has been added that can be used during random generation. The major difference is found in the generation function which is designed to support more complex and secure generation. One may define minimum safe bits, additional entropy sources, and a `prediction_resistance` flag to force reseeding from a live entropy source.

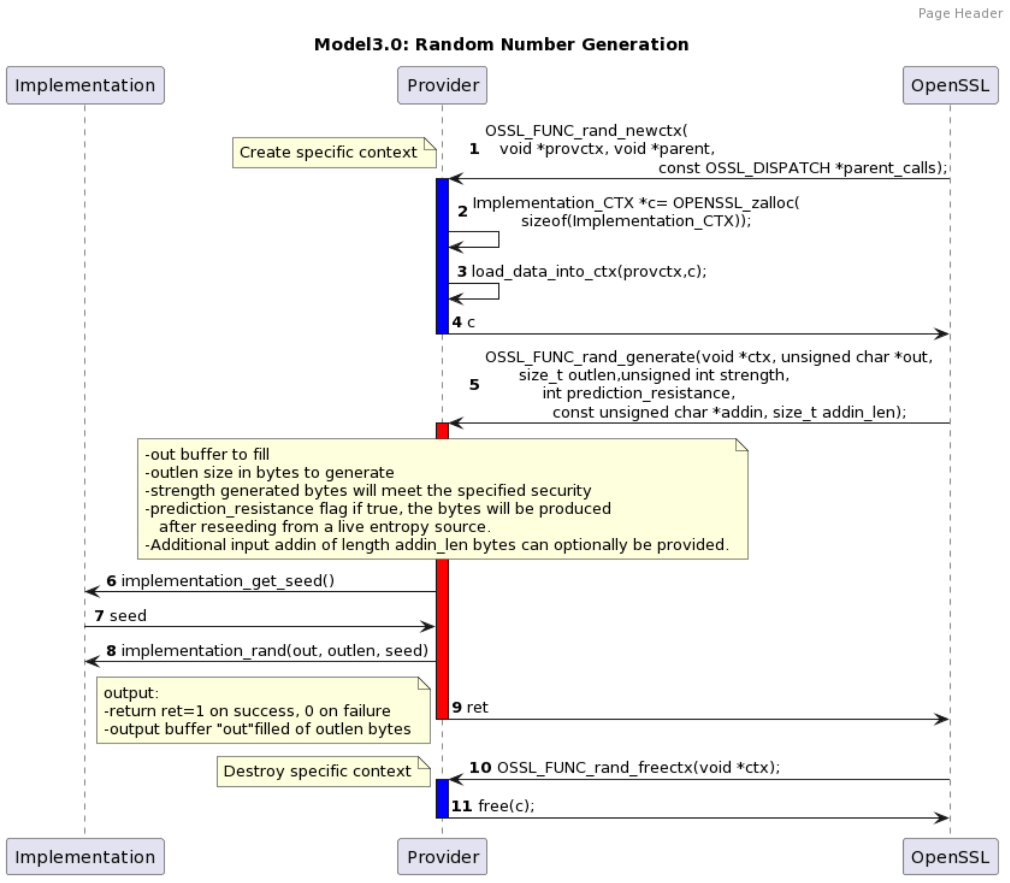

Page Header
Model3.0: Random Number Generation
Implementation
Provider
OpenSSL
Create specific context
1 OSSL_FUNC_rand_newctx(void *provctx, void *parent, const OSSL_DISPATCH *parent_calls);
2 Implementation_CTX *c= OPENSSL_zalloc(sizeof(Implementation_CTX));
3 load_data_into_ctx(provctx,c);
4 c
5 OSSL_FUNC_rand_generate(void *ctx, unsigned char *out, size_t outlen,unsigned int strength, int prediction_resistance, const unsigned char *addin, size_t addin_len);
-out buffer to fill
-outlen size in bytes to generate
-strength generated bytes will meet the specified security
-prediction_resistance flag if true, the bytes will be produced after reseeding from a live entropy source.
-Additional input addin of length addin_len bytes can optionally be provided.
6 implementation_get_seed()
7 seed
8 implementation_rand(out, outlen, seed)
output:
-return ret=1 on success, 0 on failure
-output buffer "out"filled of outlen bytes
9 ret
Destroy specific context
10 OSSL_FUNC_rand_freectx(void *ctx);
11 free(c);
Implementation
Provider
OpenSSL

Figure 3.9. OpenSSL model for generating a random numer

3.4.3 Signature creation and verification

This model has undergone a substantial change from its previous model. Since OpenSSL 3.0 has grouped the algorithms that deal with the same functions into groups called operations, now creation and verification of a signature have the same model, regardless of the algorithm chosen.

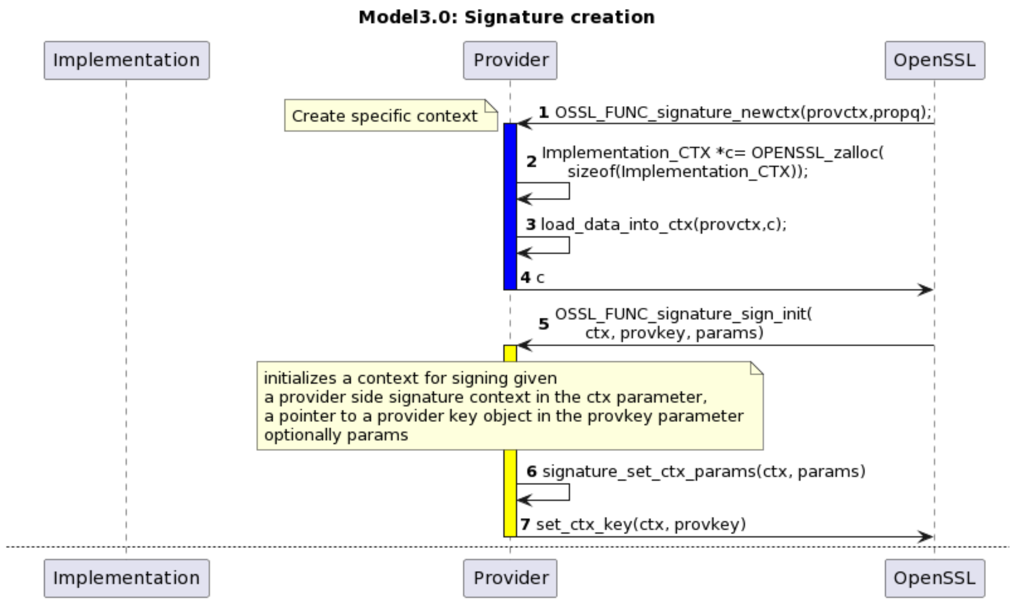

Model3.0: Signature creation
Implementation
Provider
OpenSSL
Create specific context
1 OSSL_FUNC_signature_newctx(provctx,propq);
2 Implementation_CTX *c= OPENSSL_zalloc(
sizeof(Implementation_CTX));
3 load_data_into_ctx(provctx,c);
4 c
5 OSSL_FUNC_signature_sign_init(
ctx, provkey, params)
initializes a context for signing given
a provider side signature context in the ctx parameter,
a pointer to a provider key object in the provkey parameter
optionally params
6 signature_set_ctx_params(ctx, params)
7 set_ctx_key(ctx, provkey)
Implementation
Provider
OpenSSL

Figure 3.10. OpenSSL model for generating a signature 1/2

OSSL_FUNC_signature_newctx and OSSL_FUNC_signature_freectx

These functions are concerned with creating and destroying a provider-side context used by the provider during the signature creation phase.

OSSL_FUNC_signature_sign_init

This function is used for the preparation of a context used during the signing. Through the object passed as parameter `provkey`, the provider sets the key that will be used during the signing. Optionally, through the vector `params` it is possible to set parameters that are needed during the signing phase.

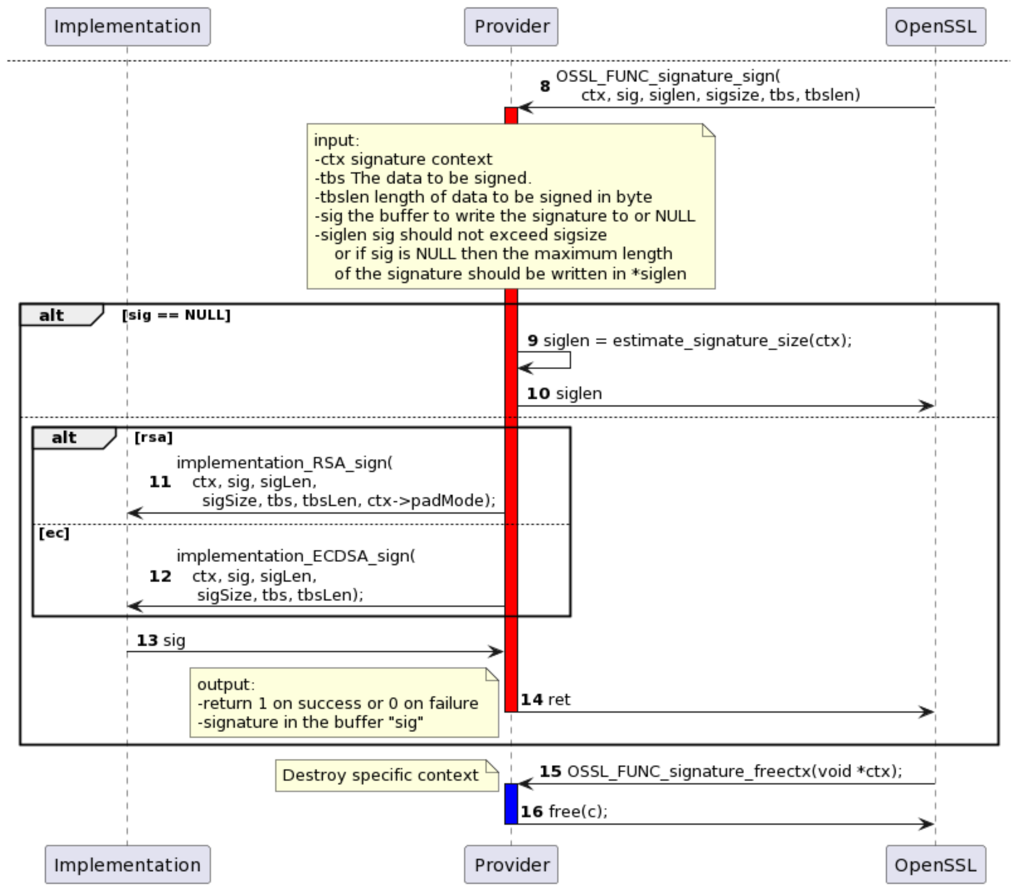

Implementation
Provider
OpenSSL
8 OSSL_FUNC_signature_sign(
ctx, sig, siglen, sigsize, tbs, tbslen)
input:
-ctx signature context
-tbs The data to be signed.
-tbslen length of data to be signed in byte
-sig the buffer to write the signature to or NULL
-siglen sig should not exceed sigsize
or if sig is NULL then the maximum length
of the signature should be written in *siglen
alt [sig == NULL]
9 siglen = estimate_signature_size(ctx);
10 siglen
alt [rsa]
11 implementation_RSA_sign(
ctx, sig, sigLen,
sigSize, tbs, tbsLen, ctx->padMode);
[ec]
12 implementation_ECDSA_sign(
ctx, sig, sigLen,
sigSize, tbs, tbsLen);
13 sig
output:
-return 1 on success or 0 on failure
-signature in the buffer "sig"
14 ret
Destroy specific context
15 OSSL_FUNC_signature_freectx(void *ctx);
16 free(c);
Implementation
Provider
OpenSSL

Figure 3.11. OpenSSL model for generating a signature 2/2

OSSL_FUNC_signature_sign

This function deals with the creation of the signature. As parameters, it receives the previously created context and sets it; the data to be signed and its length in bytes; the buffer to be filled with the signature; a parameter `siglen` which can be an integer indicating the maximum size of the signature or `NULL`. If `sig` is NULL than the provider uses the implementation function to estimate the maximum signature size, writes it in `*siglen` and returns to OpenSSL. If `sig` is not NULL, the provider uses the implementation function to generate the signature, using the correct function based on the key type with its required parameters. It fills the buffer `sig` and returns to OpenSSL 1 or 0 to indicate success or failure, respectively.

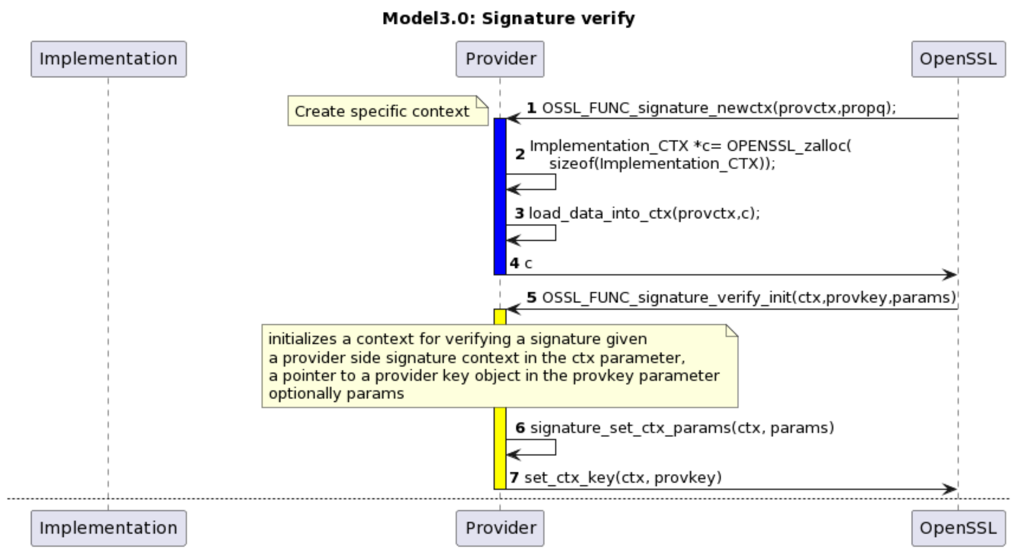

Model3.0: Signature verify
Implementation
Provider
OpenSSL
Create specific context
1 OSSL_FUNC_signature_newctx(provctx,propq);
2 Implementation_CTX *c= OPENSSL_zalloc(sizeof(Implementation_CTX));
3 load_data_into_ctx(provctx,c);
4 c
5 OSSL_FUNC_signature_verify_init(ctx,provkey,params)
initializes a context for verifying a signature given a provider side signature context in the ctx parameter, a pointer to a provider key object in the provkey parameter optionally params
6 signature_set_ctx_params(ctx, params)
7 set_ctx_key(ctx, provkey)

Figure 3.12. OpenSSL3.0 model for verifying a signature 1/2

OSSL_FUNC_signature_newctx and OSSL_FUNC_signature_freectx

These functions are concerned with creating and destroying a provider-side context used by the provider during the signature creation phase.

OSSL_FUNC_signature_verify_init

This function is used for the preparation of a context used during the verification. Through the object passed as parameter `provkey`, the provider sets the key that will be used during the verification. Optionally, through the vector `params` it is possible to set parameters that are needed during the verification phase.

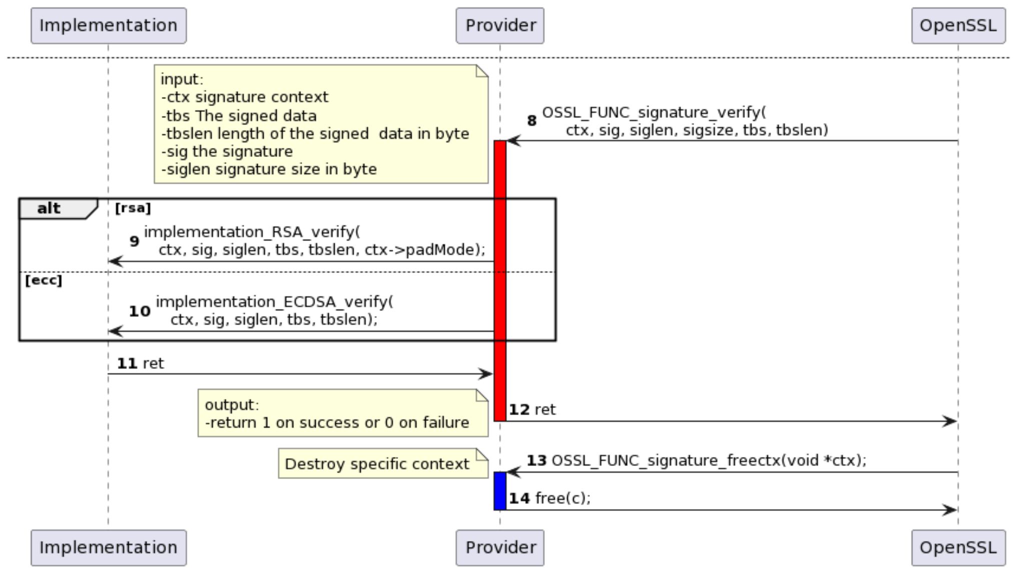

Implementation
Provider
OpenSSL
input:
-ctx signature context
-tbs The signed data
-tbslen length of the signed data in byte
-sig the signature
-siglen signature size in byte
8 OSSL_FUNC_signature_verify(ctx, sig, siglen, sigsize, tbs, tbslen)
alt [rsa]
9 implementation_RSA_verify(ctx, sig, siglen, tbs, tbslen, ctx->padMode);
[ecc]
10 implementation_ECDSA_verify(ctx, sig, siglen, tbs, tbslen);
11 ret
output:
-return 1 on success or 0 on failure
12 ret
Destroy specific context
13 OSSL_FUNC_signature_freectx(void *ctx);
14 free(c);

Figure 3.13. OpenSSL3.0 model for verifying a signature 2/2

OSSL_FUNC_signature_verify

This function takes care of signature verification. As parameters, it receives the previously created context and sets it; the signed data and its length in bytes; the signature to be verified, and its length in bytes. The provider uses the implementation function to generate the signature, using the correct function according to the type of key and the required parameters. The implementation performs the verification and returns the result to the provider. Based on the response, the provider returns to OpenSSL 1 or 0 to indicate success or failure, respectively.

3.5 Design of provider from models

The updated models show how this version aims to have more uniform code, greater simplicity of development, and more detailed parameter handling. Exactly as I did with the templates, I updated the engine by writing the corresponding provider. At the level of functionality, it remained similar but in addition, it has an internal ECDSA and RSA key generation, all again taking advantage of the cryptographic primitives offered by Mbed-TLS[36] library. Another new feature compared to the engine, is the presence of an encoder, a unit of code that is responsible for changing the encoding of an object, for example, a key from its internal representation to PEM encoding. More detailed information can be found in the developer manual.

3.6 DID operation

After analyzing both versions of OpenSSL it was decided that the best version to develop our work was 3.0. Being the latest version there was guaranteed to be more time support, more innovation to the thesis, and in general, the best at the code level highlighted in the previous paragraphs. The first step in integrating Self-Sovereign Identity into OpenSSL 3.0 is to design and develop an operation involving DIDs and a set of supporting APIs.

3.6.1 Design

To define a new operation you must first define a constant indicating the new operation. In *openssl/include/openssl/core_dispatch.h* the operation `OSSL_OP_DID` was added and assigned a unique number. The next step is to define the functions that make up the operation. Through the `#defines` we define the integers that identify the functions. With the `OSSL_CORE_MAKE_FUNC` macro, you define the function template in the form **return type, function name, input parameters**

```
#define OSSL_OP_DID 23
...
#define OSSL_FUNC_DID_CREATE 1
#define OSSL_FUNC_DID_RESOLVE 2
#define OSSL_FUNC_DID_UPDATE 3
#define OSSL_FUNC_DID_REVOKE 4
OSSL_CORE_MAKE_FUNC(void *, did_create, (void *sig1, size_t siglen1,int
    type1,void *sig2, size_t siglen2,int type2))
OSSL_CORE_MAKE_FUNC(int, did_resolve, (char * index, DID_DOCUMENT* did_doc))
OSSL_CORE_MAKE_FUNC(int, did_update, (char * index, void *sig1, size_t
    siglen1,int type1,void *sig2, size_t siglen2,int type2))
OSSL_CORE_MAKE_FUNC(int, did_revoke, (char * index))
```

Figure 3.14. Functions defining the operation DID.

These functions will be implemented by providers and, regardless of the underlying DID Method, should follow the following guidelines:

did_create

This function has to create a DID document and write it into the DID Method, starting with two public keys, authentication, and assertion, of type `type` and length `siglen`. the return value is a `void*` to pass a generic structure, depending on the developer's needs.

did_resolve

This function searches for a DID document from the DID Method, starting with the DID and possibly filling in the empty structure `DID_DOCUMENT`. The return value is an integer indicating the result of the resolve: `DID_OK` the document was found and received correctly; `DID_NOT_FOUD` the document was not found; `DID_REVOKED` the document was found but was revoked; `DID_INTERNAL_ERROR` represents a generic error received during the process of the resolve.

did_update

This function updates a DID document starting with the DID to be updated and the new keys to be embedded in the document in the same form as `did_create`. The return value is an integer indicating the result of the update: `DID_OK` the document was found and modified correctly; `DID_NOT_FOUD` the document was not found; `DID_REVOKED` the document was found but was revoked; `DID_INTERNAL_ERROR` represents a generic error received during the process of the update.

did_revoke

This function revokes a DID document starting with the DID. The return value is an integer indicating the result of the revoke: `DID_OK` the document was found and revoked successfully;`DID_NOT_FOUD` the document was not found; `DID_REVOKED` the document has already been revoked; `DID_INTERNAL_ERROR` represents a generic error received during the revoke process.

Data structures

OpenSSL has two types of structures: internal, which can be used only by OpenSSL, and external, which can also be used by applications by including the correct .h file. This division was created to make the data structures opaque so that, in case of changes due to future updates, applications do not have to be modified. Following this philosophy, there have also been two levels of structures for the DID operation, recognizable by the uppercase name for the external ones and lowercase for the internal ones.

The first structure is `DID_CTX`, a context to be used in conjunction with the API, and the second structure is `DID_DOCUMENT`, which represents a DID document within the application. In *openssl/include/openssl/types.h* correspondence between external and internal structures are defined.

```

typedef struct did_ctx_st DID_CTX;
typedef struct did_document_st DID_DOCUMENT;
```

Figure 3.15. Declaration of external data structures.

Internal structures are defined in *openssl/include/crypto/did.h.*

```

struct did_ctx_st {
    OSSL_LIB_CTX *libctx;
    char *methodtype;
    /* Method associated with this operation */
    OSSL_FUNC_did_create_fn *didprovider_create;
    OSSL_FUNC_did_resolve_fn *didprovider_resolve;
    OSSL_FUNC_did_update_fn *didprovider_update;
    OSSL_FUNC_did_revoke_fn *didprovider_revoke;
    OSSL_PROVIDER *prov;
};

struct did_document_st {
    //authorization methods
    unsigned char * sig1;
    size_t siglen1;
    int type1;
    //assertion methods
    unsigned char * sig2;
    size_t siglen2;
    int type2;
};
```

Figure 3.16. Definition of internal data structures.

did_ctx_st/DID_CTX

This is a context used in conjunction with the API. The fields `*didprovider_create`, `*didprovider_resolve`, `*didprovider_update`, and `*didprovider_revoke` are function pointers that contain the implementations of the provider `*prov`, obtained after the fetch phase. The fields `*libctx` and `methodtype` are currently present only for possible future compatibility, in case of applications that will use different library contexts`OSSL_LIB_CTX` or different DID method implementations in the same provider.

did_document_st/DID_DOCUMENT

This is a context that represents internally to OpenSSL a DID document. The `sig` field is the pointer to a buffer containing a public key of type `type`, length `siglen` in PEM format. The authorization method is defined by variable set 1, while 2 represents the assertion method.

3.6.2 API

The set of supporting APIs, defined in *openssl/include/openssl/did.h* and implemented in *openssl/crypto/did/did_meth.c*, are explained below.

DID_CTX_new and DID_CTX_free

These functions create and release the object `DID_CTX`.

DID_DOCUMENT_new and DID_DOCUMENT_free

These functions create and release the object `DID_DOCUMENT`.

DID_DOCUMENT_set

This function fills an object `DID_DOCUMENT` with the signatures, their type, and their length.

DID_DOCUMENT_set_auth_key and DID_DOCUMENT_set_assertion_key

These functions set the authorization key and assertion key, respectively.

DID_DOCUMENT_get_auth_key and DID_DOCUMENT_get_assertion_key

These functions get the authorization key and assertion key, respectively.

DID_fetch

This function takes care of searching CRUD functions against a certain DID Method chosen by the parameter `*algorithm`. It calls OpenSSL's internal `ossl_provider_query_operation` function, which, by calling the `provider_query` function of the provider previously loaded into the context, returns if present the `OSSL_DISPATCH` vector. If present, it searches all rows for the implementation corresponding to the chosen DID Method and, thanks to the unique function identifiers, fills in the empty function pointers present in the context.

DID_create

This function executes the provider create function. On input it receives the `DID_CTX` object and the `DID_DOCUMENT` object pre-filled. After doing an input check it calls the context function pointer containing the function found during `DID_fetch` and returns the pointer to the newly created DID document, or NULL in case of failure.

DID_resolve

This function executes the provider's resolve function. As input, it receives the `DID_CTX` object, a string containing the DID to be searched for, and the empty `DID_DOCUMENT` object that will be filled if found. After checking the inputs it returns the context function pointer containing the function found during `DID_fetch`.

DID_update

This function executes the provider's update function. As input, it receives the `DID_CTX` object, a string containing the DID to be updated, and the `DID_DOCUMENT` object containing the updated document. After doing an input check it returns the context function pointer containing the function found during `DID_fetch`.

DID_revoke

This function executes the provider's revoke function. As input it receives the object `DID_CTX`, a string containing the DID to be revoked. After doing an input check it returns the context function pointer containing the function found during `DID_fetch`.

To make these functions exportable, in addition to defining them in the public .h file, all of them were also added to the *openssl/util/libcrypto.num* file in this form since all of libcrypto's public functions are in this file.

```
DID_fetch 5558 3_0_4 EXIST::FUNCTION:
DID_CTX_new 5559 3_0_4 EXIST::FUNCTION:
DID_create 5560 3_0_4 EXIST::FUNCTION:
...
```

Figure 3.17. Definition of exportable libcrypto function.

3.7 DID Method: OTT

The DID method chosen to use the integration of Self Sovereign Identity in OpenSSL is the DID method, designed by the LINKS Foundation, **did:ott** which uses as the underlying technology the Distributed Ledger Technology (DLT) provided by Iota, designed as a scalable, lightweight solution perfect for an Internet-of-Things (IoT) ecosystem, and WAM as the communication protocol, again developed by LINKS Foundation from an IoT perspective. As a starting point, a library was provided containing what is needed to write and read from the DLT, through a read function (`WAM_read`) and a write function (`WAM_write`) and an object, `WAM_channel`, containing the data needed for communication. Read and write functions work on the WAM packet and allow it to read and write the `APPDATA` field, then take care of the rest of the protocol previously described in section 2.5. Specifically, the `APPDATA` field is filled with a text string containing the DID Document. Since it is a JSON document and there is no native support in the C language, the open-source cJSON [37] library that takes care of parsing has been added as support. As for the key types the DID method currently supports Ed25519, RSA, and ECDSA Secp256k1 curves keys.

save_channel and load_channel

This function is responsible for saving and loading the `WAM_channel` structure, representing the communication channel with the DLT of a certain DID, created at its creation and used in revoke update operations.

did_ott_create

This function implements the create function of the DID method OTT. The function as input parameters receives an empty `did_new` pointer that will be filled with the string containing the future DID and a `method` structure that acts as a container for the data needed to create the DID method, to be filled in by the caller. It first initializes a new `WAM_channel` by loading the pre-shared key, the IOTA endpoint (dev net in our case), and a context for authentication. From the newly created channel we extract the source index i.e., the DID Method-Specific Identifier, as explained in Figure 2.1. Then the string containing the DID document in JSON form is created from the fields in the `method` structure using the functions offered by cJSON. Finally the function `WAM_write` is called which creates the WAM packet and sends it to the tangle. If the message is successfully sent, the channel is saved by calling the function `save_channel` and the string containing the new DID is created by having it pointed to by `did_new` and the function returns the constant `DID_CREATE_OK`.

did_ott_resolve

This function implements the resolve function of the DID method OTT. The function as input parameters receives an empty structure `did_document` representing the DID document to be filled in case of success for the resolve and a string pointer `*did` containing the DID to be found. It first initializes a new `WAM_channel` by loading the pre-shared key, the IOTA endpoint (dev net in our case). From the DID string, we get the index which is loaded into the channel for reading. Then the function `WAM_read` is called in a while loop, with the loop condition of receiving a valid message. For each iteration, it checks that the next_index, which is the future DID Method-Specific Identifier is not equal to all zeros. In that case, it means that the document was found but was revoked so it returns the constant `DID_RESOLVE_REVOKED`. If it exits the loop it checks that there was at least one document found otherwise it returns the constant `DID_RESOLVE_NOT_FOUND`. If a document was found the text string is parsed and the `did_document` object is re-created, filling the input parameter and returning the constant `DID_RESOLVE_OK`.

did_ott_update

This function implements the update function of the DID OTT method. The function receives as input parameters a pointer `did` containing the DID to be updated and the structure `method` containing the data of the document to be updated. As a first step, the function `load_channel` is called, which takes care of loading the channel allocated upon creation of the DID document. Then, starting from the `method` structure, the DID document is created, in the same manner as the create function. The newly updated document is then passed to the `WAM_write` function, which sends it to the tangle. If it is successfully sent, the new DID is obtained from the channel index and the input parameter `did` is modified, returning the constant `DID_UPDATE_OK`

did_ott_revoke

This function implements the revoke function of the DID OTT method. The function receives as input parameters a pointer `did` that contains the DID to be revoked. As a first step, the function `load_channel` is called, which takes care of loading the channel allocated upon creation of the DID document. The function `WAM_write` is called by setting the revoke flag to true which produces a message with next_index with all zeros, making a revoked DID document immediately recognizable. In case of success, it returns the constant `DID_REVOKE_OK`.

3.8 DID provider

To use the method DID along with OpenSSL's new DID operation, a provider is needed to link the API with the OTT functions. Following the knowledge gained from the models and the test provider, the **didprovider** was developed to provide OpenSSL with the CRUD functions of the DID method OTT.

```
static const OSSL_ALGORITHM didprovider_did[] = {
    {"ETH","provider=didprovider", didprovider_fake_functions},
    {"OTT","provider=didprovider", didprovider_crud_functions},
    { NULL, NULL, NULL }
};
..
static const OSSL_ALGORITHM* didprovider_query_operation(void* provCtx, int
    id,int* no_cache)
{
    *no_cache = 0;
    printf("DID QUERY\n");
    switch (id) {
        case OSSL_OP_DID:
            return didprovider_did;
        break;
    }
    return NULL;
}
```

Figure 3.18. Provider query of didprovider.

The function of `provider_query` provides the vector of algorithms `OSSL_ALGORITHM` in case the DID operation is requested. This vector contains two algorithms, one that mimics the presence of a DID Method `ETH` formed only by empty functions for testing purposes, and the `OTT` that implements the DID Method OTT functions. The function vector `OSSL_DISPATCH` contains tuples of functions and unique identifiers. Below is a brief explanation of the functions.

```
const OSSL_DISPATCH didprovider_crud_functions[] = {
    {OSSL_FUNC_DID_CREATE, (void(*)(void))didprovider_create},
    {OSSL_FUNC_DID_RESOLVE, (void(*)(void))didprovider_resolve},
    {OSSL_FUNC_DID_UPDATE, (void(*)(void))didprovider_update},
    {OSSL_FUNC_DID_REVOKE, (void(*)(void))didprovider_revoke},
    { 0, NULL }
};
```

Figure 3.19. Provider query of didprovider.

didprovider_create

This function receives as input the two public keys of the DID document, their length and type, and fills the `method` structure. It allocates an empty string of DID length and calls the `did_ott_create` method function by passing it the structure and string. If successful, this function returns the string of the new DID or NULL in case of error.

didprovider_resolve

This function receives as input the DID to be found and the empty OpenSSL structure `DID_DOCUMENT`. The function allocates an empty structure `did_document` and passes it to the DID Method function `did_ott_resolve`. If the resolve returns `DID_RESOLVE_ERROR`, `DID_RESOLVE_REVOKED` or

DID_RESOLVE_NOT_FOUND this function returns DID_INTERNAL_ERROR, DID_REVOKED or DID_NOT_FOUND, respectively. If the resolve returns successfully, the supporting API function DID_DOCUMENT_set is called to fill in the DID_DOCUMENT structure, using the data contained in the did_document structure. Finally, the provider function returns success via the constant DID_OK.

didprovider_update

This function receives as input the DID to be updated and the two public keys of the updated DID document, their length, and type. It fills the method structure and calls the did_ott_update method function by passing it the structure and the DID. If successful, the function returns the constant DID_OK or DID_INTERNAL_ERROR if unsuccessful.

didprovider_revoke

This function receives as input the DID to be revoked. It calls the method function did_oct_revoke passing it the structure and the DID. If successful, the function returns the constant DID_OK or DID_INTERNAL_ERROR if unsuccessful.

Chapter 4

Validation and results

This chapter describes how the work was validated and the results obtained. A case of a user developing his application to take advantage of the features offered by OpenSSL in the DID domain was simulated as an application scenario. The user is interested in: creating a DID document, checking that the creation was successful, making a change to that document, checking that the change was successful, performing a revocation of his DID document, and finally checking that the revocation was successful. A demo application has been developed that performs the previously described operations, specifically:

- **Key creation phase** - For completeness, this phase is added, although it is not directly present in the demo application, which expects public key files in PEM format, as it is not important in the OpenSSL perspective because the library is designed to be modular so it must have no internal dependencies and work out-of-the-box with any key in the right format. To create keys, one can take advantage of the applications offered by OpenSSL. Specifically, if you want to create software public keys you can use the command `openssl pkeyutl` [38] and generate the key pair and then use the algorithm-specific command, for example, `openssl rsa` [39] with the -pubout parameter in the case of RSA keys. If, on the other hand, you are interested in non-volatile key pairs protected by TPM2.0, you can take advantage of the combination of the *tpm2provider* provider and the *tpm2-tools*, creating a persistent key bound to a handle and then using the algorithm-specific command, for example, `openssl rsa` [39] with the -pubout parameter specifying -provider tpm2 and -handle [handle_id]. An example is present among the tests provided on the tpm2-provider repository [40].
- **Loading phase** - Through the command `OSSL_PROVIDER_load` the providers *default* and *didprovider* are loaded, which are responsible for loading keys and performing DID operations, respectively. The empty DID document is then instantiated via the `DID_DOCUMENT_new` command and the *didprovider* functions are loaded via the `DID_fetch` command. Finally, the keys are loaded into the application using the functions offered by OpenSSL and the `DID_DOCUMENT` structure is then filled using the `DID_DOCUMENT_set` command.
- **Execution phase** - The DID document is added to the DLT by the command `DID_create` followed by a subsequent `DID_resolve`. As a further check, the two public keys contained in the received document are extracted by the commands `DID_DOCUMENT_get_auth_key` and `DID_DOCUMENT_get_assertion_key`. An updated DID document is then created with different keys and loaded with the command `DID_update` followed by a resolve to check and extract the keys, the same way as the create function. Finally, the updated document is revoked thanks to the `DID_revoke` and checked via the resolve.

4.1 Testbed

The demo application was tested on two different systems. The first is a Raspberry Pi 4 model B [41] with Raspberry Pi OS installed via the provided software *Raspberry Pi Imager* and a virtual machine on which is installed Ubuntu 20. After several tests, it quickly became apparent how the two platforms have very similar latency on CRUD operations because the wait time is not given by the processing power of the CPU but by the time to wait for processing by the Iota node. The first test was to measure operations regardless of what type of public keys are used. Each operation is measured individually, with a particular interest in the resolve operation. In one iteration of the demo application is measured one create function, one update function, one revoke function, and three resolve functions, for a total of 24 iterations.

Operation	**Mean**	**Median**
Create	14.6	10.1
Resolve	0.6	0.6
Update	14	8.4
Revoke	2.6	2

Table 4.1. Mean and median latency of the CRUD operations, in seconds.

With the collected data, the mean, i.e., the sum of the data divided by its number, and the median, i.e., the middle value or the mean of the middle two in the case of even element of numbers, were calculated.

It is evident from the results that the slowest operations are Create and Update. Resolve is the fastest and Revoke is quite fast. The results are consistent but it is interesting to go and look at the time distributions in detail to try to better understand the individual functions.

4.2 Create

The Create function has very scattered times, with noticeable peaks as long as 30 or 40 seconds. This may be caused by the write function on the DLT in that after sending a message, it remains in a queue until it is validated by other messages, adding a random component to the timings. From the distributions, however, we can see that in 24 iterations 50% of the timings are within 10 seconds, in line with the median, but specifically, we can record in the range of 5 to 10 seconds. This time is perfectly acceptable because the Create function is called only once when the DID document is created.

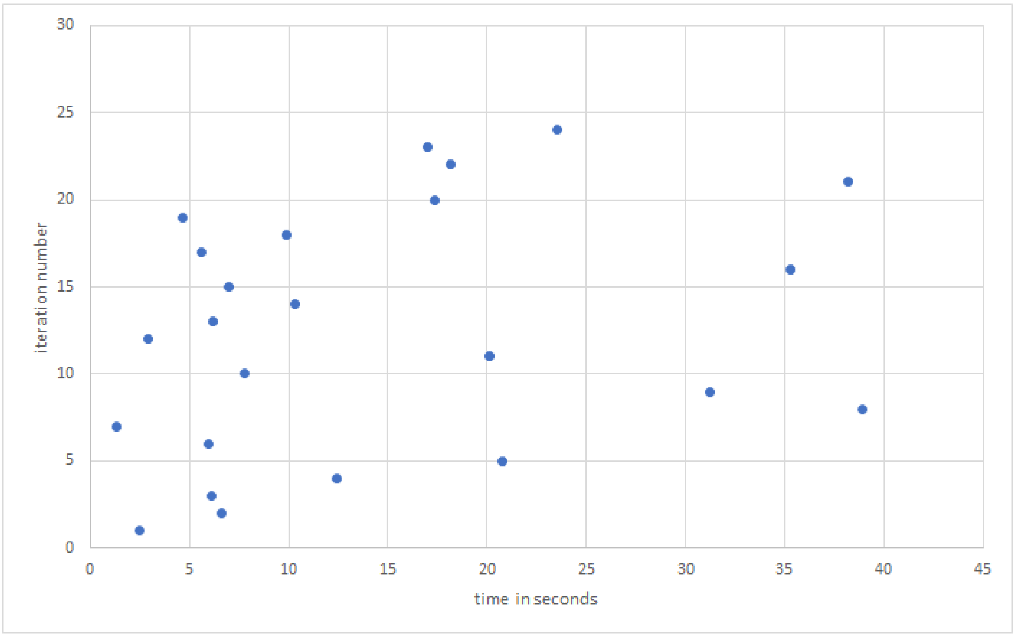


Figure 4.1. Scatter plot of the latency times of the create function

4.3 Resolve

Resolve is the most important function because it is used more times in DID communication than any other function, often in contexts where low latencies are important. From the mean and median we can see that it is the fastest function and it is easy to imagine why: resolve only makes a read request from the DLT and does not send data, reducing both network latency and waiting time because there is no need for message validation and IOTA's internal algorithms are well optimized. From the latency graph, there is an increasing trend in the number of iterations. In the first twenty iterations, it stays around 0.4s while from 20 to 60 iterations it rises around 0.6s and from 60 onwards it tends to rise around 0.8s. A possible explanation can be found in IOTA's *Congestion Control Algorithm*[42], which adjusts the traffic according to the load and the number of requests because, having generated 72 requests to resolve to the node from the same IP address in a short time, it could be that the node lowers the priority to requests sent by the demo application. In any case, hardly more than 20 resolutions are executed in a short time, so bring the average request to 0.4 seconds, an acceptable time for the resolve function.

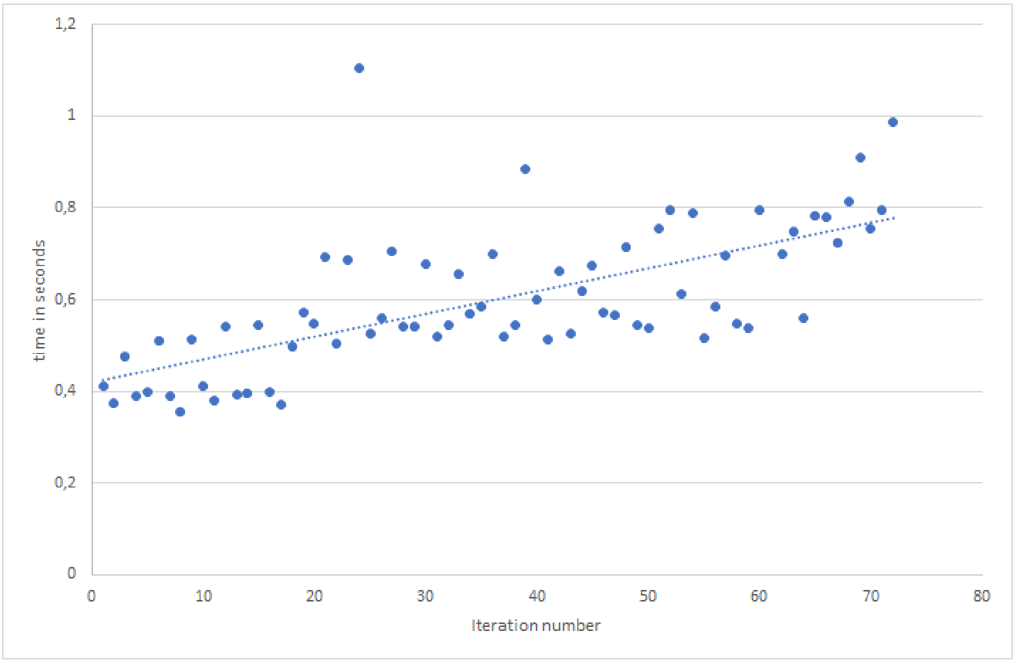


Figure 4.2. Scatter plot of the latency times of the resolve function

4.4 Update

The update function has a similar situation to the create function, with long and variable times due to a write in the DLT. The latencies are also similar ranging between 5 and 10 seconds in 50% of cases, with sporadic very long spikes. However, the time is acceptable because updating a DID document is a process that is done very few times and never in situations where latency matters.

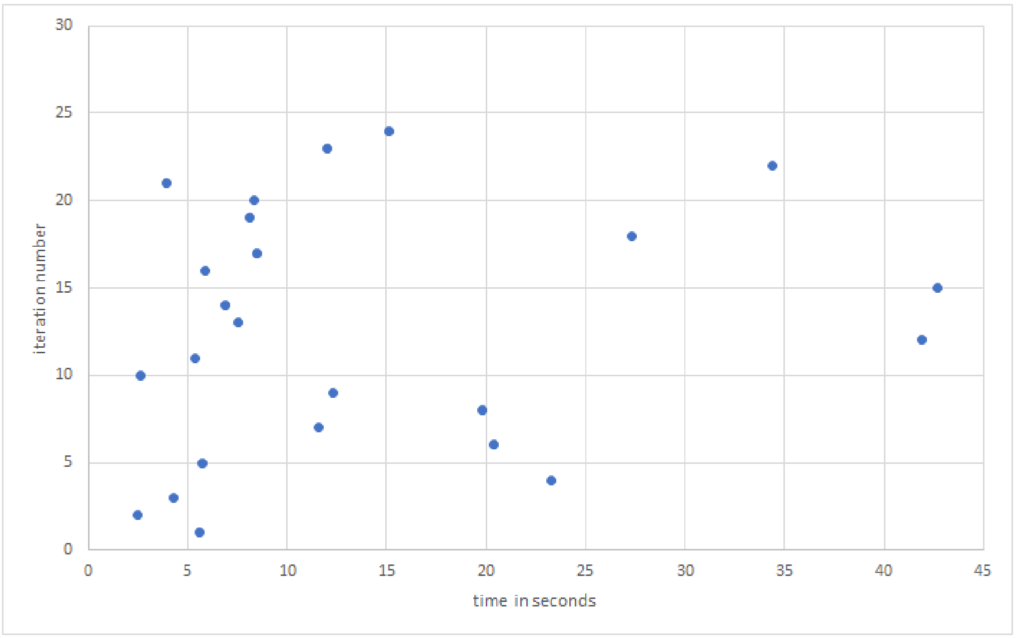


Figure 4.3. Scatter plot of the latency times of the update function

4.5 Revoke

The revoke function is a special case: in the same way, as create and update it does a write on the DLT via the write function, but at the same time most iterations take less than two seconds. This is due to a very small message sent to the DLT. This is a time well above what is needed for a one-time revocation.

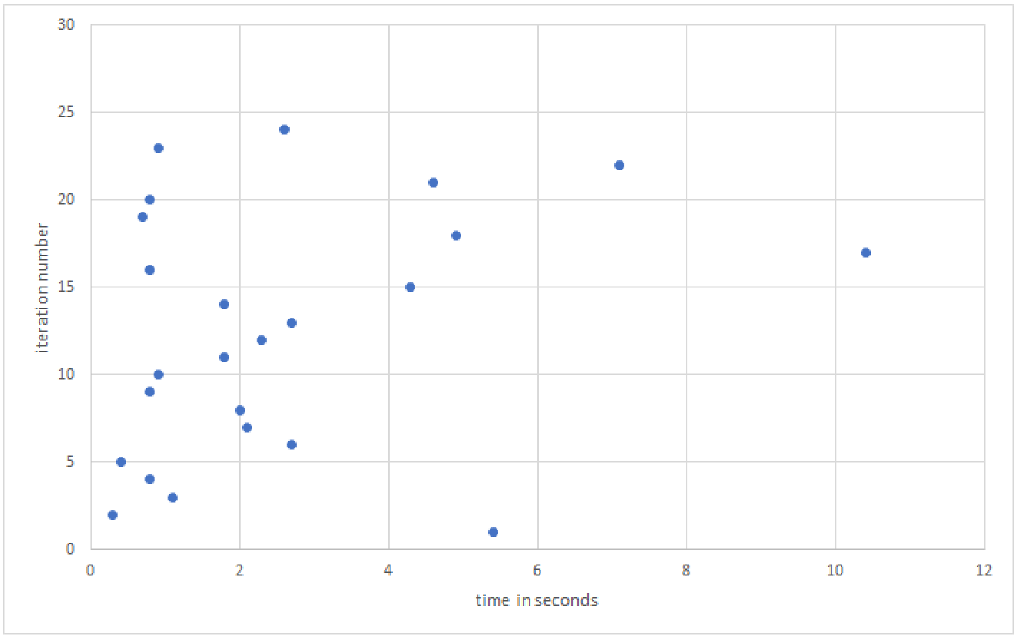


Figure 4.4. Scatter plot of the latency times of the revoke function

4.6 Key type

The revoke latency shows how the duration of functions may also be related to the size of the DID document. It comes naturally to consider how the latency of CRUD functions changes as the size of the DID document changes, and as it stands, the only variable parts are the public keys. I first evaluate how the size in bytes of the DID document changes if it is created only with the same type of key among the supported ones, i.e. Ed25519, RSA, ECDSA Secp256k1 and compare it with the average byte size of the first test, since the keys were changed between the create function and the update function .

Mixed type	**Ed25519**	**RSA 1024**	**RSA 2048**	**RSA 4096**	**ECDSA**
1189	946	1262	1626	2334	1092

Table 4.2. Size in bytes of the message sent to the DLT by key type.

It is evident from the data that keys of type Ed25519 and type RSA 4096bit are the keys that produce the most different documents concerning the size. Therefore, the latencies generated by documents containing only Ed25519 and RSA 4096bit keys were analyzed. From the size, it is immediately clear that the performance of ECDSA keys will be in line with that of Ed25519 keys and RSA 1024bit and 2048bit keys in line with the results of the first test.

Twenty-five iterations of the demo application were run according to key type, then peaks were removed from the data, and the mean and median were calculated.

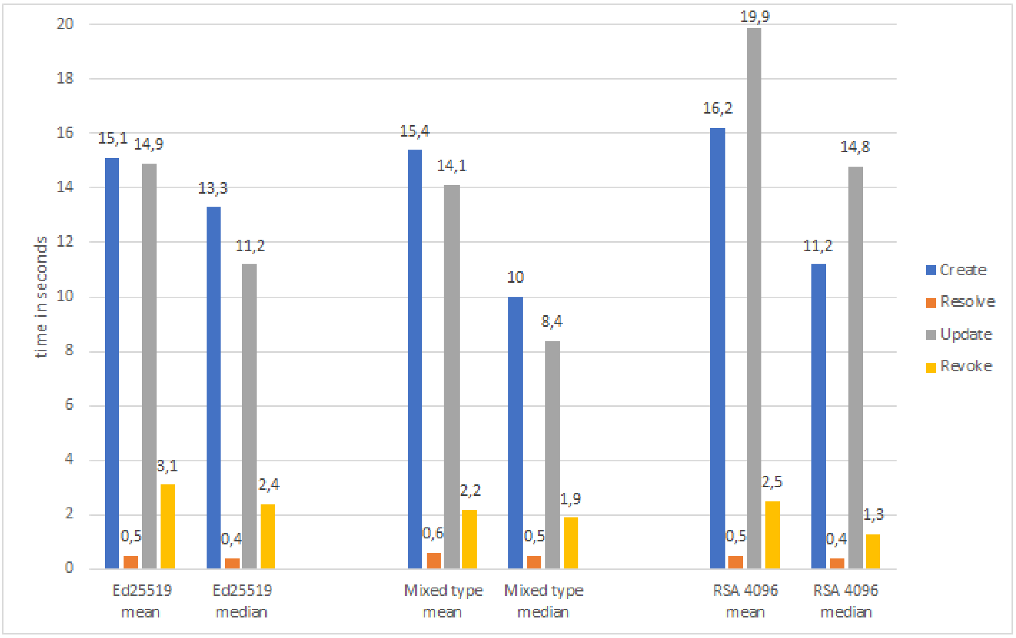


Figure 4.5. Comparison of the latency times based on the key type

The starting hypothesis was that as document size increased, average latency increased. This, however, occurred only partially. The data show that the test with mixed keys, document size 1189 bytes, is on average slightly faster than in the case with RSA 4096bit keys, with document size 2334 bytes, except for the case of revoke which is slower. In contrast, the test with Ed25519 keys that produces the smallest document, of 946 bytes, has no less latency and is slightly slower than in the case of mixed keys. This difference can be attributed to the fact that nodes can be loaded and take longer, regardless of the message size, also the difference in DID document size in the 3 cases is small.

From these analyses, it can be seen that there is not much difference in using different key types, increasing the document size may have small increases in latencies but does not change the order of magnitude of the times. Net of random spikes, the create and update functions on average have between 5s and 15s latency, the revoke function under 3s, and the resolve function remains stable between 0.4s and 0.s. All of these times are acceptable for the needs of DID documents and do not highlight preferred key types, leaving the choice of type up to the user based on his or her needs.

Chapter 5

Conclusion and future developments

This thesis project is part of a larger research project on Self-Sovereign Identity carried out by the LINKS Foundation that aims at the development and integration of SSI into modern commonly used technologies. The work was based on the desire to integrate SSI within the most popular and widely used open-source cryptographic library OpenSSL.

To achieve that goal, the work started with a phase of the initial study of the problem and the state-of-the-art in this field. Currently, there are two Long-term support versions and it was decided to start from the 1.1.1 version of OpenSSL which led to the development of minimal models of operation and applied in the development of a test engine. The study then focused on the new version, 3.0, with its operation and new features. The results were satisfactory for our project needs so it was decided to update the models with the changes and consequently write a provider from those models.

After gaining familiarity with the last version of the library, the final part of the thesis then focused on the actual design of the integration of SSI into OpenSSL through the creation of a new operation to manage DIDs, supporting APIs, and a provider. In addition, a DID Method was developed using IOTA's tangle as the architecture. The thesis project concluded with a final testing phase, carried out through a proof of concept that combined and validated the DID method, provider, operation, and API. The results showed that the latency times of the functions are acceptable for our designated use and that there is no public key type that guarantees lower latency times.

Possible future developments of this thesis are many. The models developed in the study phase were used to familiarize with the engines and providers but provide important information on how they should be developed in the specific case of RNG, signature creation, and verification, so in case a new software or hardware cryptographic implementation like a new secure element will be easy to adapt and develop an engine/provider based on the models. Another possible future development will be the integration of the DID operation with another project in the current development, namely the TLS connection using DID documents instead of x509 certificates. The APIs will also need to be reviewed and adapted to the future needs required by that project. Regarding the DID method, it will be necessary to update it when IOTA 2.0 is released.

Appendix A

User Manuals

This section explains how to install and how use the software.

A.1 tpm2-tss-engine

This engine, developed by the community *Linux TPM2 & TSS2 Software* [31], allows the functionality of TPM2.0 to be used with OpenSSL 1.1.1x.

A.1.1 Requirements and installation

The requirements and installation guide can be found in the *INSTALL.md* file in the official repository [43].

A.1.2 Use

To use the engine there are two methods: within your developed code or at the command line.

In your code, this command should be added before any cryptographic operations

```
ENGINE *e = ENGINE_by_id("tpm2tss");
if(!e) exit(EXIT_FAILURE);
```

and this command should be added at the end to free the object.

```
ENGINE_free(e);
```

To use it at the command line just add `-engine tpm2tss` to the supported OpenSSL commands. The list of supported OpenSSL applications can be found in the Github description [31].

A.2 linksengine

Linksengine is an engine developed for OpenSSL 1.1.1x as a test of the models studied. It uses the cryptographic primitives of Mbed-TLS and implements: RSA and ECDSA key loading; random number generation; signing and verification of RSA-PKCS1.5 and ECDSA algorithms (SECP256R1, SECP521R1, SECP192K1 curves). Being a test program to get familiar with the engines, it should not be considered a complete example of an engine implementing all available library functions and applications.

A.2.1 Requirements

The following dependencies must be installed before installing linksengine.

OpenSSL 1.1.1x

OpenSSL 1.1.x can be found in the OpenSSL's official site[44].

Mbed-TLS 2.28

The engine is based on Mbed-TLS's cryptographic primitives and must be installed following the official instructions [36] or, if available in your distro, using this command

```
sudo apt install libmbedtls-dev
```

A.2.2 Installation

After correctly editing the installation path and include path in the Makefile according to your system, run the following commands

```
sudo make
sudo make install
make tests
```

A.2.3 Use

To test this engine, a set of tests is given and they are called simply by calling the chosen test [TEST_NAME] with this line command

```
./test/[TEST_NAME].sh
```

[TEST_NAME] is the name of one of the following tests:

- **ecdsa_sign_failtest** - This test creates a random digest, signs it through the engine using ECDSA as the algorithm, makes a change to the digest, and verifies that the signature check fails both with the engine and without.
- **ecdsa_sign_prime256v1** - This test creates a random digest, signs it via the engine using ECDSA curve prime256v1 as the algorithm, and verifies that the signature verification is correct both with the engine and without.
- **ecdsa_sign_secp192k1** - This test creates a random digest, signs it via the engine using ECDSA curve secp192k1 as the algorithm, and verifies that the signature verification is correct both with the engine and without
- **ecdsa_sign_secp521r1** - This test creates a random digest, signs it via the engine using ECDSA curve secp521r1 as the algorithm, and verifies that the signature verification is correct both with the engine and without.
- **random** - This test generates 300 bytes of random text using the engine.
- **rsa_sign** - This test creates a random digest, creates an RSA random key pair, signs the digest both through the engine and without, using the RSA algorithm with pkcs1.5 padding, and verifies that the two signature files are identical.

- **rsa_sign_failtest** - This test creates a random digest, creates an RSA random key pair, signs the digest through the engine using the RSA algorithm with pkcs1.5 padding. It then makes a modification to the digest and creates a signature with the same key on the modified digest without the engine and verifies that the two signature files are different.
- **rsa_sign_withpad** - This test creates a random digest smaller than the block size, creates an RSA random key pair, signs the digest both through the engine and without, using RSA algorithm with pkcs1.5 padding, and verifies that the two signature files are identical.
- **rsa_verify_eng** - This test creates a random digest, creates an RSA random key pair and divides the public and private key, signs the digest with the engine using the RSA algorithm with pkcs1.5 padding. Then, it performs signature verification with the engine checking that it is successful.
- **rsa_verify_eng_sign_no_eng** - This test creates a random digest, creates an RSA random key pair and divides the public and private key, signs the digest without the engine using the RSA algorithm with pkcs1.5 padding. Then, it performs signature verification with the engine checking that it is successful.
- **rsa_verify_failtest** - This test creates a random digest, creates an RSA random key pair and divides the public and private key, signs the digest with the engine using the RSA algorithm with pkcs1.5 padding. Then, modifies the digest and performs signature verification with the engine checking that fails.
- **rsa_verify_no_eng** - This test creates a random digest, creates an RSA random key pair and divides the public and private key, signs the digest with the engine using the RSA algorithm with pkcs1.5 padding. Then, it performs signature verification without the engine checking that it is successful.
- **rsa_verify_pss** - This test creates a random digest, creates an RSA random key pair and divides the public and private key, signs the digest both through the engine and without, using RSA algorithm with PSS padding. Then, it performs signature verification with and without the engine, checking that it is successful in both cases.
- **rsa_verify_withpad** - This test creates a random digest smaller than the block size, creates an RSA random key pair and divides the public and private key, signs the digest with the engine using the RSA algorithm with pkcs1.5 padding. Then, it performs signature verification with and without the engine, checking that it is successful in both cases.
- **x509_ec** - This test creates an ECDSA curve secp521r1 key pair then creates the self-signed certificate on the private key. Finally, it creates a TLS server using the key and certificate.
- **x509_rsa** - This test creates an RSA 2048 key pair and then creates the self-signed certificate on the private key. Finally, it creates a TLS server using the key and certificate.

To use the engine in your code you must add this code before any cryptographic operations

```
ENGINE *e = ENGINE_by_id("linksengine");
if(!e) exit(EXIT_FAILURE);
```

and this command should be added at the end to free the object.

```
ENGINE_free(e);
```

A.3 linksprovider

Linksprovider is a provider developed for OpenSSL 3.0.x as a test of the models studied. It uses the cryptographic primitives of Mbed-TLS and implements: RSA and ECDSA key generation; a set of encoding functions; random number generation; signing and verification of RSA-PKCS1.5 and ECDSA algorithms (SECP256R1, SECP521R1, SECP192K1, SECP384R1 curves). Being a test program to get familiar with the engines, it should not be considered a complete example of an engine implementing all available library functions and applications.

A.3.1 Requirements

The following dependencies must be installed before installing linksprovider.

OpenSSL 3.0.x

OpenSSL 3.0.x can be found in the OpenSSL's official site [44].

Mbed-TLS 2.28

The provider is based on Mbed-TLS's cryptographic primitives and must be installed following the official instructions [36] or, if available in your distro, using this command

```
sudo apt install libmbedtls-dev
```

A.3.2 Installation

After correctly editing the installation path and include path in the Makefile according to your system, run the following commands

```
sudo make
sudo make install
make tests
```

A.3.3 Use

To test this provider, a set of tests is given and they are called simply by calling the chosen test [TEST_NAME] with this line command

```
./test/[TEST_NAME].sh
```

[TEST_NAME] is the name of one of the following tests:

- **rsa_testkey** - This test creates with the provider an RSA key pair and tests that, both the public and private parts are valid.
- **ec_testkey** - This test creates with the provider a pair of ECDSA keys with the curve of your choice between secp384r1, secp192k1, secp521r1, prime256v1 based on which row you remove the comment and tests that, both the public and private parts are valid.
- **rsa_sign** - This test creates a random digest, creates an RSA random key pair with the provider, and divides the public and private keys. Then, it signs the digest using the RSA algorithm with pkcs1.5 padding, and verifies it, checking that it is successful.
- **rsa_sign_prov** - This test creates a random digest, creates an RSA random key pair with the provider, and divides the public and private keys. Then, it signs the digest with the provider using the RSA algorithm with pkcs1.5 padding, and verifies it, checking that it is successful.
- **rsa_sign_prov_verif_prov** - This test creates a random digest, creates an RSA random key pair with the provider, and divides the public and private keys. Then, it signs the digest with the provider using the RSA algorithm with pkcs1.5 padding, and verifies it with the provider, checking that it is successful.
- **ec_sign** - This test creates with the provider a pair of ECDSA keys, with the curve of your choice between secp384r1, secp192k1, secp521r1, prime256v1 based on which row you remove the comment, and divides the public and private key. Then, it signs the digest using the ECDSA algorithm, and verifies it, checking that it is successful.

- **ec_sign_prov** - This test creates with the provider a pair of ECDSA keys, with the curve of your choice between secp384r1, secp192k1, secp521r1, prime256v1 based on which row you remove the comment, and divides the public and private key. Then, it signs the digest with the provider using the ECDSA algorithm, and verifies it, checking that it is successful.
- **ec_sign_prov_verif_prov** - This test creates with the provider a pair of ECDSA keys, with the curve of your choice between secp384r1, secp192k1, secp521r1, prime256v1 based on which row you remove the comment, and divides the public and private key. Then, it signs the digest with the provider using the ECDSA algorithm, and verifies it with the provider, checking that it is successful.
- **random** - This test generates 1000 bytes of random text using the provider.

To use the provider in your code you must add this code before any cryptographic operations

```
OSSL_PROVIDER *provider = OSSL_PROVIDER_load(NULL, id));
if (provider == NULL) exit(EXIT_FAILURE);
```

and this command should be added at the end to free the object.

```
OSSL_PROVIDER_unload(provider);
```

A.4 OpenSSL 3.0.4 operation DID

This is the modified version of OpenSSL with integrated the new operation to handle DIDs and their supporting APIs.

A.4.1 Requirements

The requirements are the same as the official version, which can be found on the OpenSSL's official github [45].

A.4.2 Installation

In the OpenSSL directory and launch these commands

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

in case of problems when launching OpenSSL commands use this command

```
sudo ldconfig
```

(or sudo ldconfig [file .so path])

A.5 didprovider

This provider allows the CRUD functions of the DID method OTT to be used with the modified version of OpenSSL.

A.5.1 Requirements

The following dependencies must be installed before installing didprovider.

Iota.c

Iota.c from branch dev [46]. First of all, keep a note of the installation path([Iotapath]).

```
git clone --branch dev https://github.com/iotaledger/iota.c.git
cd iota.c
mkdir build && cd build
cmake -DCMAKE_C_COMPILER=gcc -DCMAKE_CXX_COMPILER=gcc
    -DIOTA_WALLET_ENABLE:BOOL=FALSE -DWITH_IOTA_CLIENT:BOOL=TRUE
    -DCMAKE_INSTALL_PREFIX=$PWD -DCryptoUse=libsodium ..
```

Go to the folder *[Iotapath]/iotac/cmake* and open the file *sodium.cmake* and, in the row `CONFIGURE_COMMAND`, add `cxxflags=-fPIC` and remove `-no_shared`. Then run these commands

```
sudo make
sudo make install
```

In case of runtime problems of didprovider, you need to copy the files regarding `libsodium.so` in the folder *[Iotapath]/iotac/build/lib* and move them to the dedicated shared libraries folder on your system.

OpenSSL 3.0.4 operation DID

see section A.4

A.5.2 Installation

In the Makefile, change the inclusion path of iota.c files accordingly with your [Iotapath] and the installation path and include path according to your system. Then run these commands

```
sudo make
sudo make install
```

A.6 Demo application

This application tests the modified version of OpenSSL with the DID operation and its DID provider.

A.6.1 Installation

Check the OpenSSL .h file inclusion path according to your system, possibly modifying it in the Makefile, then run this command

```
sudo make
```

A.6.2 Use

The demo is ready to use i.e. already provided with public keys. With OpenSSL it is possible to generate other public keys paying attention to that: they are in PEM format; they are of one of the supported types and the filename is the same as the ones present. The demo application takes care of creating a DID document from the provided keys, does the resolve function of the new DID, does update the document with other keys, and lastly revokes it. A more detailed description can be found in section 4.

To get it started this command is used

```
./main.o
```

Each step of the program is shown on the screen, each operation is verified by a subsequent resolve, and thanks to the API created, the keys entered in the document are extracted, allowing verification with the input keys.

Given the modular nature of the API and the provider, it is possible to implement your code that also partially performs one of the operations performed by the demo application. In any case, the code should contain the following commands

```
//load the did provider for did operations
    provider = OSSL_PROVIDER_load(NULL, "didprovider");
    if(provider == NULL){
        printf("DID provider load failed\n");
        goto error;
    }

 //Creation of new did context
    didctx = DID_CTX_new(provider);
    if(didctx == NULL){
        printf("DID CTX new failed\n");
        goto error;
    }

//Creation of new did document
    did_doc = DID_DOCUMENT_new();
    if(did_doc == NULL){
        printf("DID document new failed\n");
        goto error;
    }

    ret = DID_fetch(NULL,didctx,"OTT","property");
    if(ret == 0){
        printf("DID fetch failed\n");
        goto error;
    }
```

With this piece of code, you load the didprovider, create an empty context, create an empty structure where you insert the DID Document, and finally fetch the instructions into the context. With the context loaded in this way, you can call one of the four CRUD functions explained in section 3.6.2. To test and validate the code you can, in addition to the provided on-screen output, either run the resolve function after one of the other functions or use `DID_DOCUMENT_get_auth_key` or `DID_DOCUMENT_get_assertion_key` on an obtained document to extract the public keys and compare them with the expected ones. Another possible way to validate is to search in IOTA Explorer [47], with devnet as the network, for the massage index created after a create, update or revoke operation. On the site it will be possible to see that there is a message on that index.

Appendix B

Developer Manuals

This section explains in detail how software works

B.1 linksengine

This software is divided into several modules on multiple files: *links-engine.c* contains the control and general functions of the engine; *ecc.c* contains all functions regarding ECDSA algorithms; *rsa.c* contains all functions regarding RSA algorithms; *rand.c* contains all functions regarding random number generation.

Below, the most important functions to understand its operation are given and briefly explained.

bind

Input parameters:

- **ENGINE *e** - pointer to the ENGINE object, used by some others OpenSSL's setters function
- **const char *id** - unused

Return values:

- **1** - on success
- **0** - on failure

This is the first function that is called when the engine starts and loads all the functions offered and calls the init functions of the various modules.

init_rand

Input parameters:

- **ENGINE *e** - passed to the `ENGINE_set_RAND` function

Return values:

- **1** - on success
- **0** - on failure

In engines to define RNG functions, one must fill in a function structure where each parameter indicates a specific function. This function sets the structure through the command `ENGINE_set_RAND`.

random_bytes

Input parameters:

- **unsigned char *buf** - buffer to be filled with random data
- **int num** - size in bytes of random data to be created

Return values:

- **1** - on success
- **0** - on failure

This function is responsible for generating random bytes. After initializing the entropy structure of Mbed-TLS `mbedtls_entropy_init( &entropy )`, it generates random bytes through the function `mbedtls_entropy_func` via a while loop because the function, in the single iteration, generates at most *MBEDTLS_ENTROPY_BLOCK_SIZE*.

init_rsa

Input parameters:

- **ENGINE *e** - pointer to the ENGINE object, used by the RSA OpenSSL's setters function

Return values:

- **1** - on success
- **0** - on failure

This function deals with modifying the standard RSA functions of OpenSSL. Since there are only a few modified functions, it copies the standard methods via the command `RSA_PKCS1_OpenSSL` and then individual functions are modified via the appropriate functions, e.g. `RSA_meth_set_priv_enc`.

rsa_priv_enc

Input parameters:

- **int from** - the data to be signed
- **const unsigned char *flen** - length of the from buffer
- **unsigned char *to** - the buffer to write the signature to
- **RSA * rsa** - the RSA key object
- **int padding** - the padding scheme to be used

Return values:

- **size_t size** - size of the returned signature
- **0** - on failure

This function performs the encrypt function using the private key in RSA, used to perform signature and authentication. After a padding check, the Mbed-TLS creation context is initialized via `mbedtls_rsa_init` and the RSA key, in OpenSSL form, is translated into the form accepted by Mbed-TLS by copying the individual fields from Bignum [48] to string and then imported into the mpi [49] form via `mbedtls_mpi_read_string`. The context is loaded with the RSA key parameters and then signed according to the padding type, `mbedtls_rsa_private` in the case without padding or `mbedtls_rsa_rsassa_pkcs1_v15_sign` for PKCS1V1.5 padding.

rsa_pub_dec

Input parameters:

- **int flen** - length of the from buffer
- **const unsigned char *from** - buffer to decrypt
- **unsigned char *to** - buffer to place plaintext in
- **RSA *rsa** - the RSA key object
- **int padding** - the padding scheme to be used

Return values:

- **size_t size** - size of the returned plaintext on success
- **-1** - on failure

This function performs the decryption function using the public key in RSA. After a padding check, the Mbed-TLS creation context is initialized via `mbedtls_rsa_init` and the RSA key, in OpenSSL form, is translated into the form accepted by Mbed-TLS by copying the individual fields from Bignum to string and then imported into the mpi form via `mbedtls_mpi_read_string`. The context is loaded with the public RSA key and then verified according to the padding type, `mbedtls_rsa_public` in the case without padding, or `mbedtls_rsa_rsassa_pkcs1_v15_decrypt` for PKCS1V1.5 padding.

init_ecc

Input parameters:

- **ENGINE *e** - unused

Return values:

- **1** - on success
- **0** - on failure

This function deals with modifying the standard ECDSA functions of OpenSSL. Since there are only a few modified functions, it copies the standard methods via the command `EVP_PKEY_meth_new-(EVP_PKEY_EC, 0);` and then individual functions are modified via the appropriate functions, e.g. `EVP_PKEY_meth_set_sign`.

pkey_ecdsa_sign

Input parameters:

- **EVP_PKEY_CTX *ctx** - PKEY context of operation
- **unsigned char *sig** - buffer to hold signature data or NULL to indicate the request of length signature
- **size_t *sigLen** - length of signature buffer
- **const unsigned char *tbs** - bo Be Signed data.
- **size_t tbsLen** - length of To Be Signed data.

Return values:

- **1** - on success
- **0** - on failure

This function signs data with a private EC key. The Mbed-TLS creation context and key object is initialized via `mbedtls_ecdsa_init` and `mbedtls_ecp_keypair_init`, respectively. The ECDSA key object is extracted from the `EVP_PKEY` object, alongside the information about the curve and the group. With this information, the Mbed-TLS key object is filled by copying the individual fields from Bignum to string and then imported into the mpi form via `mbedtls_mpi_read_string`. After loading the key inside the context via `mbedtls_ecdsa_from_keypair`, the signature is created with the function `mbedtls_ecdsa_write_signature_det`.

pkey_ecdsa_verify

Input parameters:

- **EVP_PKEY_CTX *ctx** - pkey context of operation
- **unsigned char *sig** - signature data
- **size_t *sigLen** - length of signature data
- **const unsigned char *tbs** - to be signed data.
- **size_t tbsLen** - length of to be signed data.

Return values:

- **1** - on successful verification
- **0** - on failure
- **-1** - on error

This function verifies a signature with a public EC key. The Mbed-TLS creation context and key object is initialized via `mbedtls_ecdsa_init` and `mbedtls_ecp_keypair_init`, respectively. The ECDSA public key object is extracted from the `EVP_PKEY` object, alongside the information about the curve and the group. With this information, the Mbed-TLS key object is filled by copying the individual fields from Bignum to string and then imported into the mpi form via `mbedtls_mpi_read_string`. The signature is verified with the function `mbedtls_ecdsa_verify`.

B.2 linksprovider

This software is divided into several modules on multiple files: *links-provider.c* contains the control and general functions of the provider;*bio_prov.c*, taken from the OpenSSL library, contains some library functions to handle buffers and library context; *keymgmt.c* contains all functions to create RSA and EC keys; *encoder.c* contains all functions to encode the generated data; *signature.c* contains all functions to create and verify RSA and ECDSA signatures; *rand.c* contains all functions regarding random number generation.

Below, the most important functions to understand its operation are given and briefly explained.

OSSL_provider_init

Input parameters:

- **const OSSL_DISPATCH **out** - pointer used to be pointed to the provider dispatch table
- **const OSSL_CORE_HANDLE *handle** - pointer to be set in the provider context
- **void **provctx** - provider context to be passed to other functions
- **const OSSL_DISPATCH *in** - vector from which the parameters to be set are extracted

Return values:

- **1** - on success
- **0** - on failure

This is the provider input function that is responsible for receiving the initial parameters, the library context, and providing the provider function vector.

links_query_operation

Input parameters:

- **void* provCtx** - unused
- **int id** - an integer that represents the operation id to be fetched
- **int* no_cache** - unused

Return values:

- **OSSL_DISPATCH * pointer** - pointer of the `OSSL_DISPATCH` id type
- **NULL** - on failure

This function is called by OpenSSL whenever it wants to access implementations of the operations provided by the provider. Based on the `Operation_Id` it returns the corresponding `OSSL_ALGORITHM` vector.

links_rsa/ec_keymgmt_gen_init

Input parameters:

- **void *provctx** - provider context of operation
- **int selection** - unused
- **const OSSL_PARAM params**[] - vector from which the parameters to be set are extracted

Return values:

- **RSA_CTX/EC_CTX *gen** - the context of generation created
- **NULL** - on failure

These functions perform the preparatory operations for key generation and call the function that defines the command-definable parameters.

links_rsa/ec_keymgmt_gen

Input parameters:

- **void *ctx** - the previously created context of generation
- **OSSL_CALLBACK *cb** - unused
- **void *cbarg** - unused

Return values:

- **RSA_PKEY/EC_PKEY** - on successful creation returns the key object filled
- **NULL** - on failure

These functions perform key generation. In the case of RSA keys, the function initializes a context `RSA_PKEY` containing the RSA key parameter variables and, via the command `mbedtls_rsa_gen_key`, generates the key which is then exported to the context via the command `mbedtls_rsa_export`. In the case of ECDSA keys, the function initializes a context `EC_PKEY` containing the ECDSA key parameter variables. It loads the group and the curve and then generates the key thanks to the function `mbedtls_ecp_gen_keypair`.

links_rsa/ec_keymgmt_gettable_params

Input parameters:

- **void *provctx** - unused

Return values:

- **OSSL_PARAM * gettable** - vector containing the obtainable parameters

These functions return a structure `OSSL_PARAM` in which, for each line, it indicates a parameter type that can be extracted from the key of that particular algorithm. For example, `OSSL_PARAM_BN(-OSSL_PKEY_PARAM_RSA_N, ...)` indicates that the N parameter of an RSA key can be extracted in a variable of type Bignum.

links_rsa/ec_keymgmt_get_params

Input parameters:

- **void *keydata** - key object
- **OSSL_PARAM params**[] - vector from which set the requested parameters

Return values:

- **1** - on success
- **0** - on failure

These functions extract the parameters defined in the function `links_rsa/ec_keymgmt_gettable-_params`. The function receives as input a vector `OSSL_PARAM` that contains the parameters to be extracted. Through the function `p = OSSL_PARAM_locate(params, param_name);` it looks for the parameter named *param_name*and, after having retrieved the parameter from the key object, is inserted in the vector.

links_rsa/ec_keymgmt_import_types

Input parameters:

- **int selection** - unused

Return values:

- **OSSL_PARAM * rsa/ec_key_params** - vector containing the importable parameters

These functions return a structure `OSSL_PARAM` in which, for each line, it indicates a parameter type that can be imported for that particular algorithm. For example, `OSSL_PARAM_BN(OSSL_PKEY_PARAM-_RSA_N, ...)` indicates that the RSA key can import the N parameter of type Bignum.

links_rsa/ec_keymgmt_import

Input parameters:

- **void *keydata** - key object
- **int selection** - an integer indicating which part of the key to import
- **OSSL_PARAM params**[] - vector from which import the parameters

Return values:

- **1** - on success
- **0** - on failure
-

These functions extract the parameters defined in the function `links_rsa/ec_keymgmt_import-_types`. The function receives as input a vector `OSSL_PARAM` that contains the parameters to be imported. Through the function `p = OSSL_PARAM_locate(params, param_name);` it looks for the parameter named *param_name* and, in case it is found, it is pointed to by p and and can be imported in the key.

links_rsa/ec_encoder_encode_text

Input parameters:

- **void *ctx** - context of the encoding process
- **const void *key** - key to encode
- **OSSL_CORE_BIO *cout** - BIO buffer to fill
- **OSSL_PASSPHRASE_CALLBACK *cb, const OSSL_PARAM key_abstract**[], **void *cbarg** - unused

Return values:

- **1** - on success
- **0** - on failure

These functions take care of printing the key passed to the screen, for example in the case of parameter -text during generation. Each key parameter is transformed into a string and printed through the `BIO_printf` command, with a buffer pointing to stdout.

links_rsa/ec_encoder_encode_PEM

Input parameters:

- **void *ctx** - context of the encoding process
- **const void *key** - key to encode
- **OSSL_CORE_BIO *cout** - BIO buffer to fill
- **OSSL_PASSPHRASE_CALLBACK *cb, const OSSL_PARAM key_abstract[], void *cbarg** - unused

Return values:

- **1** - on success
- **0** - on failure

These functions take care of transforming the key in Mbed-TLS form to a string in PEM format. After creating a new BIO buffer [50], the key is saved in PEM form by the following function `mbedtls_pk_write_key_pem` in a temporary buffer and finally copied to the BIO buffer through the function `BIO_write`.

links_rand_instantiate and links_rand_uninstantiate

Input parameters:

- **void *ctx** - context of operation
- **unsigned int strength,int prediction_resistance,const unsigned char *pstr, size_t pstr_len,const OSSL_PARAM params[]** - unused

Return values:

- **1** - on success
- **0** - on failure

These functions instantiate and free the contexts used during the creation of random bytes.

links_rand_generate

Input parameters:

- **void *ctx** - context of the encoding process
- **unsigned char *out** - buffer to be filled with random data
- **size_t outlen** - size in bytes of the random data to be generated
- **unsigned int strength, int prediction_resistance,const unsigned char *adin, size_t adinlen** -unused

Return values:

- **1** - on success
- **0** - on failure

This function generates random bytes. Bytes are generated through the repetition of the function `mbedtls_ctr_drbg_random` as, at each iteration, it generates at most `MBEDTLS_CTR_DRBG_MAX-_REQUEST` bytes.

links_rsa/ec_signature_sign_init

Input parameters:

- **void *ctx** - signature context to be filled
- **void *provkey** - provider context to be added in the signature context
- **const OSSL_PARAM params**[] - vector from which set the requested parameters

Return values:

- **1** - on successful verification
- **0** - on failure

These functions set the default parameters for signature generation and some configurable parameters passed from the `params` vector.

links_rsa/ec_signature_sign

Input parameters:

- **void *ctx** - signature context previously filled
- **unsigned char *sig** - signature data
- **size_t *sigLen** - length of the computed signature or NULL, in that case, this variable will be filled with an estimation of signature size
- **const unsigned char *tbs** - to be signed data.
- **size_t tbsLen** - length of to be signed data.

Return values:

- **1** - on success
- **0** - on failure

These functions deal with signature creation. If the functions receive the field containing the buffer where to save the signature, they return the estimate of how much the signature will occupy for that particular type of algorithm, otherwise, they take care of generating the signature. In the case of RSA keys, the Mbed-TLS context is initialized via `mbedtls_rsa_init` and the RSA key is imported in via the function `mbedtls_rsa_import`. The signing then is performed through the function `mbedtls_rsa_pkcs1_encrypt`. In the case of ECDSA keys, the context of Mbed-TLS is created through the function `mbedtls_ecdsa_init`. After loading the key within the context via `mbedtls_ecdsa_from_keypair`, the signature is created with the function `mbedtls_ecdsa_write_signature_det`.

links_rsa/ec_signature_verify_init

Input parameters:

- **void *ctx** - verification context to be filled
- **void *key** - object that contains the key
- **const OSSL_PARAM params**[] - vector from which set the requested parameters

Return values:

- **1** - on success
- **0** - on failure

These functions set the default parameters for signature verification.

links_rsa/ec_signature_verify

Input parameters:

- **void *ctx** - verification context previously filled
- **unsigned char *sig** - signature data
- **size_t *sigLen** - length of the signature
- **const unsigned char *tbs** - to be signed data.
- **size_t tbsLen** - length of to be signed data.

Return values:

- **1** - on successful verification
- **0** - on verification failure

These functions deal with signature verification. In the case of RSA keys, the Mbed-TLS context is initialized via `mbedtls_rsa_init` and the RSA key is imported in via the function `mbedtls_rsa_import` and then, the verification is performed through the function `mbedtls_rsa_rsassa_pkcs1_v15_verify`. The result of the function is returned. In the case of ECDSA keys, the context of Mbed-TLS is created through the function `mbedtls_ecdsa_init`. After loading the key within the context via `mbedtls_ecdsa_from_keypair`, the signature is verified with the function `mbedtls_ecdsa_read_signature` and the result of the verification is returned.

B.3 OpenSSL 3.0.4 operation DID

Previously explained in section 3.6.1.

B.4 didprovider

Previously explained in section 3.8.

Bibliography

[1] A Europe fit for the digital age, https://ec.europa.eu/info/strategy/priorities-2019-2024/europe-fit-digital-age_en

[2] European Digital Identity, https://ec.europa.eu/info/strategy/priorities-2019-2024/europe-fit-digital-age/european-digital-identity_en

[3] Introduction to Digital Identity by walt.id, https://walt.id/s/Introduction-to-Digital-Identity-waltid.pdf

[4] D. W. Chadwick, "Federated identity management", Foundations of Security Analysis and Design V (A. Aldini, G. Barthe, and R. Gorrieri, eds.), pp. 96–120, Springer, 2009, DOI 10.1007/978-3-642-03829-7_3

[5] How Decentralized Identity Is Reshaping Privacy For Digital Identities, https://www.forbes.com/sites/forbestechcouncil/2021/12/10/how-decentralized-identity-is-reshaping-privacy-for-digital-identities/

[6] W3C Recommendation Decentralized Identifiers (DIDs) v1.0, https://www.w3.org/TR/did-core/

[7] Ethr-DID Library repository, https://github.com/uport-project/ethr-did

[8] JSON - Introduction, https://www.w3schools.com/js/js_json_intro.asp

[9] W3C Recommendation Decentralized Identifiers (DIDs) v1.0 chapter Core Properties, https://www.w3.org/TR/did-core/#did-document-properties

[10] Introduction to Trust Over IP Version 2.0, https://trustoverip.org/wp-content/uploads/Introduction-to-ToIP-V2.0-2021-11-17.pdf

[11] DIDComm Messaging v2.x Editor's Draft, https://identity.foundation/didcomm-messaging/spec/

[12] A. Deshpande, K. Stewart, L. Lepetit, and S. Gunashekar, "Distributed ledger technologies/blockchain: Challenges, opportunities and the prospects for standards", Overview report The British Standards Institution (BSI), vol. 40, 2017, p. 40

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

[14] The Tangle, https://wiki.iota.org/learn/about-iota/tangle

[15] S. Popov, "The tangle", White paper, vol. 1, no. 3, 2018

[16] A simple explanation of the IOTA Coordicide, https://medium.com/@markusgebhardt/a-simple-explanation-of-the-iota-coordicide-8c9362472188

[17] Roadmap to Decentralization, https://wiki.iota.org/learn/about-iota/roadmap-to-decentralization

[18] IOTA Streams, https://www.iota.org/solutions/streams

[19] P. Rogaway, "Authenticated-encryption with associated-data", Proceedings of the 9th ACM Conference on Computer and Communications Security, Washington, DC (USA), 2002, pp. 98–107

[20] M. J. Saarinen and J. P. Aumasson, "The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)." RFC-7693, November 2015, DOI 10.17487/RFC7693

[21] The OpenSSL project, https://www.openssl.org/

[22] OpenSSL Strategic Architecture, https://www.openssl.org/docs/OpenSSLStrategicArchitecture.html

[23] OpenSSL version, https://www.openssl.org/docs/man3.0/man3/OpenSSL_version.html

[24] OpenSSL migration guide, https://www.openssl.org/docs/man3.0/man7/migration_guide.html

[25] OpenSSL crypto, https://www.openssl.org/docs/manmaster/man7/crypto.html

[26] provider-encoder, https://www.openssl.org/docs/man3.0/man7/provider-encoder.html

[27] OpenSSL 3.0.0 Design, https://www.openssl.org/docs/OpenSSL300Design.html

[28] OpenSSL default provider source code, https://github.com/openssl/openssl/blob/master/providers/defltprov.c

[29] OpenSSL default provider: decoder operation source code, https://github.com/openssl/openssl/blob/master/providers/implementations/encode_decode/decode_pem2der.c

[30] Trusted Platform Module Library Part 1: Architecture Level 00 Revision 01.59, https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part1_Architecture_pub.pdf

[31] tpm2-tss-engine, https://github.com/tpm2-software/tpm2-tss-engine

[32] TPM Software Stack (TSS), https://trustedcomputinggroup.org/work-groups/software-stack/

[33] TCG TSS 2.0 Enhanced System API (ESAPI) Specification Version 1.00 Revision 14, https://trustedcomputinggroup.org/wp-content/uploads/TSS_ESAPI_v1p0_r14_pub10012021.pdf

[34] tpm2tss-genkey manual page, https://github.com/tpm2-software/tpm2-tss-engine/blob/master/man/tpm2tss-genkey.1.md

[35] tpm2-tools manual: tpm2_evictcontrol, https://github.com/tpm2-software/tpm2-tools/blob/master/man/tpm2_evictcontrol.1.md

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

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

[38] openssl-pkeyutl, https://www.openssl.org/docs/manmaster/man1/openssl-pkeyutl.html

[39] openssl-rsa, https://www.openssl.org/docs/manmaster/man1/openssl-rsa.html

[40] tpm2-openssl rsa_createak_sign_handle.sh test, https://github.com/tpm2-software/tpm2-openssl/blob/master/test/rsa_createak_sign_handle.sh

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

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

[43] tpm2-tss-engine/INSTALL.md, https://github.com/tpm2-software/tpm2-tss-engine/blob/master/INSTALL.md

[44] OpenSSL source, https://www.openssl.org/source/

[45] OpenSSL installation instructions, https://github.com/openssl/openssl/blob/master/INSTALL.md

[46] iotaledger repository, https://github.com/iotaledger/iota.c/tree/dev

[47] IOTA Explorer, https://explorer.iota.org/devnet

[48] bn, https://www.openssl.org/docs/man1.0.2/man3/bn.html

[49] mbedTLSLibrary bignum.h File Reference, https://os.mbed.com/users/ansond/code/mbedTLSLibrary/docs/tip/bignum_8h.html

[50] BIO, https://wiki.openssl.org/index.php/BIO