UNIVERSITY OF TRENTO

Department of Information Engineering and Computer Science

Bachelor's Degree in
Computer Science

Final Dissertation

PKCS#11 Compliance Testing of the Italian Identity Card

Supervisor
Prof. Marco Patrignani
Co-Supervisor
Dr. Andrea Stedile

Student
Matteo Parma
226800

Academic year 2024/2025

Acknowledgements

I would like to express my gratitude to my supervisor, Marco Patrignani, for his support in overseeing my thesis project. I am also deeply grateful to my co-supervisor, Andrea Stedile, for his help, his constant availability and his fundamental support throughout my thesis work. I would like to thank the professors, who inspired and guided me during my studies, and my university colleagues, for sharing this experience with me and for helping me grow both personally and professionally. I also want to thank my friends, my companions in adventures and study, for encouraging and supporting me. A special thank you for my family, for their constant support throughout this journey.

Contents

1 Introduction

PKCS#11 is one of the Public Key Cryptographic Standards defined by RSA Laboratories, which defines a standard for operating with cryptographic devices, such as Hardware Security Modules (HSM) and Smart Cards, and cryptographic libraries. These devices can hold several cryptographic objects and perform different cryptographic functions. The main goal of PKCS#11 is to provide interoperability between these devices, defining an interface that abstracts from their details.

The PKCS#11 standard presents two problems. The first is that the standard has been shown to be vulnerable to some known attacks that could, for example, compromise the sensitive keys stored on the device [15]. Indeed, the real problem for cryptography, rather then perform cryptographic operations, is the secure storage and use of the keys. Since there are not well established fixes, vendors, in practice, try to protect against these threats by restricting the functionality of the interface, or by adding extra features. This, in turn, could cause other problems: for example, an effective security patch for one operation may disable a functionality that is vital for another, generating the possibility for the library to deviate from the standard.

The second problem is that PKCS#11 is extremely complex and intrinsically ambiguous. In spite of the basic usage pattern appears clear, the behaviour of particular function invocations is often not explicitly specified and must be inferred by examining the return values. For each function the specification supplies a list of possible return values, corresponding to a set of behaviours, but the standard only provides generic descriptions for each of them, outside the context of any particular function. In addition, the list of return values some times is not exhaustive, as highlighted in a paragraph of the specification: *Because of the complexity of the Cryptoki*[1] *specification, it is recommended that Cryptoki applications attempt to give some leeway when interpreting Cryptoki functions' return values. We have attempted to specify the behaviour of Cryptoki functions as completely as was feasible; nevertheless, there are presumably some gaps.* Indeed, some functions have behaviours whose related return value is not listed. Cryptoki complexity and ambiguity make it difficult for vendors to implement libraries that are 100% accurate and compliant with the specification and often different PKCS#11 implementations show some differences. When a PKCS#11 library diverges from the standard, customer impact can range from delays in development to production outages or data corruption. In addition, error handling scenarios defined in the standard, if not properly handled, will cause program crashes or other serous problems.

The Italian Identity Card, known as CIE (Carta di Identità Elettronica), is among the many devices supporting PKCS#11 standard. The CIE is a smart card, issued by the Ministry of the Interior, which, thanks to a radio frequency chip, constitutes a digital identity tool for accessing online services and allows an easier recognition of the owner's identity.

This thesis aims to study the PKCS#11 module of CIE, evaluating the quality of the implementation and its compliance to the standard. In order to accomplish this goal, after a preliminary source code review to understand how the PKCS#11 API implemented by CIE work in detail, we have conducted several tests in order to verify the compliance to the standard. We have checked if all the on-chip object attributes are available and valid, and if the behaviour of the functions aligns with their documentation; in addition, we have compared the behaviour of the CIE library with another PKCS#11 implementation by performing the same set of operations. Some operations require a sequence of function calls interacting with each other in a stateful manner. For this reason, a test focusing only on the single function call, without concerning about what the actual state is, is not sufficient. Therefore, we also have conducted several stateful tests, in particular about digesting, signing and signature verification operations.

[1]The PKCS#11 standard is commonly called Cryptoki

Since CIE PKCS#11 module is built to comply with the PKCS#11 v2.11 specification, we referred to the documentation of that specific version.
Assuming the comparison library is fully compliant with the standard, any deviations found from its output, indicate a non-compliance element of the CIE library. During testing, we have assumed that the card is never disconnected during the operation because as the PKCS#11 documentation states: *Realistically, Cryptoki may not be constantly monitoring whether or not the token is present, and so the token absence could conceivably not be noticed until a Cryptoki function is executed. If the token is re-inserted into the slot before that, Cryptoki might never know that it was missing.* [...] *In practice, it is often not crucial (or possible) for a Cryptoki library to be able to make a distinction between a token being removed before a function invocation and a token being removed during a function execution.*
The CIE exclusively communicates with a contactless interface compliant with ISO 14443 supporting the NFC technology. In order to perform these tests we used a NFC Ewent smart card reader [3] that supports PC/SC protocol, working in a Windows 11 Home environment and developing using Visual Studio 2022 as IDE.

This thesis will proceed with a deeper exploration of the CIE and PKCS#11 in Section 2. This will be followed by the description of my work core with a detailed examination of the CIE PKCS#11 module implementation and the associated testing in Section 3. The thesis will conclude with a summary of the results of these tests in Section 4.

2 Background

This section describes in detail the CIE project and its applications [10] [1] in section 2.1, followed by an overview of the PKCS#11 standard [14] in section 2.2

2.1 CIE

The Electronic Identity Card (CIE) version 3.0, issued by the Ministry of the Interior, thanks to the presence of a radio frequency chip [8] containing the holder's personal and biometric data and a digital authentication certificate, extends the traditional concept of document ID, offering three services:

- Physical identity, in which the CIE is used as a personal identity verification device, allowing an easier recognition of the owner at customs and during police checks.
- Digital identity, using CIE to access online services.
- Document identity, using the Identification Number for Services (NIS)[9], a document identification number used for fast physical accesses, e.g. public transportation.

These three functionalities are implemented by two distinct on-chip applications: the first one by MRTD (Machine Readable Travel Document) and the last ones by IAS ECC (Identification Authentication Signature - European Citizen Card), the name is taken from the related specification [12].

2.1.1 MRTD: Personal Identity Verification

The MRTD application, similar to those used in passports or residence permits, allows personal identity verification, complying with international ICAO Recommendation 9303 [13]. The CIE, in accordance with Doc 9303, incorporates an MRZ (Machine Readable Zone), easily readable by OCR machines, improving police checks security and efficacy. Consisting of a different representation of the data found in the VIZ (Visual Inspection Zone) such as Document Number, Data of Birth and Date of Expiry, the MRZ provides the verification of this information. The compliance to the ICAO Doc 9303 allows the use of the CIE as a (Machine Readable) Travel Document recognised by countries in the Schengen area.

When checked by an operator, the reading and verification process of the personal and biometric data contained in the microchip verifies the authenticity of the document and the identity of the owner. In accordance with international standards, in order to protect that information from unauthorized accesses, CIE supports some security mechanisms defined by ICAO, including BAC (Basic Access Control). To process information with BAC, the reader needs to, in a challenge-response protocol, provide an access key pair, which is derived from the MRZ. A chip that is protected by the Basic Access Control mechanism denies access to its contents unless the inspection system can prove that it is authorized to physically access the chip. Therefore, personal data cannot be accessed without the owner's knowledge.

CIE supports other security mechanisms, such as Passive Authentication to protect the integrity and authenticity of the data stored in the chip, or PACE (Password Authentication Connection Establishment) providing an access control mechanism stronger than BAC, and EAC (Extended Access Control) to perform Chip and Terminal Authentication.

2.1.2 IAS ECC: NIS and Access to Online Services

The IAS ECC application supports two kinds of authentication mechanisms: document identification, using the NIS, and owner's identification by RSA private key and associated client authentication certificate. The first one does not require the terminal providing access to the service to have specific cryptographic capabilities, indeed there is no need to setup data encryption on the communication channel since the user's PIN or any other sensitive information is not required. The only data exposed is the NIS, which is public, and without personal information it only identifies the document, not the owner. Access using the NIS can be supported only in cases with low security level requirements. This access method can be used from services that require a quick verification, which would not be feasible or not convenient with PIN submission. However, the integrity, authenticity and validity of the NIS must be verified.

The second mechanism is supported by applications that need the maximum security level. Since personal owner's information, i.e. full name and personal ID (Codice Fiscale), are supplied, user's PIN and data encryption are required. For this reason the terminal must be capable to perform symmetric (3-DES) and asymmetric (RSA) cryptographic algorithms.
Since a secure communication channel is required, IAS specifies a Diffie-Hellman key exchange protocol to derive session keys for secure messaging. Diffie-Hellman key exchange is secure against an eavesdropping attack, but it is vulnerable to a Man-In-The-Middle attack. In order to resist to this kind of threat, the CIE uses an asymmetric key pair to conduct internal and external authentication. The IAS specification defines this mechanism through the "Devide Authentication with Privacy Protection" (DAPP) protocol, which involves the use of CVCs (Card Verifiable Certificate) for the external authentication, similarly to the Terminal Authentication described in the ICAO specification. In order to use CVCs, it is required that these certificates are issued by a PKI (Public Key Infrastructure) and distributed to the terminals. However, while this approach is conceivable for the MRTD application, where there is a limited number of terminals in a controlled environment, it is not feasible for IAS application, having extremely heterogeneous usage contexts, possibly open to any Service Provider that decides to use the CIE as authentication tool. In order to avoid the PKI, it has been decided that the authentication level required is the PIN for the terminal/user and the internal authentication specified in "Device Authentication with Privacy Protection" for the document.

The IAS application also offers a mechanism of protection against cloning, with a system similar to the Chip Authentication of the MRTD application, using an asymmetric key. The Internal Authentication step of the "Device Authentication with Privacy Protection" protocol already includes this mechanism, therefore no additional operation is required. In the case of access with the NIS, however, the terminal must be able to perform an asymmetric key cryptographic operation to complete a challenge/response protocol.
In order to ensure the data in the document is not forged, a Passive Authentication mechanism like the one used in the MRTD application is provided.

The IAS application supplies the basis for authentication and a secure data management, crucial for the "Entra con CIE" identification scheme. This scheme, interoperable also at European level, implements a federated authentication system for the identification of citizens with public and private entities that provide digital services online.
The "Entra con CIE" authentication scheme offers different access mechanisms based on the security level required by the service the user is accessing to. In particular:

- A so-called "low" level of access (level 1), which involves the use of username/password credentials that can be activated by the cardholder.
- A so-called "substantial" level of access (level 2), which involves the use of a second factor of authentication (OTP code via SMS or PUSH notification on the CieID app).
- A so-called "high" level of access (level 3), which involves the use of the CIE and the X.509 digital authentication certificate on board, together with the card's PIN.

For level 3 access, a direct interaction with the card is required. If the operation is performed on a computer, the CIE software, also called Middleware[11], must be installed. The Middleware is entitle to handle the interaction with the card and, thanks to the CieID[2] application supplied by the software, enables the owner to use their card to access services, but first, in order to habilitate the CIE for online accesses, the application needs to pair the card in order to verify that the document is valid. This is done performing the mutual authentication with DAPP. The CieID not only allows access to online services, but also offers other CIE functionalities, such as the document signature, signature verification, as well as card management operations like the possibility to change the PIN and unlock the card.

The Middleware implements several standards and tools for card communication, including the CryptoTokenKit and SmartCardMinidriver frameworks, as well as the PKCS#11 standard. My work focuses on the latter.

2.2 PKCS#11

PKCS#11, also known as Cryptoki, developed by RSA Laboratories and now managed by OASIS, is a standard that, following a simple object-base approach, specifies a C-language Application Programming Interface (API) to interact with devices that hold cryptographic information and perform cryptographic functions.

2.2.1 General Overview

The main goals of PKCS#11 are: providing a technology-independent interface, that abstracts the detail of the device, presenting to application a common logical view of the device called "cryptographic token", and resource sharing, allowing different applications to access various types of cryptographic tokens.

The standard specifies only the interface to the library, not its features. Indeed, the supported features will differ from each Cryptoki library implementation. In particular, libraries generally will support only a limited number of kinds of devices, and a subset of the operations defined in the standard, and for each operation not all libraries will support all the mechanisms.

Applications can interact with a token through the interface provided by Cryptoki, consisting of a set of "slots", generally corresponding to a physical reader to which a token can be connected.
To Cryptoki, an application consists of a single address space. An application becomes a "Cryptoki application" by calling the function `C_Initialize`; after this call is made, the application can call other Cryptoki functions. When the application is done using Cryptoki, it calls the function `C_Finalize` and it will no longer be a Cryptoki application.

2.2.2 Token and Objects

The logical view of a token is a device that stores cryptographic objects. Each object belongs to a specific class defined by Cryptoki. There are three main types of objects:

- Data Object: a class holding information defined by an application
- Certificate Object: divided in two classes holding respectively X.509, public key and attribute, certificates
- Key Object: holds encryption or authentication keys, they can be public, private or secret, Cryptoki defines a class for each specific type

Users will never interacts directly with an object, but they will always use a *handle* of that object (of type CK_OBJECT_HANDLE). Object handles are identifiers used to manipulate Cryptoki objects. They can be considered as a sort of pointer to its object. When, for example, a function requires a key object as argument, it will be passed its handle. Cryptoki also defines the session handle (of type CK_SESSION_HANDLE), a Cryptoki-assigned value that identifies a session, and its specified to functions to indicate in which session they should act on.

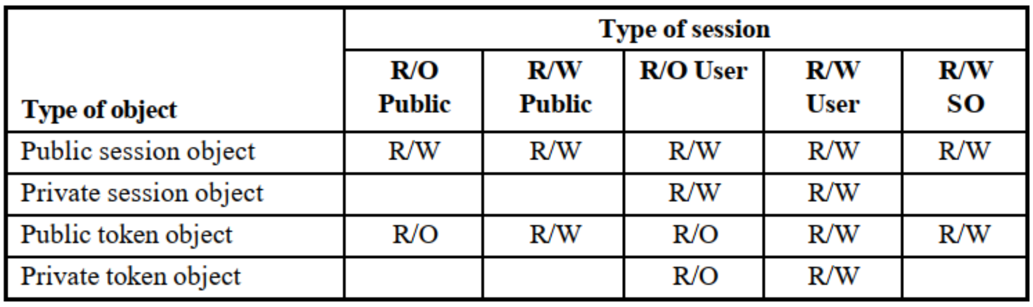
Each object possesses some attributes, which are characteristics that distinguish an instance of an object from the others. Cryptoki defines general attributes possessed by every object, other attributes belongs only to a specific type of object, such as any Key Object, and there are also attributes that are specific to a particular class, such as any RSA Private Key Object.

Objects are also classified according to their lifetime and visibility: "Token Objects" are potentially visible to all application connected to the token and they are persistent. "Session Objects" are visible only inside the associated session (connection between application and token) and they are temporary: whenever a session is closed, all the associated Session Objects are destroyed.
Further classification distinguish objects based on their access requirement. Session and Token objects can be Public or Private. The possibility to access to an object depends on the user's authorization level.

2.2.3 User and Session

Cryptoki recognizes two token user types. One type is the Security Officer (SO). The other type is the normal user. Only the normal user is allowed, after authenticating, to access to private objects on the token. The role of the SO is to initialize a token and to set the normal user's PIN, and possibly to manipulate some public objects.

Cryptoki requires that an application opens a session with a token in order to interact with it. A session consists in a logical connection between the application and the token, providing access to the token objects and functions. A session can be a read/write (R/W) session or a read-only (R/O) session. A session can have several states, produced by the combination of the session types and the user's levels of authorization. Each session state determines objects visibility and the actions that can be performed on them. A summary of all the possible session states and their associated objects accesses levels is shown in figure 2.1. Notice that a Security Officer can login only in a read/write session, and session objects accesses, if allowed, are always in a read/write state.



	Type of session				
Type of object	**R/O Public**	**R/W Public**	**R/O User**	**R/W User**	**R/W SO**
Public session object	R/W	R/W	R/W	R/W	R/W
Private session object			R/W	R/W	
Public token object	R/O	R/W	R/O	R/W	R/W
Private token object			R/O	R/W	

Source: `https://cryptsoft.com/pkcs11doc/STANDARD/pkcs-11v2-11r1.pdf`

Figure 2.1: Access to different types of objects by different types of sessions

We can identify three main types of operations an open session can perform: administrative operations (such as logging in); object management operations (such as creating or destroying an object on the token); and cryptographic operations (such as computing a message encryption).

2.2.4 Functions and Mechanisms

The Cryptoki API consist of a set of functions, used for session, slot, token and object management, as well as cryptographic operations. Functions can be accessed through a structure, CK_FUNCTION_LIST, which contains a pointer to each function in the Cryotoki API. A Cryptoki library does not need to support every function in the Cryptoki API. However, even an unsupported function must have a "stub" in the library which simply returns the value CKR_FUNCTION_NOT_SUPPORTED.

Each function returns a specific value, corresponding to a success, or mostly to a specific failure case. Cryptoki supplies a big number of return values, some of them can be returned by every function, while others concern only a subset of functions, but they all follow a common hierarchy. When multiple errors occurs during a function execution, only the error with the highest priority should be returned.

Cryptoki functions for cryptographic or digesting operations require a Mechanism to specify in order to be conducted. A mechanism corresponds to an algorithm, specifying how a certain operation is to be performed. For example an encryption operation can be performed using 3DES or RSA. Cryptoki specifies a large number of mechanisms, but a token can support only a subset of them.

3 CIE PKCS#11 Module Analysis and Compliance

This chapter starts with the analysis of the CIE PKCS#11 module deepening into the implementation details in section 3.1, introducing some elements not compliant with the standard. Followed by the compliance test in section 3.2.

3.1 Library Implementation

The CIE Middleware source code is accessible to anyone, giving the possibility to study how the PKCS#11 module is implemented.

3.1.1 General Overview

For each relevant PKCS#11 element, i.e. slot, session, object and mechanism, the Middleware defines a class, in which are stored attributes and methods used to accomplish the functionalities concerning them. These functionalities are channelled into the PKCS#11 functions in which they are actually used. For example, the slot class has a global map of all the slots and the function `C_GetSlotList` defined in the API iterates over this map, in order to supply the list of the slots.
Other information that does not specifically concern the elements of Cryptoki, but instead related to the CIE and its PKCS#11 objects, are reported in the CIEP11Template module, which defines the CIEData class including, for example, an IAS object, which implements the IAS ECC specifications, the asymmetric key pair used for signature, the certificate and the PIN. The module also defines functions vital to make some PKCS#11 API functions work, such as `C_Login` and `C_Sign`. The reason why an IAS object is present is that all the functions defined in CIEP11Template perform the operations defined in the IAS ECC specification. For example the `C_Login` API function calls the `Login` session method, which in turn calls the `CIEtemplateLogin` function defined in the CIEP11Template module, performing Diffie-Hellman key exchange and mutual authentication with DAPP.
In reality, referring to the previous example, `Login` session method do not directly call the CIEP11Template function, but it is called through a CCardTemplate object using a TemplateFuncList, a class with a pointer for each function defined in CIEP11Template. The reason of this additional layer is due to the necessity to provide the possibly to interact with multiple CIEs. Indeed, the CCardTemplate possesses a global map, more precisely a vector of shared pointers, of the CCardTemplates.
IAS module offers some functions used to support the more complex operation like DAPP, but they are also used by PKCS#11 API for simpler operations, such as reading the certificate. For example this operation is performed by `CIEtemplateInitSession`, executed calling the `C_OpenSession` API function, which establishes a connection with the card and reads the certificate.
The IAS class also supplies functions for PIN management, used by the CieID application, for example to verify the PIN during the CIE habilitation, but also by Cryptoki functions, i.e. `C_Login`, `C_Sign` and `C_SetPIN`.

3.1.2 Low Level Communication

The PKCS#11 API functions, as well as the IAS ECC operations like DAPP, obviously need to communicate with the card. In Windows environment, this task is delegated to the winscard [7] module, which, implementing the PC/SC protocol [5], provides an API for low-level communication with smart cards and smart card readers. The communication using PC/SC protocol consists of sending data packets, called APDU (Application Protocol Data Unit), corresponding to a sequence of commands and responses. In this way an application can read on-chip information, for example, sending an APDU command to read Diffie-Hellman domain parameters and receiving an APDU response with their values.

3.1.3 CIE Habilitation

Regarding CIE habilitation, during this operation, performed within the CieID application, the Middleware stores the document certificate, that will be used to access online services. If this operation is not performed, it is not possible to interact with the CIE through the PKCS#11 interface. Indeed, an attempt to open a session with a CIE not habilitated using `C_OpenSession` API function, it will return CKR_TOKEN_NOT_RECOGNIZED. The reason is that, before opening a new session with the card (in reality the session is opened with a slot), the Middleware will verify that the CIE is valid and that can be used. Indeed, the Middleware will iterate over the global map of the cards templates, in order to find the one that matches with that card. To verify that a template matches the card, the Middleware reads the CIE information, such as ATR, PAN and the certificate. If, for example, the ATR is not recognized or the certificate is not found in the Middleware cache, then the match operation will fail. If no successful match occurred, then the PKCS#11 function will fail with the CKR_TOKEN_NOT_RECOGNIZED error code.

3.1.4 Token Initialization

The operations that make the token operative are executed calling `C_OpenSession`, more precisely by the `CIEtemplateInitSession` function, which, after CIE habilitation verification, reads the certificate, from which also public key information are retrieved. Then the function initializes an instance of CIEData, setting the attributes of its PKCS#11 objects, i.e. certificate and RSA key pair, with the information retrieved by the previous certificate reading.
Regarding the attributes that the token provides, retrieved by the `C_GetTokenInfo` function, only a subset of those are supporte by the CIE. In particular the attributes ulTotalPublicMemory, ulTotalPrivateMemory, ulFreePublicMemory and ulFreePrivateMemory are not available.

3.1.5 Functions and Mechanisms Supported by the CIE

CIE PKCS#11 module supports only a limited number of functions. Indeed, the module offers the same functionalities of the CieID application, such as signature and signature verification operations, while data encryption or decryption are not supported, as well as key wrapping and unwrapping. Other unsupported functions are, for example, the ones whose result implies some kind of writing operation, such as `C_CopyObject`, `C_CreateObject` and `C_CancelFunction`. Indeed, the module, as specified in the software GitHub repository[4], is built to handle the card in read-only. But there are many others such as `C_InitToken` or `C_GetObjectSize`. There could be many reasons why these function are not supported and they are not explicitly mentioned, but, as the standard requires, an attempt to call any unsupported function will bring to the same result, retuning CKR_FUNCTION_NOT_SUPPORTED.
There is also a small number of supported mechanisms. These mechanisms can be used for digesting, signature and signature verification operations. More specifically, supporting CKM_SHA_1, CKM_SHA256 and CKM_MD5 mechanisms, digesting operation can be performed with their associated hashing algorithms; supporting CKM_RSA_PKCS, CKM_SHA1_RSA_PKCS,
CKM_SHA256_RSA_PKCS and CKM_MD5_RSA_PKCS, signature operation, always computed with the PKCS#1 RSA key, can be performed without hashing or using the associated algorithms, as well as signature verification.

3.1.6 Ambiguities and Deviations from the PKCS#11 Standard

The CIE specifies that the PKCS#11 standard to which it refers is the version 2.11. But we have seen that PKCS#11 module uses some elements, such as the signature with hash SHA-256 mechanism, CKM_SHA256_RSA_PKCS, which are introduced in later versions. This makes unclear what documentation to reference to during the analysis. Anyway, we have continued consulting the one for version 2.11.
Another strange thing we have noticed during code review concerns the PKCS#11 signature recover operation, in which the function `C_SignRecoverInit` is supported, while `C_SignRecover` is not, without a specified reason. This could bring to several issues, but, as we continued the study of the CIE PKSC#11 module, we realized that are more serious problems that make the library deviate from the behaviour required by the PKCS#11 standard.

While analysing the PKCS#11 module, we have noticed several non-compliance elements. Regarding the objects attributes , the standard states in section 13.3: *In Cryptoki, every object (with the possible exception of RSA private keys) always possesses all possible attributes specified by Cryptoki for an object of its type.* But in the `CIEtemplateInitSession` function, where the PKCS#11 objects are initialized, some attributes are missing. The specific missing attributes are listed below, divided by object category:

- Storage Object: its attributes must be present in all CIE objects. This kind requires four attributes: CKA_TOKEN, CKA_PRIVATE, CKA_LABEL and CKA_MODIFIABLE. The last one is not present in any of the objects.
- Keys: Cryptoki defines some common attributes for all the kind of keys. In CIE implementation some common attributes, i.e. CKA_START_DATE, CKA_END_DATE, CKA_DERIVE, CKA_LOCAL, CKA_KEY_GEN_MECHANISM, are missing in both keys.
- Public Key: Cryptoki provides six attributes specific for public keys, but CIE public key has only one of them: CKA_VERIFY
- Private Key: there are several attributes for private keys, but only a subset of them is present in CIE private key, in particular are missing the following attributes: CKA_SUBJECT, CKA_SIGN_RECOVER, CKA_UNWRAP, CKA_ALWAYS_SENSITIVE and CKA_NEVER_EXTRACTABLE.
- RSA Private Key: for this type Cryptoki makes an exception in the statement above, meaning that not all the associated attributes need to be present, but the standard requires the presence of at least the CKA_MODULUS and CKA_PRIVATE_EXPONENT attributes, but the last one is no present in CIE RSA private key.

In addition, attempting to call `C_GetAttributeValue` to retrieve the value of some unsupported attribute, the function returns CKR_ATTRIBUTE_TYPE_INVALID, even if Cryptoki specification in section 10.1.1 states: *An attribute is valid if it is either one of the attributes described in the Cryptoki specification or an additional vendor-specific attribute supported by the library and token.* If the PKCS#11 module wanted to deny the access to the value of an attribute, for example the private exponent of the RSA key, then the right way to implement it is to set the private key attributes CKR_SENSITIVE to true or CKR_EXTRACTABLE to false, therefore the correct value to return is CKR_ATTRIBUTE_SENSITIVE.
Regarding CIE PKCS#11 unsupported functions, some of them, as required by the PKCS#11 specification, simply return CKR_FUNCTION_NOT_SUPPORTED, but others, for some reason, perform some checks before returning this error. Although the functions do not complete the operation, these suggest that there is a possibility of returning different errors. For example, calling `C_GetObjectSize` with an invalid session handle, the function should incorrectly return CKR_SESSION_HANDLE_INVALID.
In addition, we have noticed that some implemented PKCS#11 functions returns some error codes which are not listed in their description. Since Cryptoki states: *it is possible that a particular error code which might apply to a particular Cryptoki function is unfortunately not actually listed in the description of that function as a possible error code. It is conceivable that the developer of a Cryptoki library might nevertheless permit his/her implementation of that function to return that error code*, this behaviour represent a clear non-compliant element.
The unexpected return values that we have found are:

- `C_InitPIN`: CKR_PIN_INCORRECT
- `C_Sign`: CKR_KEY_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN
- `C_SignUpdate`: CKR_KEY_FUNCTION_NOT_PERMITTED
- `C_SignFinal`: CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN
- `C_VerifyUpdate`: CKR_KEY_FUNCTION_NOT_PERMITTED

- `C_VerifyFinal`: CKR_KEY_FUNCTION_NOT_PERMITTED

In order to verify and prove that these non-compliant elements actually exist, we built an application in order to test the functions provided by the CIE PKCS#11 module.

3.2 PKCS#11 Compliance Testing

Our test focuses on the PKCS#11 functions, in particular those concerning digesting, signature and signature verification operations. Every function of interest has been tested calling it several times, passing different combinations of NULL arguments values, with invalid objects handles or other kind of arguments that could cause some problems. Then we have analysed the outputs, checking if the retuned value was the expected one, also checking if the return values hierarchy was respected in case of multiple errors. Cryptoki defines some specific situations in which a failure to a function call implicates the entire termination of the operation, or, on the contrary, situations in which a failure causes the operation not to terminate. In these cases, the operation status is checked.
Some functions, such as `C_Sign`, `C_SignFinal`, `C_Digest` and `C_DigestFinal`, follow the "Conventions for functions returning output in a variable-length buffer", depending on the mechanism used, described by Cryptoki in section 11.2 of its documentation. According to these conventions, when the pointer to the output buffer is a null pointer, then the functions return a number of bytes which would suffice to hold the output produced from the input of the function. Therefore, we have also checked that the output size returned by these function calls respects the constrains on the length of the output specified for each mechanism used.
Since cryptographic operations are defined in terms of a sequence of functions interacting with each other, a single function call, without considering the state of the operation, is not sufficient. For this reason we have also tested the functions behaviour considering the previous function calls and the state of the operation.
Regarding the tests performed with NULL argument values, Cryptoki usually does not describe a particular behaviour if a certain argument is set to NULL, but it defines a return value, CKR_ARGUMENTS_BAD, which is returned when arguments supplied to the function are not appropriate. In order to verify that the correct behaviour is returning that value, we have performed the same tests with another PKCS#11 library, SoftHSM [6], using it to compare its outputs with the ones produced by the CIE module, helping us to understand what the correct functions behaviour should be.
The specific test cases, preceded by a brief description of the tested functions, are listed in the next subsections. Their results are presented in chapter 4.

3.2.1 Digest, Signature and Signature Verification

These operations, in order to be performed correctly, need a specific sequence of function calls. They all perform an *initialization* in which a session handler and the mechanism are specified, in addition signature and verification also require an object handle. Then there are two ways to complete the operation. The first one is calling one or more time an *update* operation, usually used for multiple-part operations, processing the data in more steps, then, with a *finalization* function, computing the output and terminating the operation. The second way to complete digesting, signature and verification operations is through a single call to, `C_Digest`, `C_Sign` or `C_Verify`, respectively, which processes the data and computes the output all at once, used for single-part operations. These functions do not even require the finalization, since after calling it the operation is always terminated.
We have tested the behaviours of the single functions, trying different combinations of invalid arguments, and for each produced digest and signature, we have verified their correctness. In addition, we have performed a stateful test, consisting in a sequence of different functions calls, testing, rather than the single function behaviour, the operation state behaviour.
A description of each test case will follow.

Concerning digest operation:

- `C_DigestInit`: this function arguments are a session handler (hSession) and a pointer to the specified mechanism (pMechanism)

1. Invalid pMechanism
2. NULL pMechanism
3. NULL hSession
4. Both hSession and pMechanism NULL (hierarchy check)
5. Invalid pMechanism parameters

- `C_Digest`: it takes as input a session handler (hSession), a pointer to the data to hash (pData), its length (ulDataLen), a pointer to the location receiving the message digest (pDigest) and a pointer to the length of the message digest (pulDigestLen)

  1. NULL pDigest
  2. Both pData and ulDataLen NULL
  3. Only pData NULL
  4. Only ulDataLen NULL
  5. Buffer too small
  6. NULL pulDigestLen
  7. All arguments set to NULL (hierarchy check)

- `C_DigestUpdate`: the function arguments are a session handler (hSession), a pointer to the data block (pPart) and its length (ulPartLen)

  1. NULL hSession
  2. Both pPart and ulPartLen NULL
  3. Only pPart NULL
  4. Only ulPartLen NULL
  5. All arguments set to NULL (hierarchy check)

- `C_DigestFinal`: its arguments are a session handler (hSession), a pointer to the location that receives the message digest (pDigest) and the pointer to the location that holds its length (pulDigestLen)

  1. NULL pDigest
  2. Buffer too small
  3. NULL pulDigestLen
  4. NULL hSession

Concerning the signature operation:

- `C_SignInit`: this function takes as arguments a session handle (hSession), a pointer to mechanism (pMechanism) and the handle of the key used to sign (hKey)

  1. Invalid pMechanism
  2. NULL pMechanism
  3. NULL pSession
  4. All arguments set tu NULL (hierarchy check)
  5. pMechanism pointing to a mechanism with invalid paramateters
  6. NULL hKey
  7. RSA Public key handle as hKey

- `C_Sign`: its arguments are a session handle (hSession), a pointer to the data to sign (pData), its length (ulDataLen), a pointer to the location receiving the signature (pSignature) and a pointer to the location that holds the length of the signature (pulSignatureLen)
    1. NULL pSignature (output length check)
    2. Both pData and ulDataLen NULL
    3. Only pData NULL
    4. Only ulDataLen NULL
    5. Buffer too small
    6. NULL hSession
    7. NULL pulSignatureLen
- `C_SignUpdate`: taking as arguments a session handle (hSession), a pointer to the data block to sign (pPart) and its length (ulPartLen)
    1. NULL hSession
    2. Both pPart and ulPartLen NULL
    3. Only pPart NULL
    4. Only ulPartLen NULL
    5. All arguments set to NULL (hierarchy check)
- `C_SignFinal`: its arguments are a session handle (hSession), a pointer to the location that receives the signature (pSignature) and a pointer to the location that holds the length of the signature (pulSignatureLen)
    1. NULL pSignature (output length check)
    2. Buffer too small
    3. NULL pulSignatureLen
    4. NULL hSession
    5. All arguments set to NULL (hierarchy check)

Concerning the signature verification operation:

- `C_VerifyInit`: this function takes as arguments a session handle (hSession), a pointer to the verification mechanism used (pMechanism) and the handle of the verification key (hKey)
    1. Invalid pMechanism
    2. NULL pMechanism
    3. NULL hSession
    4. All arguments set to NULL (hierarchy check)
    5. pMechanism pointing to a mechanism with invalid paramateters
    6. NULL hkey
    7. RSA Private key handle as hKey
- `C_Verify`: taking as arguments a session handle (hSession), a pointer to the data used to produce the signature to verify (pData), its length (ulDataLen), a pointer to the signature (pSignature) and its length (ulSignatureLen)
    1. Both pData and ulDataLen NULL, with a signature of an empty input
    2. Only ulDataLen NULL

    3. Only pData NULL
    4. Buffer too small
    5. NULL hSession
    6. NULL ulSignatureLen
    7. NULL pSignature
    8. Both pSignature and ulSingatureLen NULL
    9. All arguments set to NULL (hierarchy check)

- `C_VerifyUpdate`: its arguments are a session handle (hSession), a pointer to the data block of the data used for produce the signature to verify (pPart) and its length (ulPartLen)
    1. NULL hSession
    2. Both pPart and ulPartLen NULL
    3. Only pPart NULL
    4. Only ulPartLen NULL
    5. All arguments set to NULL (hierarchy check)
- `C_VerifyFinal`: taking as arguments a session handle (hSession), a pointer to the signature (pSignature) and its length (ulSignatureLen)
    1. NULL pSignature
    2. NULL hSession
    3. NULL ulSignatureLen
    4. All arguments set to NULL (hierarchy check)

The stateful test performs, for each operation, the following sequence of functions calls, each of which constitutes a test case :

1. Calling the *init* function with the operation already initialized
2. Calling the single-part operation function during a multi-part operation (after an *update* function call)
3. Calling the *final* function after the invalid single-part operation function call
4. Second call in a row to the single-part operation function without reinitialization
5. Calling the multi-part operation function during a single-part operation
6. Calling the *final* function with the operation not initialized

3.2.2 PIN management functions

We have tested the functions `C_SetPIN`, used to modify the PIN of the user that is currently logged in, and `C_InitPIN`, which initializes the normal user's PIN.

Regarding `C_SetPIN` Cryptoki in section 11.4 states: `C_SetPIN` *can only be called in the "R/W Public Session" state, "R/W SO Functions" state, or "R/W User Functions" state. An attempt to call it from a session in any other state fails with error CKR_SESSION_READ_ONLY.* Since we wanted to prove that the CIE function implementation complies to this requirement, we have implemented a test case calling `C_SetPIN` in a read-only session and normal user logged in.

Regarding `C_InitPIN`, Cryptoki in section 11.4 states: `C_InitPIN` *can only be called in the "R/W SO Functions" state. An attempt to call it from a session in any other state fails with error CKR_USER_NOT_LOGGED_IN.* To verify the compliance to this requirement, we have tried calling `C_InitPIN` with a user not logged in.

3.2.3 Unsupported functions and objects attributes

As said before, some unsupported functions, from their implementation, seem that in some cases they can return another return value rather then CKR_UNSUPPORTED_FUNCTION. In order to verify which functions possess this non-compliance and which do not, we have called every unsupported function, passing invalid attributes, so that there was the possibility to return a different error code.

From the specification of the `C_GetAttributeValue` function we can read: *Note that the error codes CKR_ATTRIBUTE_SENSITIVE, CKR_ATTRIBUTE_TYPE_INVALID, and CKR_BUFFER_TOO_SMALL do not denote true errors for* `C_GetAttributeValue`*. If a call to* `C_GetAttributeValue` *returns any of these three values, then the call must nonetheless have processed every attribute in the template supplied to* `C_GetAttributeValue`*. Each attribute in the template whose value can be returned by the call to* `C_GetAttributeValue` *will be returned by the call to* `C_GetAttributeValue`. We wanted to test this function behaviour, calling it with an unsupported attribute in the supplied template and verifying that all the valid attributes were returned.

The result of these tests are reported in Section 4.

4 Summary of Results

The results of the test identify several issues, which can be divided in the following categories:

- Wrong return value: the function returns an inappropriate code, or it does not respect the hierarchy, although the behaviour can be correct.
- Missing session state checks: required user type and/or read permission checks are not performed.
- Wrong behaviour: a particular behaviour specified by Cryptoki, due to some particular reason, is not respected.
- No checks on NULL arguments: in some cases, if a certain argument is set to null, the function produces an incorrect result, while in other cases the application crashes due to a memory violation. Since Cryptoki specifies a particular error code, CKR_ARGUMETNS_BAD, which generically describes that the supplied arguments are in some way not appropriate, a function in this case should return this code.

The rest of this chapter provides results of the tests in which a non-compliance behaviour occurred. The following tables show the issue, classified in one of the category above, and the relative PKCS#11 broken rule, by directly quoting the documentation, or inferred from the descriptions of the return values.

Function	`C_SetPIN`
PKCS#11 Rule	`C_SetPIN` can only be called in the “R/W Public Session” state, “R/W SO Functions” state, or “R/W User Functions” state. An attempt to call it from a session in any other state fails with error CKR_SESSION_READ_ONLY
Result	In read-only session the function returns CKR_OK, instead of returning CKR_SESSION_READ_ONLY
Classification	Missing session state checks
Impact	The operation can be also performed in read-only sessions

Function	`C_GetAttributeValue`
PKCS#11 Rule	The error codes CKR_ATTRIBUTE_SENSITIVE, CKR_ATTRIBUTE_TYPE_INVALID, and CKR_BUFFER_TOO_SMALL do not denote true errors for `C_GetAttributeValue`. If a call to `C_GetAttributeValue` returns any of these three values, then the call must nonetheless have processed every attribute in the template supplied to `C_GetAttributeValue`. Each attribute in the template whose value can be returned by the call to `C_GetAttributeValue` will be returned by the call to `C_GetAttributeValue`
Result	All the attributes listed after an invalid attribute in the template are not returned
Classification	Wrong behaviour
Impact	The operation is partially performed

Function	`C_DigestInit`
PKCS#11 Rule	If the pMechanism is a null pointer the function should return CKR_ARGUMENTS_BAD or CKR_MECHANISM_INVALID
Result	Memory access violation
Classification	No checks on NULL arguments
Impact	Application crashes with error code 0xc0000005

Function	`C_Digest`
PKCS#11 Rule	If pData is set to NULL, while ulDataLen not, the function should return CKR_ARGUMENTS_BAD
Result	Memory access violation
Classification	No checks on NULL arguments
Impact	Application crashes with error code 0xc0000005

Function	`C_Digest`
PKCS#11 Rule	if ulDataLen and is set to NULL, while pData not, the function should return CKR_ARGUMENTS_BAD
Result	Function returns CKR_OK
Classification	No checks on NULL arguments
Impact	Function computes a wrong digest

Function	`C_Digest`
PKCS#11 Rule	if ulDigestLen and is set to NULL, while pDigest not, the function should return CKR_ARGUMENTS_BAD
Result	Memory access violation
Classification	No checks on NULL arguments
Impact	Application crashes with error code 0xc0000005

Function	`C_Digest`
PKCS#11 Rule	A call to `C_Digest` always terminates the active digest operation unless it returns CKR_BUFFER_TOO_SMALL
Result	Function returning CKR_SESSION_HANDLE_INVALID does not terminate the digest operation
Classification	Wrong behaviour
Impact	Unexpected operation state

Function	`C_DigestUpdate`
PKCS#11 Rule	A call to `C_DigestUpdate` which results in an error terminates the current digest operation
Result	Function returning CKR_SESSION_HANDLE_INVALID does not terminate the digest operation
Classification	Wrong behaviour
Impact	Unexpected operation state

Function	`C_DigestUpdate`
PKCS#11 Rule	If pPart is set to NULL, while ulPartLen not, function should return CKR_ARGUMENTS_BAD
Result	Memory access violation
Classification	No checks on NULL arguments
Impact	Application crashes with error code 0xc0000005

Function	`C_DigestUpdate`
PKCS#11 Rule	If ulPartLen is set to NULL, while pPart not, function should return CKR_ARGUMENTS_BAD
Result	Function returns CKR_OK
Classification	No checks on NULL arguments
Impact	Function computes a wrong digest

Function	`C_DigestFinal`
PKCS#11 Rule	If ulDigestLen is set to NULL, while pDigest not, function should return CKR_ARGUMENTS_BAD
Result	Memory access violation
Classification	No checks on NULL arguments
Impact	Application crashes with error code 0xc0000005

Function	`C_DigestFinal`
PKCS#11 Rule	A call to `C_SignFinal` always terminates the active signing operation unless it returns CKR_BUFFER_TOO_SMALL
Result	Function returning CKR_SESSION_HANDLE_INVALID does not terminate the digest operation
Classification	Wrong behaviour
Impact	Unexpected operation state

Function	`C_SignInit`
PKCS#11 Rule	If the pMechanism is a null pointer the function should return CKR_ARGUMENTS_BAD or CKR_MECHANISM_INVALID
Result	Memory access violation
Classification	No checks on NULL arguments
Impact	Application crashes with error code 0xc0000005

Function	`C_SignInit`
PKCS#11 Rule	Function should return CKR_KEY_HANDLE_INVALID if the key handle is invalid, for example if an handle of an object that is not a key. Instead, if an attempt has been made to use a key for a cryptographic purpose, but the key's attributes are not set to allow it, the function should return CKR_KEY_FUNCTION_NOT_PERMITTED
Result	Calling the function with the public key, which has not the attribute CKA_SIGN set to true, the value returned is CKR_KEY_HANDLE_INVALID instead of the correct one
Classification	Wrong return value
Impact	Incorrect key usage

Function	`C_Sign`
PKCS#11 Rule	If pData is set to NULL, while ulDataLen is not, the function should return CKR_ARGUMENTS_BAD
Result	Memory access violation
Classification	No checks on NULL arguments
Impact	Application crashes with error code 0xc0000005

Function	`C_Sign`
PKCS#11 Rule	If ulDataLen is set to NULL, while pData is not, the function should return CKR_ARGUMENTS_BAD
Result	Function returns CKR_OK
Classification	No checks on NULL arguments
Impact	Function computes wrong singature

Function	`C_Sign`
PKCS#11 Rule	If the output of the function is too large to fit in the supplied buffer the function should return CKR_BUFFER_TOO_SMALL
Result	Calling the function with an output buffer too small it returns CKR_GENERAL_ERROR
Classification	Wrong return value
Impact	Incorrect error handling

Function	`C_Sign`
PKCS#11 Rule	A call to C_Sign always terminates the active signing operation unless it returns CKR_BUFFER_TOO_SMALL
Result	Function returning CKR_SECTION_HANDLE_INVALID does not terminate the signature operation
Classification	Wrong behaviour
Impact	Unexpected operation state

Function	`C_Sign`
PKCS#11 Rule	If pulSignatureLen is set to NULL, while pSignature is not, the function should return CKR_ARGUMENTS_BAD
Result	Memory access violation
Classification	No checks on NULL arguments
Impact	Application crashes with error code 0xc0000005

Function	`C_SignUpdate`
PKCS#11 Rule	A call to `C_SignUpdate` which results in an error terminates the current signature operation
Result	Function returning CKR_SESSION_HANDLE_INVALID does not terminate the signature operation
Classification	Wrong behaviour
Impact	Unexpected operation state

Function	`C_SignUpdate`
PKCS#11 Rule	If pPart is set to NULL, while ulPartLen is not, the function should return CKR_ARGUMENTS_BAD
Result	Memory access violation
Classification	No checks on NULL arguments
Impact	Application crashes with error code 0xc0000005

Function	`C_SignUpdate`
PKCS#11 Rule	If ulPartLen is set to NULL, while pPart is not, the function should return CKR_ARGUMENTS_BAD
Result	Function returns CKR_OK
Classification	No checks on NULL arguments
Impact	Function computes a wrong signature

Function	`C_SignFinal`
PKCS#11 Rule	If the output of the function is too large to fit in the supplied buffer the function should return CKR_BUFFER_TOO_SMALL
Result	Calling the function with an output buffer too small it returns CKR_GENERAL_ERROR
Classification	Wrong return value
Impact	Incorrect error handling

Function	`C_SignFinal`
PKCS#11 Rule	If pulSignature is set to NULL, while pSignature is not, the function should return CKR_ARGUMENTS_BAD
Result	Memory access violation
Classification	No checks on NULL arguments
Impact	Application crashes with error code 0xc0000005

Function	`C_SignFinal`
PKCS#11 Rule	A call to C_SignFinal always terminates the active signing operation unless it returns CKR_BUFFER_TOO_SMALL
Result	Function returning CKR_SESSION_HANDLE_INVALID does not terminate the signature operation
Classification	Wrong behaviour
Impact	Unexpected operation state

Function	`C_VerifyInit`
PKCS#11 Rule	If pData is set to NULL, while ulDataLen is not, the function should return CKR_ARGUMENTS_BAD
Result	Memory access violation
Classification	No checks on NULL arguments
Impact	Application crashes with error code 0xc0000005

Function	`C_VerifyInit`
PKCS#11 Rule	Function should return CKR_KEY_HANDLE_INVALID if the key handle is invalid, for example if an handle of an object that is not a key. Instead, if an attempt has been made to use a key for a cryptographic purpose, but the key's attributes are not set to allow it, the function should return CKR_KEY_FUNCTION_NOT
Result	Calling the function with the private key, which has not the attribute CKA_VERIFY set to true, the value returned is CKR_KEY_HANDLE_INVALID instead of the correct one
Classification	Wrong return value
Impact	Incorrect key usage

Function	`C_Verify`
PKCS#11 Rule	If pData is set to NULL, while ulDataLen is not, the function should return CKR_ARGUMENTS_BAD
Result	Memory access violation
Classification	No checks on NULL arguments
Impact	Application crashes with error code 0xc0000005

Function	`C_Verify`
PKCS#11 Rule	If ulDataLen is set to NULL, while pData is not, the function should return CKR_ARGUMENTS_BAD
Result	Function computes the operation
Classification	No checks on NULL arguments
Impact	Function returns CKR_SIGNATURE_INVALID

Function	`C_Verify`
PKCS#11 Rule	A call to `C_Verify` always terminates the active verification operation
Result	Function returning CKR_SESSION_HANDLE_INVALID does not terminate the signature operation
Classification	Wrong behaviour
Impact	Unexpected operation state

Function	`C_Verify`
PKCS#11 Rule	If pulSignatureLen is set to NULL, while pSignature is not, the function should return CKR_ARGUMENTS_BAD
Result	Function computes the operation
Classification	No checks on NULL arguments
Impact	Function returns CKR_SIGNATURE_LEN_RANGE

Function	`C_Verify`
PKCS#11 Rule	If pSignature is set to NULL, while pulSignatureLen is not, the function should return CKR_ARGUMENTS_BAD
Result	Memory access violation
Classification	No checks on NULL arguments
Impact	Application crashes with error code 0xc0000005

Function	`C_VerifyUpdate`
PKCS#11 Rule	A call to `C_VerifyUpdate` which results in an error terminates the current verification operation
Result	Function returning CKR_SESSION_HANDLE_INVALID does not terminate the signature operation
Classification	Wrong behaviour
Impact	Unexpected operation state

Function	`C_Verify`
PKCS#11 Rule	If pPart is set to NULL, while ulPartLen is not, the function should return CKR_ARGUMENTS_BAD
Result	Memory access violation
Classification	No checks on NULL arguments
Impact	Application crashes with error code 0xc0000005

Function	`C_VerifyUpdate`
PKCS#11 Rule	If ulPartLen is set to NULL, while pPart is not, the function should return CKR_ARGUMENTS_BAD
Result	Function returns CKR_OK
Classification	No checks on NULL arguments
Impact	Function computes wrong signature verification

Function	`C_Verify`
PKCS#11 Rule	If pSignature is set to NULL, while ulSignatureLen is not, the function should return CKR_ARGUMENTS_BAD
Result	Memory access violation
Classification	No checks on NULL arguments
Impact	Application crashes with error code 0xc0000005

Function	`C_VerifyUpdate`
PKCS#11 Rule	A call to C_VerifyFinal always terminates the active verification operation
Result	Function returning CKR_SESSION_HANDLE_INVALID does not terminate the signature operation
Classification	Wrong behaviour
Impact	Unexpected operation state

Analysing the behaviour of SoftHSM performing these previous tests, we have noticed that in those cases in which the CIE module crashes or performs incorrect computations due to NULL arguments values, SoftHSM returns CKR_ARGUMENTS_BAD, suggesting that this is the correct behaviour.

Regarding the stateful test, the following issues are emerged:

1. For all the three operations, CIE library does not check if the operation is performing in multi-part or in single-part. Indeed, a call to the single-part operation function, i.e. `C_Digest`, `C_Sign` and `C_Verify`, during a multi-part operation returns CKR_OK, instead of CKR_OPERATION_ACTIVE.
2. For signature and signature verification operation, the *final* function does not check if the operation is initialized, causing the application to crash due to a memory violation. While in digesting operation the function correctly returns CKR_OPERATION_NOT_INITIALIZED.
3. `C_Digest` does not check if the operation is initialized, causing the application to crash due to a memory violation. While the other two operation functions return CKR_OPERATION_NOT_INITIALIZED.
4. Only digesting operation function `C_DigestUpdate` checks if the operation is initialized, while the *update* function of the other two operations does not, causing the application to crash due to a memory violation.
5. The same issue described above is applied to the *final* functions.

Referring to the triage of operation, even if not specifically related to non-compliance issues, the test has found a number of problems due to memory access violation. The reason is that the library does not check if the arguments are well-formed. All the functions, except for the initialization ones, require the size of the input/output data, and if the specified size is smaller or bigger then the real size, the function will execute returning CKR_OK, possibly letting a memory access violation occur, causing heap corruptions and wrong output computation.

In conclusion, this thesis has examined the compliance of the CIE PKCS#11 module with the reference standard, through a series of tests. The results obtained highlight a frequent discrepancy between the module behaviour and the standard specifications, raising some concerns about its reliability and security. The anomalies found, ranging from memory management errors to missing requirements checks, suggest the need for a corrective intervention. Such intervention should aim to ensure full compliance of the module with the PKCS#11 standard, in order to improve the reliability of operations involving the CIE.

Bibliography

[1] https://www.cartaidentita.interno.gov.it. Accessed 12/02/2025.

[2] App CieID. https://www.cartaidentita.interno.gov.it/info-utili/cie-id/. Accessed 14/02/2025.

[3] Ewent NFC smart card reader. https://www.ewent-eminent.com/it/prodotti/143-connettivit%C3%A0/lettore-nfc-di-smart-card-%7C-cie-30. Accessed 03/02/2025.

[4] Middleware CSP-PKCS#11 per la CIE 3.0 (Windows). https://github.com/italia/cie-middleware. Accesed 14/02/2025.

[5] PC/SC Workgroup. https://pcscworkgroup.com/. Accessed 14/02/2025.

[6] SoftHSM. https://www.softhsm.org/. Accessed 21/02/2025.

[7] Winscard module. http://msdn.microsoft.com/en-us/library/ms924513.aspx. Accessed 14/02/2025.

[8] docs italia. Carta d'Identità Elettronica CIE 3.0 – Specifiche Chip. https://www.cartaidentita.interno.gov.it/downloads/2021/03/cie_3.0_-_specifiche_chip.pdf. Accessed 28/01/2025.

[9] docs italia. Manuale operativo per servizi di accesso pinless basati sulla CIE. https://docs.italia.it/italia/cie/cie-accessi-pinless-manuale-docs/it/stabile/index.html. Accessed 12/02/2025.

[10] docs italia. Manuale Tecnico per gli erogatori di servizi pubblici e privati. https://docs.italia.it/italia/cie/cie-manuale-tecnico-docs/it/master/index.html. Accessed 28/01/2025.

[11] docs italia. Software CIE - Manuale Utente. https://docs.italia.it/italia/cie/cie-middleware-windows-docs/it/master/index.html. Accessed 12/02/2025.

[12] GIXEL. IAS ECC - Identification Authentication Signature European Citizen Card. Technial Specifications Revision: 1.0.1.

[13] ICAO. Doc 9303 - Machine Readable Travel Documents. https://www.icao.int/publications/pages/publication.aspx?docnum=9303. Accessed 03/02/2025.

[14] RSA Laboratories. PKCS11 v2.11: Cryptographic Token Interface Standard. https://cryptsoft.com/pkcs11doc/STANDARD/pkcs-11v2-11r1.pdf. Accessed 28/01/2025.

[15] Graham Steel St´ephanie Delaune, Steve Kremer. Formal Analysis of PKCS#11.