UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Sander-Karl Kivivare

Secure Channel Establishment for the NFC Interface of the New Generation Estonian ID Cards

Bachelor's Thesis (9 ECTS)

Supervisor: Arnis Paršovs, MSc

Tartu 2020

Secure Channel Establishment for the NFC Interface of the New Generation Estonian ID Cards

Abstract:
The latest generation Estonian ID card introduced in the December 2018 has a contactless interface that can be used to communicate with the card via near-field communication (NFC). This thesis describes the cryptographic protocol that is used to communicate over the contactless interface and provides detailed instructions with code examples in Python to help software developers to create applications that can make use of this new NFC interface on Estonian ID card.

Keywords:
Estonian ID card, EstEID, NFC, PACE

CERCS:
P170 Computer science, numerical analysis, systems, control

Turvalise ühenduse loomine NFC liidesega uue põlvkonna Eesti ID-kaartidel

Lühikokkuvõte:
Alates 2018. aasta detsembrist väljastatud viimase põlvkonna Eesti ID-kaartidel on kontaktivaba liides, mida saab kasutada lähiväljaside (NFC) kaudu. Käesolev töö annab ülevaate krüptograafilisest protokollist, mis on vajalik kontaktivaba liidese kasutamiseks, ning juhiseid selle liidese kasutamiseks koos Pythoni koodinäidetega, aitamaks kaasa uute rakenduste loomisele, mis kasutaks Eesti ID-kaardi NFC liidest.

Võtmesõnad:
Eesti ID-kaart, EstEID, NFC, PACE

CERCS:
P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

Abbreviations and Acronyms

AES	Advanced Encryption Standard
AID	Application Identifier
APDU	Application Protocol Data Unit
ASN.1	Abstract Syntax Notation One
C-APDU	Command Application Protocol Data Unit
CAN	Card Access number
CBC	Cipher block chaining
CMAC	Cipher-based Message Authentication Code
ECB	Electronic codebook
ECDH	Elliptic Curve Diffie–Hellman
IV	Initialization Vector
MAC	Message Authentication Code
NFC	Near Field Communication
PACE	Password Authenticated Connection Establishment
R-APDU	Response Application Protocol Data Unit
SW	Status Word
TLV	Tag(Type)-Length-Value

1 Introduction

With the introduction of the new generation Estonian ID card at the end of 2018 [1], a contactless interface was introduced for the card. Since then, no updates or new software has been released to support the new mode of communicating with the card. While the official card documentation [2] mentions the specification of the interface and there are claims, that the Estonian ID card functionality is also available over the contactless interface [3], no practical guidelines on implementing the communication over contactless interface currently exist.

This thesis aims to fill this gap by providing a technical documentation of the NFC communication interface. The thesis is aimed at software developers who are interested in implementing applications, that are able to communicate with the Estonian ID card over the NFC interface. Since most of the modern smartphones have a built-in NFC reader, this opens up the possibility to use the latest generation Estonian ID card with the applications running on the mobile devices.

The thesis is structured as follows. Section 2 provides background information on smart cards and the NFC-compatible Estonian ID card. Section 3 provides a high-level overview of the cryptographic protocol that has to be used to communicate with the Estonian ID card over the NFC interface. Section 4 provides a detailed walk-through with code examples in Python on how to use the Password Authenticated Connection Establishment (PACE) protocol with the card and exchange commands with the card over secure messaging channel. Section 5 provides performance analysis for the communication over the NFC interface in comparison to the communication over the standard contact interface and in Section 6, an overview of software implementations is provided. Finally, Section 7 concludes the thesis.

2 Background

This section gives a technical background of smart cards and the new generation of Estonian ID card. The overview of smart cards is based on ISO/IEC 7816 standard for integrated chip cards [4], as the platform of Estonian ID card is also compliant with this standard [2].

2.1 Smart cards

Smart Cards are usually credit card sized plastic cards, that can store data and execute programs stored on them. They are used for example in identification and financial solutions like national ID cards and bank cards.

Smart cards can have contact or contactless interface or both, depending on the implementation. For contact reading, metal pads on the chip are used for data and power transmission, while the contactless mode uses NFC for this purpose. For dual-interface cards, the same chip is connected to contact and contactless interface. For communicating with the card, a reader is required. The main benefit of the contactless mode is the possibility of reading the card from a short distance, requiring the card to not be inserted in to the reader.

Near Field Communication is a wireless communication technology, with a working distance of about 10 cm, depending on the application [5]. The data and power is transferred via an electromagnetic field between two antennas. For contactless smart cards, this field is generated by the NFC card reader, and the chip of the smart card uses that as the power source. Most smartphones today are also equipped with built in NFC readers and can therefore be used to conveniently read contactless cards.

For communicating with smart cards, structured byte arrays called Application Protocol Data Units are used. APDUs are further classified as Command APDUs (C-APDUs) and Response APDUs (R-APDUs) and their basic structure is shown in Table 1. Command APDUs have a mandatory header, that contains class, instruction and parameter bytes, and an optional body, containing data that is sent to the card. Data for APDUs is usually structured in TLV format – tag (T) describing the type of value, followed by the length (L) of the value and the value (V) itself. Response APDUs have two mandatory status bytes (status word SW) that can be preceded by additional data. Successful processing of the command is usually indicated by the status bytes `9000`, while other values usually indicate warnings and errors.

Smart cards usually implement a file system, that consists of three types of files: Master File (MF), that is the root directory for the file system, Dedicated File (DF), that is essentially a folder, and Elementary File (EF), that is used as a basic data storage element. Two-byte file identifier (FID) is used for addressing the files.

Table 1. Structure of command and response APDUs

Command APDU						
Header				**Body**		
CLA	INS	P1	P2	Lc	Data	Le
Response APDU						
Data					SW1	SW2

Secure messaging can be used to ensure privacy and integrity of the APDUs exchanged between the card and the reader. Secure messaging works by encrypting the plain data exchanged and providing mechanisms to authenticate the messages.

2.2 Estonian ID card

The Estonian identity card (ID card) is an identity document issued by the Estonian state with digital functionality, enabling electronical identification and signing of documents [6]. The Estonian state also issues other types of identity documents, that contain a contact-type smart card chip and provide cryptographic functionality. In the context of this thesis, the common term "ID card" is used to refer to these identity documents.

Figure 1. New generation Estonian identity card issued starting December 2018 [7]. Card access number (CAN) is highlighted in red

In December 2018, a new generation of Estonian ID cards manufactured by IDEMIA were introduced [8]. These new cards are NFC-capable (dual interface) and can potentially be used in both contact and contactless modes [2]. Besides the introduction of the new interface, the cards were also redesigned and a colour photo of the cardholder has been added. Also new, is a six digit number, that has been printed on the front side of the card under the cardholder's photo (see Figure 1). This is the Card Access Number (CAN), that must be used to establish communication with the ID card over the new contactless

interface. When viewing the new card with a flashlight, the wiring required for NFC can be seen, with the antennas being located around the edges of the card (Figure 2).

Figure 2. Antenna wiring for the NFC interface on the new generation ID card

To use the contactless interface, a secure channel must be established and used for exchanging commands [2]. Although the cryptographic parameters that are used for establishing the secure channel are described in the official documentation [2], no examples of using the contactless interface have been provided by the authorities. The following sections of this thesis describe the establishment of the required secure channel and its usage in more detail.

The idea of a contactless interface for the Estonian ID cards had been explored before the release of this generation's cards. A prototype contactless card was built for the previous generation ID card, allowing its users to authenticate and give digital signatures [9]. Unfortunately, this functionality has not been provided for public use. The feasibility of using an NFC based smart card solution using smartphones for authenticating to websites has also been analysed and concluded that a similar approach could be used for Estonian ID cards, if an NFC interface was to be added [10].

The only research on the current generation of ID cards regarding NFC is about its use for third party applications, that could be uploaded to the card and used in contactless mode [11].

3 Security mechanisms

Due to privacy concerns, the electronic functionality of the Estonian ID card is accessible over the NFC interface only after a secure channel based on the password known to the card and terminal has been established. It is not clear how much privacy this security measure provides, as the password required to establish the connection is in the form of a 6-digit Card Access Number, that is printed on the card and is clearly visible to anyone who sees the front of the ID card. This would, however, prevent the trivial attack where the attacker, who has not seen the card, tries to establish a connection with the card while the card is in the victim's pocket.

To implement this password-authenticated secure channel, the Estonian ID card uses a protocol that is widely used in the Machine Readable Travel Documents (MRTDs) also known as biometric passwords or ePassports. The protocol is described in the standard "ICAO Doc 9303 Part 11 - Security Mechanisms for MRTDs" [12], therefore this thesis will follow the ICAO specification, but focusing on the protocol features that are supported specifically by the Estonian ID card.

The subsections below briefly introduce cryptographic primitives that are used by the protocol and provide a high-level overview of the Password Authenticated Connection Establishment (PACE) protocol, that is used to establish a shared secret between the card and the terminal based on the password known by both parties.

3.1 Cryptographic primitives

The PACE protocol is based on several symmetric and asymmetric cryptographic primitives. For better understanding of the security mechanisms, a brief overview of these primitives is provided below.

Elliptic Curve cryptography (ECC). ECC uses the multiplication of elliptic curve point, to generate the necessary hard to solve problem suitable for public-key cryptography. An elliptic curve key pair consists of a secret key, which is an integer, and an elliptic curve point, calculated by multiplying a base point (generator) with the secret key [13]. The curve parameters and the base point used have to be shared by the parties. Standard curves have been established to simplify sharing the parameters and ensure the curve has suitable cryptographic properties. The curve parameters and the base point used in Estonian ID card for PACE is NIST P-256, defined by U.S. National Institute of Standards and Technology [2].

Elliptic Curve Diffie-Hellmann (ECDH). For establishing a shared secret over an insecure channel, elliptic curve Diffie-Hellman key exchange algorithm is used [13]. Two parties, in this case the card and the terminal, generate an elliptic curve key pair

and exchange public keys. The received public key is multiplied with the secret key and usually the x-coordinate of the resulting point is used as the shared secret.

Symmetric encryption. Symmetric encryption uses a single secret key, shared between the parties, to both encrypt and decrypt data [14]. Different symmetric encryption schemes exist, but in the context of the Estonian ID card the block cipher AES (Advanced Encryption Standard) is used. To prevent the same input being encrypted to the same output, cipher block chaining (CBC) mode of encryption with an initialisation vector (IV) is used.

Hash functions. Hash functions are functions, that take in an arbitrary sized string and output a string of fixed length. For cryptographic use, the hash functions used need to be one-way, meaning that it should be infeasible to calculate the input value from the output, and collision resistant, making it practically impossible to find two different inputs that result in the same output [14]. For the implementation of a secure channel for the Estonian ID card, the Secure Hash Algorithms (SHA) family of hash functions is used.

Message Authentication Code (MAC). MAC is used to generate a tag, that can be used to verify the authenticity of the message. MAC is generated by using a symmetric key that is shared between the parties. This tag can be then used to validate that the message has not been modified. [14]

3.2 Password Authenticated Connection Establishment

PACE protocol is used to establish a secure connection over insecure channel using a shared password. PACE protocol defines four steps (see Figure 3), that need to be performed by the chip and terminal to establish a common secret key and derive session keys based on this secret [15]. As the first step, a nonce is exchanged. The nonce (random number) is selected by the chip and encrypted using a symmetric key derived from a password known to both parties. The nonce is then mapped using ECDH to a new elliptic curve point. The mapping is performed by multiplying the curve standard base point with the nonce and adding the secret point calculated using ECDH. This mapped point is used as the base point for the next step.

As the third step, key agreement is performed. Using the mapped base point ECDH key exchange is performed and a shared secret established. Finally, the chip and the terminal derive session keys (`Kenc` and `Kmac`) from the shared secret. To authenticate the established keys, a MAC is calculated for the public keys received in the previous step. These MAC values are exchanged and verified, concluding the PACE protocol.

Chip		Terminal
Step 1: Exchanging nonce		
Derive key from CAN $K = Hash(CAN\|\|3)$		Derive key from CAN $K = Hash(CAN\|\|3)$
Choose randon nonce s Compute $z = Enc_K(s)$		
	$z \longrightarrow$	
		Compute $s = Dec_K(z)$
Step 2: Nonce mapping		
Generate ephemeral EC key pair $(Chip_{priv'}, Chip_{pub'})$		Generate ephemeral EC key pair $(Terminal_{priv'}, Terminal_{pub'})$
	$\longleftarrow Terminal_{pub'}, Chip_{pub'} \longrightarrow$	
Calculate ECDH shared secret $H = Terminal_{pub'} \times Chip_{priv'}$		Calculate ECDH shared secret $H = Chip_{pub'} \times Terminal_{priv'}$
Map new EC base point $G' = G \times s + H$		Map new EC base point $G' = G \times s + H$
Step 3: Key agreement		
Generate ephemeral EC key pair using G' $(Chip_{priv}, Chip_{pub})$		Generate ephemeral EC key pair using G' $(Terminal_{priv}, Terminal_{pub})$
	$\longleftarrow Terminal_{pub}, Chip_{pub} \longrightarrow$	
Calculate ECDH shared secret $X = Terminal_{pub} \times Chip_{priv}$		Calculate ECDH shared secret $X = Chip_{pub} \times Terminal_{priv}$
Step 4: Mutual authentication		
Derive K_{enc} and K_{mac} $K_{enc} = Hash(X\|\|1)$ $K_{mac} = Hash(X\|\|2)$		Derive K_{enc} and K_{mac} $K_{enc} = Hash(X\|\|1)$ $K_{mac} = Hash(X\|\|2)$
$Chip_{MAC} = MAC_{K_{mac}}(Terminal_{pub})$		$Terminal_{MAC} = MAC_{K_{mac}}(Chip_{pub})$
	$\longleftarrow Terminal_{MAC}, Chip_{MAC} \longrightarrow$	
Verify $Terminal_{MAC}$		Verify $Chip_{MAC}$

Figure 3. Overview of the PACE protocol

For the Estonian ID card, the CAN printed on the card is used as a password for PACE [2]. One of the main advantages of using PACE is that the strength of keys that are generated for the secure session are not dependant on the complexity of the password, allowing a short 6-digit CAN to be used as a password [15]. The next section describes in detail how to perform the PACE with the Estonian ID card and how to use the established secure connection for sending APDUs.

4 Establishing secure channel

This section covers the implementation of establishing the secure channel with the Estonian ID card. It focuses on the specific PACE implementation used by the card and is based on ICAO Doc 9303 Part 11 [12]. A step-by-step description of performing the PACE and sending the commands over secure channel is given along with code examples in Python 3.

Although only required when using NFC interface, a secure channel can also be established in contact mode. The implemented PACE functionality was tested in both modes and no differences in establishing and using the secure channel were observed.

4.1 PACE

To establish PACE, first the IAS ECC applet has to be selected. This applet provides electronic functionality for the Estonian ID card. It is selected by sending the `SELECT FILE` command specifying the applet identifier (AID) of IAS ECC applet (see Table 2). The AID of the IAS ECC applet is `A0000000770108000700 00FE00000100`.

Table 2. APDU for selecting IAS ECC applet

C-APDU	00 A4 04 00 10 A0 00 00 00 77 01 08 00 07 00 00 FE 00 00 01 00
R-APDU	90 00

Python library `pyscard` can be used for exchanging the APDUs with the card. Listing 1 shows an example for IAS ECC applet selection. The APDUs are sent as a list containing byte values using the `transmit()` method. The response data is returned as a list of byte values with separate SW1 and SW2 status word values.

```
from smartcard.CardRequest import CardRequest

channel = CardRequest().waitforcard().connection
channel.connect()

AID = bytes.fromhex("A0000000770108000700 00FE00000100")
data, sw1, sw2 = channel.transmit([0x00, 0xA4, 0x04, 0x00, len(AID)] + [b for b in AID]
     + [0x00])
```

Listing 1. Sending APDUs in Python

After selecting the applet, a series of `GENERAL AUTHENTICATE` commands are exchanged to perform PACE. Data for those commands is wrapped in so called `Dynamic Authentication Data` objects, that contain values used for the protocol. These values contained in the object are different for each step and are identified by a specific tag.

4.1.1 Reading PACE parameters from EF.CardAccess file

Details about the PACE protocol parameters supported by the applet are stored in `EF.CardAccess` file. The reading of this file is not required and can be skipped, as the parameters used on all the currently issued ID cards are fixed. The `EF.CardAccess` file (FID: `011C`) is located under the Master File of the IAS ECC applet. It contains DER-encoded ASN.1 `SecurityInfos` object, which is defined in Section 9.2 of ICAO Doc 9303 Part 11 [12]. The `SecurityInfos` object contains information about security protocols supported by the card.

The `EF.CardAccess` file is the only file, that can be read in contactless mode without first establishing a secure channel [2]. The commands used to read the file are based on examples from the developer's guide for the Estonian ID card [16] and are provided in Table 3. The `EF.CardAccess` file is selected by issuing `SELECT FILE` command with FID of `011C`. To read the contents of the file, the `READ BINARY` command is sent, with the number of bytes to read left unspecified so all the available bytes are read. The file contains DER-encoded `SecurityInfos` object padded to 64 bytes[1].

Table 3. APDUs for reading `EF.CardAccess` file

SELECT FILE 01 1C	
C-APDU	00 A4 01 0C 02 **01 1C**
R-APDU	90 00
READ BINARY	
C-APDU	00 B0 00 00 00
R-APDU	31 14 30 12 06 0A 04 00 7F 00 07 02 02 04 02 04 02 01 02 02 01 0C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 90 00

Figure 4 shows the decoded ASN.1 structure of the EF.CardAccess file. The `SecurityInfos` structure (see Figure 4) in `EF.CardAccess` file on the Estonian ID card contains only one `PACEInfo` object, specifying the cryptographic parameters that have to be used to establish PACE. According to the ICAO Doc 9303 Part 11, the object identifier (OID) encoded in the field `protocol` corresponds to `id-PACE-ECDH-GM-AES-CBC-CMAC-256`, and the value 12 specified in the `parameterId` field, corresponds to NIST P-256[2], which is the curve that is used for elliptic curve operations during PACE.

PACE protocol parameters for Estonian ID card are identified by the object identifier (OID) `id-PACE-ECDH-GM-AES-CBC-CMAC-256`. ECDH is the algorithm used for key

[1]The official ID card card documentation [2] incorrectly specifies the file size as 48 bytes.

[2]The ID card documentation [2] incorrectly identifies the curve as BrainpoolP384r1.

```
// SecurityInfos
SET {
    // PACEInfo
    SEQUENCE {
        // protocol
        OBJECT IDENTIFIER 0.4.0.127.0.7.2.2.4.2.4
        // version
        INTEGER 2
        // parameterId
        INTEGER 12
    }
}
```

Figure 4. Decoded ASN.1 from `EF.CardAccess`

agreement, GM stands for generic mapping that is used for mapping the nonce for ECDH. AES in CBC mode is used for encrypting and decrypting the data and CMAC for calculating MAC values, with key length of 256 bits.

4.1.2 Protocol initiation

To initiate the PACE protocol, the DER-encoded OID of the PACE cryptographic parameters is sent to the card along with the value identifying the password that should be used by the card to encrypt the nonce. As CAN is used to establish PACE with the Estonian ID card, the password identifier must be set to `02`. This information is sent to the card with command `MANAGE SECURITY ENVIRONMENT: SET AUTHENTICATION TEMPLATE` (Table 4).

Table 4. Command MSE: SET AT

MSE: SET AT				
C-APDU				
CLA	00		Plain	
INS	22		Manage security environment	
P1/P2	C1A4		Set authentication template	
Lc	0F		Length of data	
Data	**Tag**	**Length**	**Value**	**Comment**
	80	0A	04007F00070202040204	OID
	83	01	02	CAN
Le	00			Excpected length of response
R-APDU				
SW	9000			

4.1.3 Getting and decrypting the nonce

The encrypted nonce is obtained from the card by sending `GENERAL AUTHENTICATE` command (Table 5) with an empty `Dynamic Authentication Data` object. The nonce is a random 32-byte value that is encrypted by the card using AES-256 cipher in CBC mode with IV of zero.

Table 5. Command for obtaining encrypted nonce from the card

GET NONCE				
C-APDU				
CLA	10		Plain, command chaining	
INS	86		General Authenticate	
P1/P2	0000			
Lc	02		Length of data	
Data	**Tag**	**Length**	**Value**	**Comment**
	7C	00	-	Dynamic Authentication Data
Le	00			Excpected length of response
R-APDU				
Data	**Tag**	**Length**	**Value**	**Comment**
	7C	22	-	Dynamic Authentication Data
	80	20	4D825519E561FCA5ADCD B272D9195B229DA83F21 0F04ABE7310CDAF84BBB 245E	Encrypted Nonce
SW	9000			

The key for decrypting the nonce is generated by appending to the CAN a 4-byte value encoding integer 3, and hashing it using SHA-256 hash function. The obtained 32-byte hash value is used as a symmetric AES decryption key. If the response to this C-APDU is not returned in more than a second a delay is introduced which the card has enabled to slow down CAN brute force attacks (see Section 4.2). Listing 2 shows key derivation and nonce decryption using Python. For SHA-256 calculation Python built-in `hashlib` library is used and for AES the `PyCryptodome` library [17] is used.

```
import hashlib
from Cryptodome.Cipher import AES

decryption_key = hashlib.sha256(b"050746" + b"\x00\x00\x00\x03").digest()

iv = 16 * b'\x00'
aes = AES.new(decryption_key, AES.MODE_CBC, iv)
decrypted_nonce = aes.decrypt(encrypted_nonce)
```

Listing 2. Key derivation and nonce decryption in Python

The following values were used and obtained in the calculation:
Encrypted nonce: 4D825519E561FCA5ADCDB272D9195B22
9DA83F210F04ABE7310CDAF84BBB245E
CAN: 050746
Derived AES key: ADDB50CF5219B5224BA056BD50DE1854
02A500F48D82EF65922DF4E0C5598F69
Decrypted nonce: B307B476AFF79128778BC400939D9507
D0344088937EE47DCFDC86D9E5D0DABB

4.1.4 Nonce mapping

In this protocol step, the card and the terminal have to agree on a new EC base point that will be used for ECDH key agreement in the next step. This process is called nonce mapping. To perform this mapping, both the card and terminal generate an ephemeral EC key pair and exchange the public keys using GENERAL AUTHENTICATE command (Table 6). The elliptic curve point used as the public key is exchanged in the uncompressed form. The first byte is 04, indicating uncompressed point, followed by 32 bytes of x-coordinate and 32 bytes of y-coordinate of the point.

EC key pair generation is shown on Listing 3. The private key is generated by obtaining 32 random bytes using Python built-in os library. The integer value of the private key is multiplied with the standard base point for the NIST P-256 curve to calculate the public key. For obtaining the curve parameters and performing necessary elliptic curve operations, Python ECDSA library [18] is used.

```
import os
from ecdsa import NIST256p

private_key = int.from_bytes(os.urandom(32), 'big')

base_point = NIST256p.generator.to_affine()
public_key = base_point * private_key

public_key_x_coordinate = public_key.x().to_bytes(32, 'big')
public_key_y_coordinate = public_key.y().to_bytes(32, 'big')
```

Listing 3. EC key pair generation in Python

The following values were used and obtained in the calculation:
Terminal's private key: 1F5E08DECE8CD745DFF7686357E99236
6851E123ED80D89224620CE6E451F4FB
Terminal's public key (x-coordinate): 8B50756DE011B9C6C7F53E31AE360821
01687F8DEDEF7C193FEF365452FB922B
Terminal's public key (y-coordinate): 82CB18DC26CFD33EA17434C2C17B7EEC
4D06BFDCF56CD24DDBF6A66E47F94BDD

After the public keys are exchanged, the card and terminal calculate a shared secret point by multiplying the received public key with their private key. Calculation of the shared secret point by the terminal is shown on Listing 4.

```
secret_point = card_public_key * private_key
```

Listing 4. Shared secret calculation in Python

The following values were used and obtained in the calculation:
Terminal's private key: 1F5E08DECE8CD745DFF7686357E99236
6851E123ED80D89224620CE6E451F4FB
Card's public key (x-coordinate): 3BE5FBB970F33008575FD35CC4A2D4C1
05BC087E7FCAE1292128E6DCDCBEA2D4
Card's public key (y-coordinate): 1D96B2798E4E5F126A1C6131CF092A75
DF0A16A7E0DC5846E5FD2A95E6F028C7
Calculated shared secret (x-coordinate): 0FDF01BACE05D8EEEA1461A20CEBC584
F4BE253D891ED13952F7A42619DCA922
Calculated shared secret (y-coordinate): 63CCA540BD95E2205050725447EAAF4B
2FADBE643273C13A81E768CED8669B71

To calculate the mapped base point, the standard base point of the NIST P-256 curve is multiplied with the nonce and the shared secret is added to the result. The resulting EC point is used as a base point in the next step. Listing 5 shows mapping of the new base point.

```
mapped_base_point = base_point * int.from_bytes(decrypted_nonce, 'big') + secret_point
```

Listing 5. Calculating the mapped base point in Python

The following base point was obtained in the calculation:
Mapped point (x-coordinate): 6B55818E89BB96FDA32FAA310212E92B
0753A6031B797697E97CA07548DEB086
Mapped point (y-coordinate): 9D71E8A3634C7EDCCC134AC92725677F
0738CA88688FB3753D98D2196DA052F0

Table 6. Command for exchanging public keys in nonce mapping

MAP NONCE				
C-APDU				
CLA	10		Plain, command chaining	
INS	86		General Authenticate	
P1/P2	0000			
Lc	45		Length of data	
Data	**Tag**	**Length**	**Value**	**Comment**
	7C	43	-	Dynamic Authentication Data
	81	41		Mapping Data
			04	Uncompressed point
			8B50756DE011B9C6C7F5 3E31AE36082101687F8D EDEF7C193FEF365452FB 922B	X-Coordinate
			82CB18DC26CFD33EA174 34C2C17B7EEC4D06BFDC F56CD24DDBF6A66E47F9 4BDD	Y-Coordinate
Le	00			Excpected length of response
R-APDU				
Data	**Tag**	**Length**	**Value**	**Comment**
	7C	43	-	Dynamic Authentication Data
	82	41		Mapping Data
			04	Uncompressed point
			3BE5FBB970F33008575F D35CC4A2D4C105BC087E 7FCAE1292128E6DCDCBE A2D4	X-Coordinate
			1D96B2798E4E5F126A1C 6131CF092A75DF0A16A7 E0DC5846E5FD2A95E6F0 28C7	Y-Coordinate
SW	9000			

4.1.5 ECDH key agreement

The purpose of this step is to obtain a shared secret that will be used to derive the symmetric encryption key and MAC key. Using the mapped base point, both card and terminal generate a new EC key pair. The card and terminal exchange their public keys using `GENERAL AUTHENTICATION` command (Table 7), similarly as in the previous step. The main difference is the tag values used for the values inside `Dynamic Authentication Data` objects being `83` for the terminal's public key and `84` for the card's public key. A secret point is calculated by multiplying the card's public key with the terminal's private key.

The following values were used and obtained in the calculation:
Terminal's private key: `27F9A7A2A4C817A595A3219106CEDDB1 72B5D47FA690320088FF855773976EDB`
Terminal's public key (x-coordinate): `0D7920459ABC54DA30A50467F18196A5 AD8827CE1113952D8C2F9594F6AD9D0A`
Terminal's public key (y-coordinate): `4CC13F201E1A8B2A164B4BA08D7F04F8 7C76E02C5A758ADB72638ADB47ACD72C`
Card's public key (x-coordinate): `582455F149C2C78FC18C3270DE86D974 7D44133FF44390B79F449DA2A011B4ED`
Card's public key (y-coordinate): `36558F11C6BC372D5C40D1D3C0EE4851 79BC43A7DF5ED802A12686CB9BDD656D`
Calculated shared secret (x-coordinate): `E4D6FFA165A5C8F69B8A554AE8B45677 6023626D63D7F5851D0C7965AC810113`

Table 7. Command for exchanging public keys in key agreement

PERFORM KEY AGREEMENT				
C-APDU				
CLA	10		Plain, command chaining	
INS	86		General Authenticate	
P1/P2	0000			
Lc	45		Length of data	
Data	**Tag**	**Length**	**Value**	**Comment**
	7C	43	-	Dynamic Authentication Data
	83	41		Terminal's Ephemeral Public Key
			04	Uncompressed point
			0D7920459ABC54DA30A5 0467F18196A5AD8827CE 1113952D8C2F9594F6AD 9D0A	X-Coordinate
			4CC13F201E1A8B2A164B 4BA08D7F04F87C76E02C 5A758ADB72638ADB47AC D72C	Y-Coordinate
Le	00			Excpected length of response
R-APDU				
Data	**Tag**	**Length**	**Value**	**Comment**
	7C	43	-	Dynamic Authentication Data
	84	41		Card's Ephemeral Public Key
			04	Uncompressed point
			582455F149C2C78FC18C 3270DE86D9747D44133F F44390B79F449DA2A011 B4ED	X-Coordinate
			36558F11C6BC372D5C40 D1D3C0EE485179BC43A7 DF5ED802A12686CB9BDD 656D	Y-Coordinate
SW	9000			

4.1.6 Session key derivation

The 32-byte x-coordinate of the established shared secret point is used to generate symmetric encryption key `Kenc` and MAC key `Kmac`, that will be used for secure messaging. These keys are generated by appending to the shared secret a 4-byte value encoding integer 1 for `Kenc` and 2 for `Kmac`, and hashing it using SHA-256 hash function. The obtained 32-byte hash values are used as the session keys.

```
import hashlib

Kenc = hashlib.sha256(shared_secret + b"\x00\x00\x00\x01").digest()
Kmac = hashlib.sha256(shared_secret + b"\x00\x00\x00\x02").digest()
```

Listing 6. Derivation of symmetric session keys

The following values were used and obtained in the calculation:
Shared secret: `E4D6FFA165A5C8F69B8A554AE8B45677`
`6023626D63D7F5851D0C7965AC810113`
Kenc: `1BFDD3D7F3A45BAB2660E85DD112D89F`
`989B67DF1BC003A06722DEA3E55A2B79`
Kmac: `A371B21C3F419BB48279376464DB196C`
`53DA8630399274FE769DD363ACE2F782`

4.1.7 Mutual authentication

To verify that the card and the terminal used the same CAN in the PACE process, the card and terminal have to confirm that the derived symmetric keys are the same. This is done by calculating MAC values from the public keys that were exchanged in the ECDH key agreement step. To calculate these values, `Authentication Token` object must be constructed that is used as the input for MAC calculation. It is a TLV object with the tag `7F49` containing the OID of the PACE protocol used and an uncompressed point of the public key (see Table 8). For the terminal's `Authentication Token` the public key of the card is used and for the card's `Authentication Token` the terminal's public key is used.

Table 8. Data for authentication token MAC calculations

Input data for terminal's authentication token			
Tag	**Length**	**Value**	**Comment**
7F49	4F		Input data
06	0A	04007F00070202040204	PACE OID
86	41	-	Card's public point
		04	Uncompressed point
		582455F149C2C78FC18C3270DE86D9747D44133FF44390B79F449DA2A011B4ED	X-Coordinate
		36558F11C6BC372D5C40D1D3C0EE485179BC43A7DF5ED802A12686CB9BDD656D	Y-Coordinate
Input data for card's auhentication token			
Tag	**Length**	**Value**	**Comment**
7F49	4F		Input data
06	0A	04007F00070202040204	PACE OID
86	41	-	Terminal's public point
		04	Uncompressed point
		0D7920459ABC54DA30A50467F18196A5AD8827CE1113952D8C2F9594F6AD9D0A	X-Coordinate
		4CC13F201E1A8B2A164B4BA08D7F04F87C76E02C5A758ADB72638ADB47ACD72C	Y-Coordinate

MAC value is calculated using CMAC with the derived key `Kmac`. The implementation of CMAC is provided by `PyCryptodome` library [17]. The first 8 bytes from the calculated MAC are used as the MAC tokens, that are exchanged with the card using `GENERAL AUTHENTICATE` command (Table 9). The received card's MAC value is compared to the one that was calculated. If they match, the PACE has been successfully performed and the derived session keys verified. If the card fails to verify terminal's MAC token (due to the incorrect CAN used to establish PACE or some other error) the card will respond with SW `6300` and the counter of incorrect PACE tries will be increased (see Section 4.2).

```
from Cryptodome.Hash import CMAC

cmac = CMAC.new(Kmac, ciphermod=AES)
cmac.update(authentication_token)
mac = cmac.digest()[:8]
```

Listing 7. MAC calculation

The following values were used and obtained in the calculation:
Terminal's Authentication Token: `7F494F`
`060A04007F00070202040204`
`8641`
`04`
`582455F149C2C78FC18C3270DE86D9747D44133FF44390B79F449DA2A011B4ED`
`36558F11C6BC372D5C40D1D3C0EE485179BC43A7DF5ED802A12686CB9BDD656D`
Card's Authentication Token: `7F494F`
`060A04007F00070202040204`
`8641`
`04`
`0D7920459ABC54DA30A50467F18196A5AD8827CE1113952D8C2F9594F6AD9D0A`
`4CC13F201E1A8B2A164B4BA08D7F04F87C76E02C5A758ADB72638ADB47ACD72C`
Terminal's MAC token: `EEBC2B60D4A490D3`
Card's MAC token: `F7E5336BDF05B977`

Table 9. Command for exchanging MAC tokens in mutual authentication

MUTUAL AUTHENTICATION				
C-APDU				
CLA	00		Plain	
INS	86		General Authenticate	
P1/P2	0000			
Lc	0C		Length of data	
Data	**Tag**	**Length**	**Value**	**Comment**
	7C	0A	-	Dynamic Authentication Data
	85	08	EEBC2B60D4A490D3	Terminal's Authentication Token MAC
Le	00			Excpected length of response
R-APDU				
Data	**Tag**	**Length**	**Value**	**Comment**
	7C	0A	-	Dynamic Authentication Data
	86	08	F7E5336BDF05B977	Card's Authentication Token MAC
SW	9000			

4.2 Protection against CAN brute force attacks

It was experimentally discovered that the card counts the number of times an incorrect MAC token has been sent to the card in the mutual authentication step of the PACE protocol (Section 4.1.7). After 10 consecutive incorrect MAC tokens are received by the card, the card introduces a 30 second delay before returning the encrypted nonce (Section 4.1.3). The delay is removed after a successful PACE establishment either on the contact or contactless interface. We conclude that this is a security measure implemented by the card to prevent CAN guessing attacks by the terminal.

To brute force the 6-digit CAN, on average $\frac{10^6}{2}$ attempts would be needed. Considering that the average time for performing PACE using the contactless interface is 1.6 seconds (see Section 5), it would take on average 9.26 days to find the correct CAN if the delay measure was not introduced. With this security measure the time required is increased to 182.27 days.

4.3 Secure messaging

Secure messaging is used to provide secure channel for communication between the card and the terminal. To achieve this, the symmetric encryption key `Kenc` and MAC key `Kmac` established using PACE are used. The encryption key is used to encrypt the data (if present) and MAC key is used to calculate MAC value for the command. Secure messaging APDUs are formed from the encrypted data and the MAC value. An overview of this process for transforming the command APDUs is provided in Figure 5.

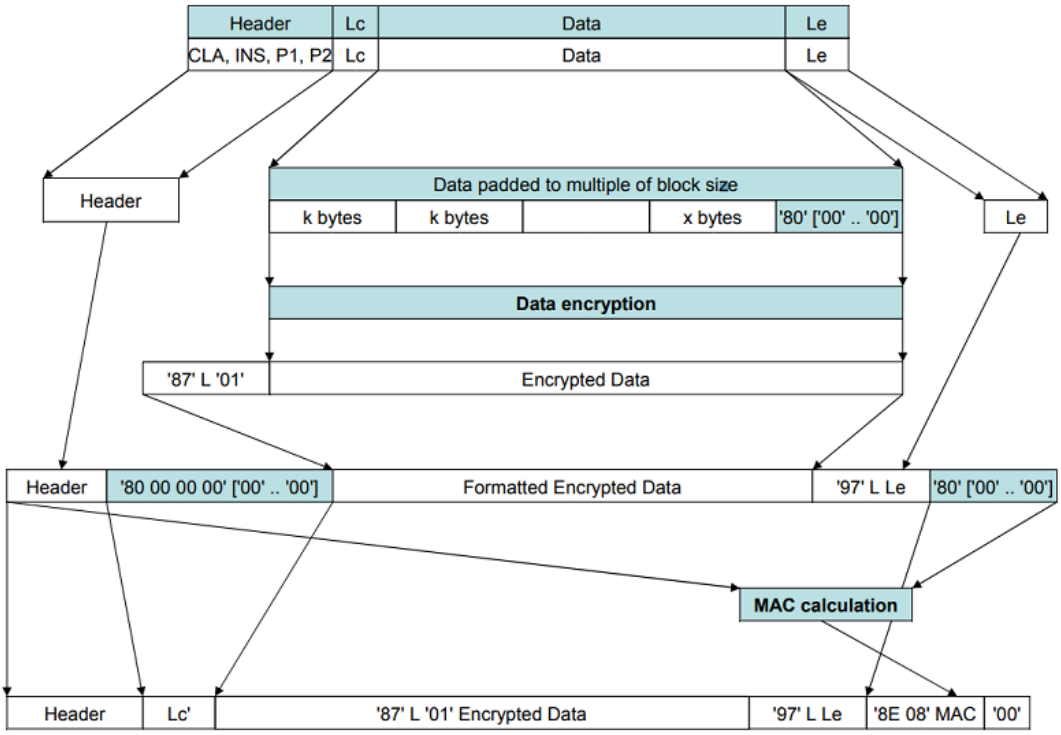

Figure 5. Transformation of C-APDU containing command data [19]

As an additional security mechanism, a send sequence counter (SSC) is used. The SSC is used to keep count of the exchanged APDUs, allowing to detect if an extra command has been sent or lost during communication. After a successful PACE, the 16-byte SSC is initialised to the value 0. The SSC is incremented before sending C-APDU and processing R-APDU.

To better illustrate this process, the reading of the first entry from the personal data file on the Estonian ID card is used as an example below. The personal data file of the Estonian ID card contains information about the cardholder – name, citizenship, personal ID code, date and place of birth, document number and other data fields (page 16 in [2]).

In total, the personal data file contains 15 entries. To read any entry from the personal data file, the personal data file needs to be selected, then the specific entry is selected, and finally, bytes from that file are read [16]. Since a successful PACE has to be performed to use the secure channel, the IAS ECC applet is already selected. The commands used for reading the personal data file are described in Table 10.

For commands sent using the secure channel the first byte (CLA byte) of the APDU header is modified by applying bitwise OR with `0C`. For the first `SELECT FILE` command, as the data field (FID of personal data file `5000`) is present, it needs to be encrypted. For this AES is used in CBC mode.The IV is calculated using AES in ECB mode by encrypting the SSC that has been incremented by 1. The data that is encrypted using AES needs to have size that is in the multiple of the cipher block size (16 bytes), therefore a mandatory padding must be applied to the plaintext. To pad the data, first, the byte `80` is added to the data and then the data is zero-padded to the multiple of block size.

```
from Cryptodome.Cipher import AES

# ssc = 0
# padded_data = 0x5000800000000000000000000000000000
# k_enc = 0x18452162CC454615F6881DB69AA1B3335E8743D7871985A31CC7DB804BC9FDF3

ssc += 1

aes_ecb = AES.new(k_enc, AES.MODE_ECB)
iv = aes_ecb.encrypt(ssc.to_bytes(16, 'big'))

aes_cbc = AES.new(k_enc, AES.MODE_CBC, iv)
encrypted = aes_cbc.encrypt(padded_data)

# encrypted = 0x9068DB9E71676629B3FAA7B12632C730
```

Listing 8. Encrypting the C-APDU data

For secure messaging and MAC calculation, the encrypted data is wrapped in a data object with tag `87`, with the first byte of data being `0x01`. If `Le` for the command has been set, it is encapsulated in a data object with tag `97`.

Table 10. Reading the first entry from the personal data file using secure channel

Kenc	18452162CC454615F6881DB69AA1B333		
	5E8743D7871985A31CC7DB804BC9FDF3		
Kmac	C68BC4E85E0E8F168670C956563E6C9B		
	3DE38AF822894F3547BC3C0D6F047F0B		
SELECT FILE 5000			
Header	00A4010C	**Data**	5000
Secure messaging C-APDU		**SSC**	1
Header	0CA4010C	Lenght	1D
Data	87 11 01 9068DB9E71676629B3FAA7B12632C730		
MAC	8E 08 80D7B85009F5E6C0		
Le	00		
Secure messaging R-APDU		**SSC**	2
Status	99 02 9000		
MAC	8E 08 EBEBBDC3BA1D4EC6		
SW	9000		
SELECT FILE 5001			
Header	00A4010C	**Data**	5001
Secure messaging C-APDU		**SSC**	3
Header	0CA4010C	Lenght	1D
Data	87 11 01 114DD93D509F0D920D7D86AAE936EC9C		
MAC	8E 08 77BAA27E5C2FC36500		
Le	00		
Secure messaging R-APDU		**SSC**	4
Status	99 02 9000		
MAC	8E 08 9256C1F7C9D621BB		
SW	9000		
READ BINARY			
Header	00A4010C	**Le**	00
Secure messaging C-APDU		**SSC**	5
Header	0CB00000	Lenght	0D
Le	97 01 00		
MAC	8E 08 CC31BD80E0F53B2A		
Le	00		
Secure messaging R-APDU		**SSC**	6
Data	87 11 01 0156E02ED64BBDA4F75164DC2AEC32DE		
Status	99 02 9000		
MAC	8E 08 ED886E3016FD9998		
SW	9000		
Decrypted	4B495649564152458000000000000000		
Plain	KIVIVARE		

After the command data is encrypted, the MAC value of the APDU needs to be calculated. The input for MAC calculation is created by concatenating the newly created values together in a specific order. First comes the 16-byte SSC, followed by the modified header that has been padded to the block size. Then, if present, are added the optional data objects with tags `87` and `97`. If any of the data objects with the tag `87` or `97` are present, the input must be padded to the multiple of cipher block size. The transformation of C-APDUs without command data and `Le` byte is shown in Figure 6. The MAC value is then calculated using CMAC. From the resulting CMAC value only the first 8 bytes are used as a MAC.

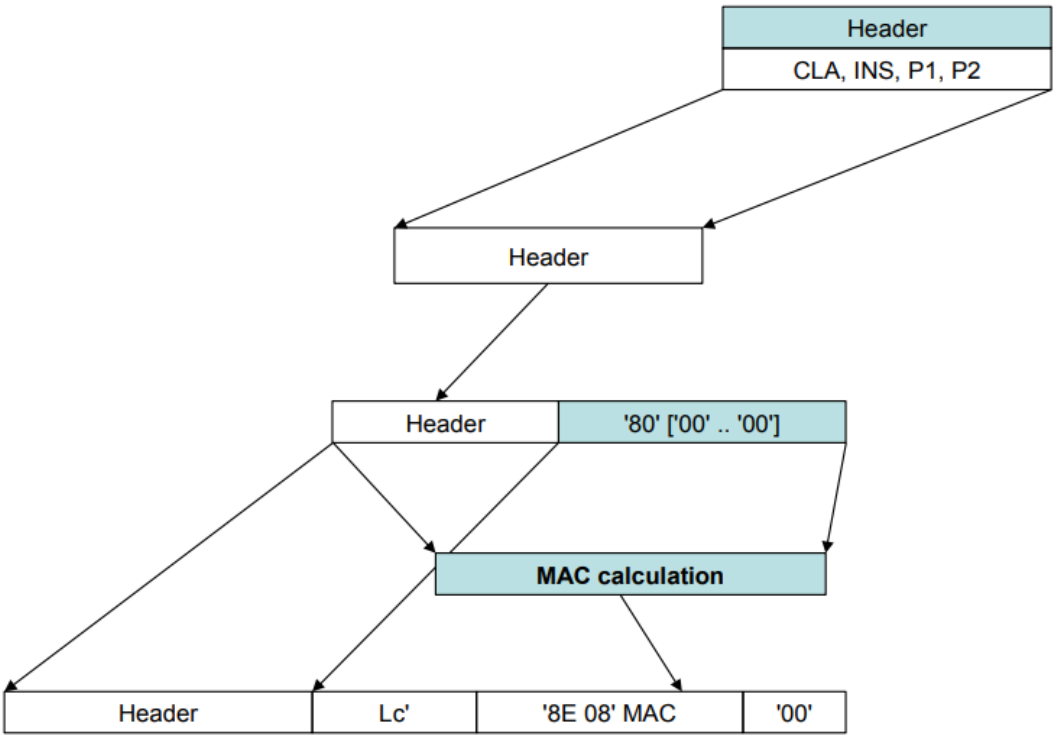

Figure 6. Transformation of C-APDU without command data [19]

```
from Cryptodome.Hash import CMAC

# k_mac = 0xC68BC4E85E0E8F168670C956563E6C9B3DE38AF822894F3547BC3C0D6F047F0B

# mac_data = 0x00000000000000000000000000000001\
# 0CA4010C8000000000000000000000\
# 8711019068DB9E71676629B3FAA7B126\
# 32C730800000000000000000000000

cmac = CMAC.new(k_mac, ciphermod=AES)
cmac.update(mac_data)
mac = cmac.digest()[:8]

# mac = 0x80D7B85009F5E6C0
```

Listing 9. Calculating MAC for the C-APDU

The final C-APDU consists of the original C-APDU header with the CLA and Lc bytes modified, followed by the 87 and 97 data objects (if present), followed by MAC value encapsulated in the tag 8E, and finally the Le byte (if present). The other commands (selecting the first entry and reading the bytes) are transformed in a similar manner.

The R-APDUs are also cryptographically protected by encrypting the response data (if present) and calculating MAC over the encrypted data and the returned SW. The transformation of R-APDU is shown in Figure 7. The data for R-APDU consists of an optional encrypted response data in tag 87, followed by 2-byte status word in tag 99, followed by MAC value in tag 8E. The input for MAC calculation is constructed by concatenating the SSC together with the encrypted data (if present), the 2-byte status word in tag 99, the byte 80 and zero-padding to the multiple of cipher block size.

```
from Cryptodome.Hash import AES

# ssc = 5
# encrypted_response_data = 0x0156E02ED64BBDA4F75164DC2AEC32DE

ssc += 1

aes_ecb = AES.new(k_enc, AES.MODE_ECB)
iv = aes_ecb.encrypt(ssc.to_bytes(16, 'big'))

aes_cbc = AES.new(k_enc, AES.MODE_CBC, iv)
decrypted_response = aes_cbc.decrypt(ciphertext)

# decrypted_response = 0x4B4956495641524580000000000000000
# plaintext = 0x4B49564956415245 (KIVIVARE)
```

Listing 10. Decrypting R-APDU data

After decryption, the padding needs to be removed to get the plaintext response data. In this example, the response data is the first entry of the personal data file corresponding to the surname of the cardholder.

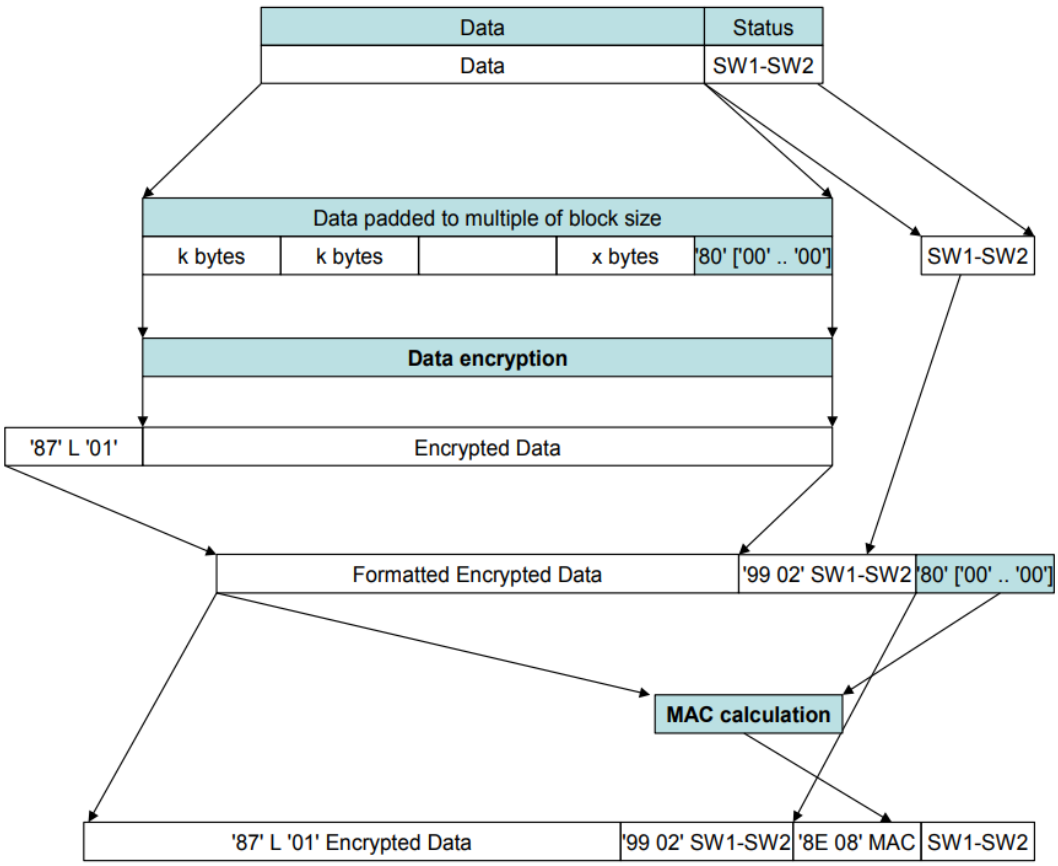

Figure 7. R-APDU transformation [19]

5 Results

To measure the added overhead of the PACE protocol and secure messaging, several experiments were performed over contact and contactless interface. For contact interface, the built-in Broadcom reader of a Dell laptop was used. For contactless mode, ACS ACR1252U NFC reader [20] was used, which is a universal contactless reader that supports ISO 14443 Type A standard that is used by the Estonian ID card.

Table 11. Timings for PACE protocol (in seconds)

Command	**Contact**	**Contactless**
Select application	0.145	0.178
MSE:SET AT	0.133	0.127
Get nonce	0.159	0.157
Nonce mapping	0.251	0.244
Key agreement	0.571	0.566
Mutual authentication	0.210	0.208
Total	1.594	1.604
Python overhead	0.125	0.124

The time taken to execute PACE was similar in both contact and contactless mode, taking around 1.6 seconds. The time taken for each command sent to the card in contact and contactless modes is shown in Table 11, along with the total execution time which includes the overhead caused by the Python code performing the necessary calculations. Although the execution time was similar, it should be noted that the placement of the card on the NFC reader had an impact on the execution time. When positioned not completely in the NFC reader's slot for the card or further away, the execution time increased. The timings provided in Table 11 were measured when the card was placed optimally on the NFC reader.

The reading of personal data file was used to analyse the impact of using the secure channel. The time for reading the personal data file was measured using contact mode without secure messaging (SM) and compared to contact and contactless modes with secure messaging. The time was measured from selecting the file until all 15 entries had been read. The timing measured for sending commands and the total time to read all entries is provided in Table 12. The time measurements for each command do not include the time for calculating the values required for secure messaging.

Table 12. Timings for personal data file reading (in seconds)

Command	**Contact (w/o SM)**	**Contact (w SM)**	**Contactless (w SM)**
Select personal data	0.022	0.032	0.031
Select Entry (avg)	0.021	0.032	0.031
Read entry (avg)	0.024	0.037	0.036
Total (for 15 entries)	0.696	1.079	1.049

As expected, the addition of secure messaging caused the commands to take a longer time to process. Calculating the MAC value and encrypting the data for secure messaging requires additional calculations to be made by the card. Similarly to the execution times for PACE, the difference between using a contact reader and contactless reader was negligible. For practical purposes, the impact of PACE to establish the keys is the most significant, as it takes a relatively long time to perform compared to the actual reading of the entries. This is important when considering its use for scenarios where the card is not placed statically in a reader and the execution time is important.

For example, the reading of the personal data file and its entries is currently used in loyalty card and access control systems [21]. The main drawback of the contact interface is the need to insert the card into the reader and the wear caused by it to the chip. The use of NFC interface for these applications would remove this drawback, but the current security mechanism makes it impractical. The reading of the necessary data in the contactless mode requires the establishment of secure channel and therefore a successful PACE. Along with the increased execution time, the entering of the CAN is also required. This would make the system more complex, as additional user inputs are needed or optical character recognition implemented, negating the effect of not having to insert the card into the reader.

6 Software implementation

A simple Python 3 implementation of PACE and secure messaging that can be used as a reference to demonstrate the functionality described in this thesis is available in [22]. The implementation supports only the PACE protocol features implemented in the Estonian ID card. The secure messaging is simplified and is not fully in compliance with the standard.

To implement the required functionality several external libraries are used. For communication with smart card the library `pyscard` [23] is used. For cryptographic functions the library `PyCryptodome` [17] is used, as it has the necessary implementations of AES and CMAC. The Python library `ECDSA` [18] is used for elliptic curve operations as it supports the NIST P-256 curve used by PACE implementation of the Estonian ID card.

6.1 Existing libraries

Below are the results of testing the existing Java and Python libraries that implement PACE support. Some modifications to the code are expected as the IAS ECC applet that is specific to the Estonian ID card has to be selected.

JMRTD [24] is an open source Java implementation of ICAO Machine Readable Travel Document Standards available under GNU Lesser General Public License. It features both card side JavaCard application for implementing ePassport application and host side API for reading cards. It also has an implementation of PACE, making it suitable for use with the Estonian ID card. This is the only library that was able to establish secure channel with the Estonian ID card and was able to exchange commands using secure messaging.

Animamea [25] is an open source Java implementation of PACE according to BSI TR-03110 [19], a German specification for establishing the secure channel that is based on ICAO Doc 9303 [12]. Unfortunately, it fails to perform successful PACE with the Estonian ID card. The analysis of the source code shows that during the decryption of nonce the 32 byte nonce is incorrectly truncated to 16 bytes.

Pypace [26] is the only Python-based library that was found to implement the PACE protocol. Unfortunately, the implementation for secure messaging is missing and it supports only one specific elliptic curve that is not supported by the Estonian ID card. As the author of Animamea and Pypace is the same, a similar nonce truncation as in Animamea is present. This library was used as an example for the implementation presented in this thesis.

7 Conclusion

For using the contactless interface of the new generation Estonian ID card, a secure channel needs to be established. This thesis gave a detailed walk-through on how to establish a secure messaging channel providing a reference implementation in Python 3. The performance impact of establishing the channel was measured along with the impact of using it for reading data.

As this thesis was aimed to give a practical overview of using NFC and the secure channel, the potential security impacts were not analysed. Although only the reading of the personal data file via NFC was described, all of the electronic functionality of the ID card is available using the contactless interface. This functionality can be used to integrate the Estonian ID card support in mobile applications. For example, the RIA DigiDoc application [27] that currently allows digital signing of documents with the Estonian ID card using an external reader could be extended with the NFC support.

References

[1] ID Help Centre. Estonian ID - Changes in the new ID-card. https://www.id.ee/index.php?id=38693/. (2020.05.02).

[2] Estonian Information System Authority. Estonia ID1 Chip/App 2018 Technical Description v0.8. https://installer.id.ee/media/id2019/TD-ID1-Chip-App.pdf. (2020.05.02).

[3] Martin Paljak. EstEID Wiki - NFC. https://github.com/martinpaljak/esteid.java/wiki/NFC. (2020.05.02).

[4] CardWerk. ISO 7816 Smart Card Standard overview. https://cardwerk.com/iso-7816-smart-card-standard. (2020.05.02).

[5] Vedat Coskun, Kerem Ok, and Busra Ozdenizci. *Near field communication (NFC): From theory to practice*. John Wiley & Sons, 2011.

[6] Estonian Police and Border Guard Board. Applying for an ID card for an adult. https://www.politsei.ee/en/instructions/applying-for-an-id-card-for-an-adult/. (2020.05.02).

[7] Council of the European Union. PRADO - Public Register of Authentic travel and identity Documents Online: EST-BO-04001. https://www.consilium.europa.eu/prado/en/EST-BO-04001/index.html. (2020.07.12).

[8] ERR News. Estonia's first new ID cards to be issued this week, December 11, 2018. https://news.err.ee/883962/estonia-s-first-new-id-cards-to-be-issued-this-week (2020.05.02).

[9] Aivar Pau. Contactless Estonian ID-card has been built (in Estonian). Postimees. https://tehnika.postimees.ee/3607697/video-valminud-on-kontaktivaba-eesti-id-kaart. (2020.05.02).

[10] Jonas Kiiver. *NFC Security Solution for Web Applications*. MSc thesis, University of Tartu, 2015. https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=46871&year=2015.

[11] Gregor Johannson. *Technical Prerequisites for Enabling Third-Party Applications on the New Estonian ID-card*. MSc thesis, Tallinn University of Technology, 2019. https://digikogu.taltech.ee/et/Item/64c83d8f-8f2d-4311-b548-b07c9b58a6cb.

[12] International Civil Aviation Organization. Doc 9303 Part 11: Security Mechanisms for MRTDs, 2015. https://www.icao.int/publications/Documents/9303_p11_cons_en.pdf. (2020.05.02).

[13] Certicom Research. Standards for efficient cryptography, sec 1: Elliptic curve cryptography. Cryptology ePrint Archive, Report 2004/332, 2004. http://www.secg.org/sec1-v2.pdf.

[14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (Chapman Hall/Crc Cryptography and Network Security Series)*. Chapman Hall/CRC, 2007.

[15] Jens Bender and Dennis Kügler. Introducing the PACE solution, 2015. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/ElekAusweise/Keesing_10_09_Introducing_the_PACE_solution_pdf.pdf?__blob=publicationFile.

[16] ID Help Centre. ID1 Developer Guide. https://www.id.ee/public/ID1DeveloperGuide.pdf. (2020.05.02).

[17] Pycryptodome. https://pycryptodome.readthedocs.io/en/latest/src/introduction.html. (2020.05.02).

[18] Pure-Python ECDSA. https://github.com/warner/python-ecdsa. (2020.05.02).

[19] Federal Office for Information Security. BSI TR-03110 Technical Guideline Advanced Security Mechanisms for Machine Readable Travel Documents and eIDAS Token. https://www.bsi.bund.de/EN/Publications/TechnicalGuidelines/TR03110/BSITR03110.html. (2020.05.02).

[20] Advanced Card Systems Ltd. ACR1252U USB NFC Reader. https://www.acs.com.hk/en/products/342/acr1252u-usb-nfc-reader-iii-nfc-forum-certified-reader/. (2020.05.02).

[21] Danielle Morgan and Arnis Parsovs. Using the Estonian Electronic Identity Card for Authentication to a Machine (Extended Version). Cryptology ePrint Archive, Report 2017/880, 2017. http://eprint.iacr.org/2017/880.

[22] Sander-Karl Kivivare. Python Implementation of PACE for the Estonian ID card. https://github.com/Kivivares/estid-nfc. (2020.08.10).

[23] Pyscard - Python for smart cards. https://pyscard.sourceforge.io/. (2020.05.02).

[24] JMRTD. https://jmrtd.org/about.shtml. (2020.05.02).

[25] Animamea. https://github.com/tsenger/animamea3. (2020.05.02).

[26] Pypace. https://github.com/tsenger/pypace. (2020.05.02).

[27] Google Play. RIA DigiDoc. https://play.google.com/store/apps/details?id=ee.ria.DigiDoc. (2020.05.02).

Appendix

I. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Sander-Karl Kivivare**,
*(*author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Secure Channel Establishment for the NFC Interface of the New Generation Estonian ID Cards,
   *(*title of thesis)

   supervised by Arnis Paršovs.
   *(*supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Sander-Karl Kivivare
10/08/2020