SIM Card Key Management System Phase 4

In this manual, Key Management System (KMS) on the SIM card is described.

Current version of the KMS applet includes master key generation on SIM card, key wrapping, key generations, transaction Id, key request, key received notification, key removal based on key received notification, symmetric encryption and decryption, asymmetric encryption and decryption, pin enrolling and pin verification.

1. Introduction to KMS SIM Card Applet

In order to communicate with the KMS applet, initially the applet must be selected using the following AID of the applet:

Applet AID: 04 5A D1 9E 6C 58 B9 41 CC

Applet Select Command: 00 A4 04 00 09 04 5A D1 9E 6C 58 B9 41 CC

Next step after selecting the KMS applet is to communicate with it using specific APDU messages that match the purpose of communication. The structure of an APDU message is given in Table 1 and Table 2.

Table 1. Command APDU Format (That needs to be sent to the KMS)

Parameter Name	Length	Description
CLA	1 byte	Instruction class
INS	1 byte	Instruction code, specific command.
P1 - P2	2 bytes	Additional slots for undefined usage
L_c	0,1 or 3 bytes	The length of the data
Command Data	variable	Payload of the data
L_e	0,1,2 or 3 bytes	The length of the expected response data. Usage is optional

Table 2. Response APDU Format (Response of the KMS)

Name	Length	Description
Response data	variable	Response data
SW1-SW2	2 bytes	Command processing status Operation is successful if it is 90 00

The parameters of the command APDU message that is sent to the KMS should be as follows:

- CLA : `0x80`
- INS : different for every method
- P1 : different for every method
- P2 : different for every method
- L_c : different for every method
- Command Data: different for every method
- L_e : different for every method

2. Long APDU Messages

In standard SIM-handset communication, the maximum APDU message size that they can exchange is less than 256 bytes. However, in Key Management system, they need to exchange more than 256 bytes in some requests. Thus a new methodology is needed that provides exchanging more than 256 bytes between SIM card and SIM communication module. When a request is performed which's response could be more than 256 bytes, this methodology is applied.

When that kind of a request is made from mobile handset, the first two bytes of an APDU message defines the current message sequence and total message size. For example: if the first two bytes of an APDU message is 0x01 0x02, it means that this is message 1 of 2. So the mobile handset should request the 2nd message. 0x02 0x02 means; this is message 2 of 2. So the mobile handset should not request any more messages. 0x01 0x01 means; this is message 1 of 1. So the mobile handset should not request any more messages. A total of 250 bytes after these sequence bytes are accepted from SIM applet.

Moreover, sequence number is included in some request messages (the messages sent from mobile handset). It defines the sequence number of the message that is requested. When mobile handset requests the 1st of 2 messages, it should use sequence number as 0x01. When mobile handset requests the 2nd of 2 messages, it should use sequence number as 0x02 and so on. When a request is performed with a sequence number of 1, SIM card performs the calculation and returns the first 255 bytes (The first two bytes of the response message are again the current message sequence and total message size. So, 253 bytes of data are actually returned). When a request is performed with a sequence number of 2, SIM card again performs the calculation and returns the second 255 bytes and so on for next sequence numbers.

Example: As an example; mobile handset performs an encryption request from the SIM card with a sequence number of 1. SIM performs the encryption and returns 0x01 0x03 and plus the first 253 bytes of the result. Mobile handset checks the first two bytes and understands that there will be three messages and this is the first message. Then, mobile handset performs the SAME request from the SIM card with a sequence number 2. SIM performs the encryption and returns 0x02 0x03 and plus the second 253 bytes of the result. Mobile handset understands that this is the second message. Then, mobile handset performs the SAME request from the SIM card with a sequence number 3. SIM performs the encryption and returns 0x03 0x03 and plus the remaining 50 bytes of the result. Mobile handset understands that this is the last message (3 of 3). Then it merges previous two response messages with the last one and obtains the result which is 556 bytes.

3. KMS SIM Card Applet Functionalities

In this section, we define the methods and APDU examples for calling specified methods.

Before calling the method, the application should select the AID of the applet as described in section 1. Then, it can call the methods described in here.

a. Master key generation

A 168 bit master key is generated after the applet is installed to the SIM card. This master key is used to wrap any data associated with Key Management. There is no need to perform any operation from mobile handset.

b. Create Symmetric Key

Following method is used for symmetric key generation:

```
byte[] createSymmetricKey ( byte keyLength, byte[] keyPin )
```

keyLength: Length of the key shall be specified

Key length	**`keyLength`** parameter value
168	0x01
256	0x02

keyPIN: pin to protect the key with (4 bytes)

returns: Creating a key may take some time. Thus, `createSymmetricKey` method will return a `transactionId` immediately which is 8 bytes long. Mobile handset then should use `transactionId` to request the generated wrapped key.

Usage from Mobile Handset:

```
byte[] createSymmetricKey( byte keyLength, byte[] keyPin)
```

For a 168-bit symmetric key generation, following APDU needs to be sent to SIM:

```
0x80 0x20 0x01 0x00 0x04 0x01 0x02 0x03 0x04
```

CLA	INS	P1	P2	Lc	Data
0x80	0x20	0x01	0x00	0x04	< keyPin >

CLA represents the CLA of the applet and is same for all command APDUs sent to the KMS applet
INS represents `createSymmetricKey` method
P1 represents `keyLength`
Lc represents the length of the Data part
Data represents the `keyPin`
In order to generate a key without a PIN, a default PIN can be sent to the SIM card by the framework

Returns:
The result of the APDU command is `transactionId` and will be returned as a byte array.

c. Create Asymmetric Key

Following method is used for asymmetric key generation:

```
byte[] createAsymmetricKey (byte keyLength, byte[] keyPin)
```

keyLength: Length of the key shall be specified

Key length	`keyLength` parameter value
1024	0x01

keyPIN: pin to protect the key with (4 bytes)

returns: Creating a key may take some time. Thus, `createAsymmetricKey` method will return a `transactionId` immediately which is 8 bytes. Mobile handset then should use `transactionId` to request the generated wrapped key.

exception: In some SIM cards, an exception can be thrown if the SIM card is not capable of generating the requested key. In this situation an RSA key generation exception is thrown which is A0 B0 C0

Usage from Mobile Handset:

```
byte[] createAsymmetricKey(byte keyLength, byte[] keyPin)
```

For a 1024 bit RSA key generation, following APDU needs to be sent to SIM:

```
0x80 0x26 0x01 0x00 0x04 0x01 0x02 0x03 0x04
```

CLA	INS	P1	P2	Lc	Data
0x80	0x26	0x01	0x00	0x04	<keyPin>

CLA represents the CLA of the applet and is same for all command APDUs sent to the KMS applet
INS represents `createAsymmetricKey` method
P1 represents `keyLength`
Lc represents the length of the Data part
Data represents the `keyPin`
In order to generate a key without a PIN, a default PIN can be sent to the SIM card by the framework

Returns:
The result of the APDU command is `transactionId` and will be returned as a byte array.

d. Request Key

When the mobile handset sends a key generation command, it receives a `transactionId` instead of a wrapped key; since the key generation process may take some time. For this reason, after some time passes, the terminal should request the wrapped key using `transactionId`.

`byte[] requestKey (byte[] transactionId)`

> **transactionId:** Terminal should send the `transactionId` that it received when the `createKey` method is called.

Note: Mobile handset should use sequence number in this method as described in Section 2, because SIM card may need to send the response in more than one APDU message. The details on how to use sequence number in this method is given in method specification in Section 2.

returns:

- If the key is symmetric, `requestKey` method returns $(key(S)+keyPIN)_w$
- If the key is PKI, `requestKey` method returns $(key(AS)_{priv} +keyPIN)_w + key(AS)_{public}$

Usage from Mobile Handset:

`byte[] requestKey (byte transactionId)`

This method is used to retrieve the keys created in `createKey` method based on the `transactionId`. The following APDU needs to be sent to SIM card to retrieve a wrapped key (Note that the length of the `transactionId` is 8 bytes) :

`0x80 0x21 0x01 0x01 0x08` <transactionId>

CLA	INS	P1	P2	Lc	Data
0x80	0x21	0x01	0x01	0x08	< transactionId >

CLA represents the CLA of the applet and is same for all command APDUs sent to the KMS applet
INS represents `requestKey` method
P1 represents `keyType` : 01 for symmetric, 02 for asymmetric
P2 represents sequence number
Lc represents the length of the Data part.
Data represents the `transactionId`.

Returns:

- If the key is symmetric, `requestKey` method returns $(key(S)+keyPIN)_w$
- If the key is PKI, `requestKey` method returns $(key(AS)_{priv} + keyPIN)_w + key(AS)_{public}$
- If the key is not yet generated, applet returns A0 B0 C0

e. Key Received Notification

When the mobile handset receives the wrapped key, it should send a key received notification in order for SIM card to delete the wrapped key. Otherwise, it may be stored in SIM card unnecessarily.

`keyReceived (byte transactionId)`

> **transactionId:** Terminal should send the `transactionId` that it received when the `createKey` method is called.

returns: Nothing. Just 0x90 0x00

Usage from Mobile Handset:

`void keyReceived (byte transactionId)`

This method is used to notify the SIM card on received key. When the mobile handset receives the wrapped key, it must invoke this method. Following APDU needs to be sent to SIM card:

`0x80 0x22 0x01 0x00 0x08 <transactionId>`

CLA	INS	P1	P2	Lc	Data
0x80	0x22	0x01	0x00	0x08	<transactionId>

CLA represents the CLA of the applet and is same for all command APDUs sent to the KMS applet
INS represents `keyReceived` method
P1 represents `keyType` : 01 for symmetric, 02 for asymmetric
Lc represents the length of the Data part
Data represents the `transactionId`

f. Wrapping

This method is used to wrap the keys generated and any data that is passed to the method. The mobile handset does not call it directly. SIM card uses it internally.

```
byte[] wrap(data)
```

g. Symmetric Encryption

This method is used to encrypt data using symmetric key.

`byte[] symmetricEncryptionRequest(byte[] wrappedKey, byte[] keyPin, byte[] data)`

wrappedKey: wrapped key consists from $(key(S) + keyPIN)_w$

keyPIN: pin to protect the key with (4 bytes)

data: data that will be encrypted

returns: Encrypted data in byte array

Usage from Mobile Handset:

`byte[] symmetricEncryptionRequest(byte[] wrappedKey, byte[] keyPin, byte[] data)`

This method is used to encrypt the data with symmetric key. Following APDU needs to be sent to SIM card:

`0x80 0x23 0x20 0x00 <lengthOfData> < wrappedKey + keyPin + data >`

CLA	INS	P1	P2	Lc	Data
0x80	0x23	0x20	0x00	0x50	< wrappedKey + keyPin + data >

CLA represents the CLA of the applet and is same for all command APDUs sent to the KMS applet
INS represents `symmetricEncryptionRequest` method
P1 represents the length of the `wrappedKey`
Lc represents the length of the Data part
Data represents `wrappedKey+keyPin+data`

Returns:
Encrypted data in byte array

h. Symmetric Decryption

This method is used to decrypt data using symmetric key.

```
byte[] symmetricDecryptionRequest(byte[] wrappedKey, byte[]
keyPin,  byte[] cipherText))
```

wrappedKey: wrapped key consists from $(key(S) + keyPIN)_w$

keyPIN: pin to protect the key with (4 bytes)

cipherText: data that needs to be decrypted

returns: Plaintext data in byte array

Usage from Mobile Handset:

```
byte[] symmetricDecryptionRequest(byte[] wrappedKey, byte[]
keyPin, byte[] cipherText)
```

This method is used to decrypt the data with symmetric key. Following APDU needs to be sent to SIM card:

```
0x80 0x24 0x20 0x00 <lengthOfData> < wrappedKey + keyPin +
cipherText >
```

CLA	INS	P1	P2	Lc	Data
0x80	0x24	0x20	0x00	0x50	< wrappedKey + keyPin + cipherText >

CLA represents the CLA of the applet and is same for all command APDUs sent to the KMS applet
INS represents `symmetricDecryptionRequest` method
P1 represents the length of the `wrappedKey`
Lc represents the length of the Data part
Data represents `wrappedKey + keyPin + cipherText`

Returns:
Decrypted data in byte array

i. Asymmetric Encryption

This method is used to encrypt data using asymmetric key.

```
byte[] asymmetricEncryptionRequest( byte currentMsg, byte
totalMsgs, byte[] publicKey, byte[] data)
```

currentMsg: current APDU message number

totalMsgs: total APDU messages that will be sent for the request

publicKey: the public key to protect the internally generated symmetric key with

data: the data to be encrypted that is multiple of 8 bytes (e.g. 8 bytes, 16 bytes, 24bytes,...).. If not mobile application should pad it accordingly before sending it to SIM card

This method is used to encrypt the data with user's public key. Indeed, hybrid encryption is performed. Thus, a new 168 bit symmetric key is generated in the SIM card. The data is encrypted with 3DES key. 3DES key is encrypted with user's public key.

Note: Mobile handset should use sequence number in this method as described in Section 2, because SIM card may need to send the response in more than one APDU message. The details on how to use sequence number in this method is given in method specification in Section 2.

Note: Mobile application generally needs to send more than 256 bytes for this method. Thus it needs to send more than one messages. In every message, it needs to send the current message number and total messages. Examples are given below.

returns: 3DES encrypted data and the RSA encrypted (3DES) symmetric key.

Usage from Mobile Handset:

```
byte[] asymmetricEncryptionRequest(byte currentMsg, byte
totalMsgs, byte[] publicKey, byte[] data)
```

Mobile handset should sent the public key and plaintext data to SIM card:

```
0x80 0x27 0x21 0x02 <lengthOfData> < currentMsg + totalMsgs
+ publicKey + data >
```

CLA	INS	P1	P2	Lc	Data
0x80	0x27	0x21	0x02	0x50	< currentMsg +

						totalMsgs + publicKey + data >

CLA represents the CLA of the applet and is same for all command APDUs sent to the KMS applet
INS represents `asymmetricEncryptionRequest` method
P1 represents the (length of `publicKey`**`/4`**)
P2 represents the (length of `data`**`/4`**)
Lc represents the length of the Data part
Data represents `currentMsg + totalMsgs + publicKey + data`

Returns:
3DES encrypted data (length is (length of `data` + 24) bytes) and RSA encrypted (3DES) symmetric key (length is the RSA key length/8. For 1024 bit RSA, it is 128 bytes). The response will be probably sent in multiple response messages.

Example Usage:

Mobile application sends: (Message 1 of 3)

`0x80 0x27 0x20 0x08 0xFF` **`0x01 0x03 ......`**

SIM card sends: `0x90 0x00`

Mobile application sends: (Message 2 of 3)

`0x80 0x27 0x20 0x08 0xFF` **`0x02 0x03 ......`**

SIM card sends: `0x90 0x00`

Mobile application sends: (Message 3 of 3)

`0x80 0x27 0x20 0x08 0x20` **`0x03 0x03 ......`**

SIM card sends: (Message 1 of 2)

`0x01 0x02 .....`

Mobile application needs to request remaining 1 more message from SIM card with the same last message only by incrementing the `currentMsg`:

`0x80 0x27 0x20 0x08 0x20` **`0x04 0x03 .......`**

SIM card sends: (Message 2 of 2)

`0x02 0x02 .....`

j. Asymmetric Decryption

This method is used to decrypt the asymmetric cipher text that is previously encrypted with `asymmetricEncryptionRequest` method.

```
byte[] asymmetricDecryptionRequest(byte currentMsg, byte
totalMsgs, byte[] wrappedKey, byte[] keyPin, byte[] cipherText)
```

currentMsg: current APDU message number

totalMsgs: total APDU messages that will be sent for the request

wrappedKey: the wrapped RSA private key $(key(AS)_{priv} + keyPIN)_w$

keyPIN: pin to protect the key with (4 bytes)

cipherText (result of `asymmetricEncryptionRequest` method)**:** the asymmetric cipher text to be decrypted that includes 3DES encrypted data and the RSA encrypted (3DES) symmetric key

Note: Mobile handset should use sequence number in this method as described in Section 2, because SIM card may need to send the response in more than one APDU message. The details on how to use sequence number in this method is given in method specification in Section 2.

Note: Mobile application generally needs to send more than 256 bytes for this method. Thus it needs to send more than one messages. In every message, it needs to send the current message number and total messages. Examples are given below.

returns: the decrypted data

Usage from Mobile Handset:

```
byte[] asymmetricDecryptionRequest(byte currentMsg, byte
totalMsgs, byte[] wrappedKey, byte[] keyPin, byte[] cipherText)
```

Mobile handset should sent the public key and plaintext data to SIM card:

```
0x80 0x28 0x42 0x2A <lengthOfData> < wrappedKey + keyPin +
cipherText >
```

CLA	INS	P1	P2	Lc	Data
0x80	0x28	0x42	0x2A	0x50	< wrappedKey +

					keyPin + cipherText >

CLA represents the CLA of the applet and is same for all command APDUs sent to the KMS applet
INS represents `asymmetricDecryptionRequest` method
P1 represents the (length of `wrappedKey`**`/4`**)
P2 represents the (length of `cipherText`**`/4`**)
Lc represents the length of the Data part
Data represents `wrappedKey + keyPin + cipherText`

Returns:
plaintext data

Example Usage:

Mobile application sends: (Message 1 of 3)

`0x80 0x28 0x20 0x08 0xFF` **`0x01 0x03 ......`**

SIM card sends: `0x90 0x00`

Mobile application sends: (Message 2 of 3)

`0x80 0x28 0x20 0x08 0xFF` **`0x02 0x03 ......`**

SIM card sends: `0x90 0x00`

Mobile application sends: (Message 3 of 3)

`0x80 0x28 0x20 0x08 0x20` **`0x03 0x03 ......`**

SIM card sends: (Message 1 of 2)

`0x01 0x02 .....`

Mobile application needs to request remaining 1 more message from SIM card with the same last message only by incrementing the `currentMsg`:

`0x80 0x28 0x20 0x08 0x20` **`0x04 0x03 .......`**

SIM card sends: (Message 2 of 2)

`0x02 0x02 .....`

k. Key Exporting (New method in Phase 4)

This method is used to export the plain key from a wrapped key.

```
byte[] keyExport( byte currentMsg, byte totalMsgs, byte[]
wrappedKey, byte[] keyPin )
```

currentMsg: current APDU message number

totalMsgs: total APDU messages that will be sent for the request

wrappedKey: wrapped key consists from either

- $(key(S) + keyPIN)_w$
- $(key(AS)_{priv} + keyPIN)_w$

keyPIN: pin that protected the key with (4 bytes)

returns: the plain key itself

Note: Mobile handset should use sequence number in this method as described in Section 2, because SIM card may need to send the response in more than one APDU message. The details on how to use sequence number in this method is given in method specification in Section 2.

Note: Mobile application generally needs to send more than 256 bytes for this method. Thus it needs to send more than one messages. In every message, it needs to send the current message number and total messages. Examples are given below.

Usage from Mobile Handset:

```
byte[] keyExport (byte currentMsg, byte totalMsgs, byte[]
wrappedKey, byte[] keyPin)
```

Following APDU needs to be sent to SIM card:

```
0x80 0x31 0x01 0x00 <lengthOfData> < currentMsg + totalMsgs
+ wrappedKey + keyPin >
```

CLA	INS	P1	P2	Lc	Data
0x80	0x31	0x01	0x20	0x50	< currentMsg + totalMsgs + wrappedKey + keyPin >

CLA represents the CLA of the applet and is same for all command APDUs sent to the KMS applet

INS represents `keyExport` method
P1 represents `keyType`: 01 for symmetric, 02 for asymmetric
P2 represents the (length of `wrappedKey`**/4**)
Lc represents the length of the Data part
Data represents `currentMsg + totalMsgs + wrappedKey`

Returns:
Either

- the symmetric key or
- the RSA private key's modulus and exponent. (Example: If the return length is 256 byte, than the first 128 byte is modulus and the last 128 byte is exponent)

if the pins do not match, returns A0 B0 C0

l. Key Importing (New method in Phase 4)

This method is used to import an already existing key and receive a wrapped key.

`byte[] keyImport (byte[] key, byte[] keyPIN )`

keyPIN: pin to protect the key with (4 bytes)

key: a symmetric key or an asymmetric RSA private key

returns: the wrapped key. Based on the key type:

- $(key(S) + keyPIN)_w$ or
- $(key(AS)_{priv} + keyPIN)_w$

Note: Mobile handset should use sequence number in this method as described in Section 2, because SIM card may need to send the response in more than one APDU message. The details on how to use sequence number in this method is given in method specification in Section 2.

Note: Mobile application generally needs to send more than 256 bytes for this method. Thus it needs to send more than one messages. In every message, it needs to send the current message number and total messages. Examples are given below.

Usage from Mobile Handset:

`byte[] keyImport (byte[] key, byte[] keyPIN)`

This method is used to import an already existing key and receive a wrapped key. Following APDU needs to be sent to SIM card:

`0x80 0x32 0x01 0x01 <lengthOfData> < currentMsg + totalMsgs + key + keyPIN >`

CLA	INS	P1	P2	Lc	Data
0x80	0x32	0x01	0x20	0x50	< currentMsg + totalMsgs + key + keyPIN >

CLA represents the CLA of the applet and is same for all command APDUs sent to the KMS applet
INS represents `keyImport` method
P1 represents `keyType`: 01 for symmetric, 02 for asymmetric
P2 represents the length of `key` **/ 4**
Lc represents the length of the Data part
Data represents `currentMsg + totalMsgs + key + keyPIN`

Returns:

the wrapped key. Based on the key type:

- $(key(S) + keyPIN)_w$ or
- $(key(AS)_{priv} + keyPIN)_w$

m. Signing

This method is used to encrypt a hash using the user's private key. So the mobile application sends a hash and SIM card signs it using user's private key.

```
byte[] sign (byte currentMsg, byte totalMsgs, byte[] wrappedKey,
byte[] keyPin, byte[] data )
```

currentMsg: current APDU message number

totalMsgs: total APDU messages that will be sent for the request

wrappedKey: the wrapped RSA private key $(key(AS)_{priv} + keyPIN)_{w}$

keyPIN: pin to protect the key with (4 bytes)

data: the data to be signed. data's length should multiple of 8 bytes (e.g. 8 bytes, 16 bytes, 24bytes,...).. If not mobile application should pad it accordingly before sending it to SIM card

Note: Mobile handset should use sequence number in this method as described in Section 2, because SIM card may need to send the response in more than one APDU message. The details on how to use sequence number in this method is given in method specification in Section 2.

Note: Mobile application generally needs to send more than 256 bytes for this method. Thus it needs to send more than one messages. In every message, it needs to send the current message number and total messages. Examples are given below.

returns: RSA private key encrypted signature of the data

Usage from Mobile Handset:

```
byte[] sign(byte currentMsg, byte totalMsgs, byte[] wrappedKey,
byte[] keyPin, byte[] data)
```

This method is used to sign a data using user's private key. Following APDU needs to be sent to SIM card:

```
0x80 0x29 0x20 0x02 <lengthOfData> < currentMsg + totalMsgs
+ wrappedKey + keyPin + data >
```

CLA	INS	P1	P2	Lc	Data
0x80	0x29	0x20	0x02	0x50	< currentMsg +

					totalMsgs + wrappedKey + keyPin + data >

CLA represents the CLA of the applet and is same for all command APDUs sent to the KMS applet
INS represents `sign` method
P1 represents the (length of `wrappedKey` **/4**)
P2 represents the (length of `data`**/4**)
Lc represents the length of the Data part
Data represents `currentMsg+totalMsgs+wrappedKey+keyPin+data`

Returns:
Encrypted signature

n. PIN Enrolling

This method is used to encrypt a pin with a symmetric key.

Mobile application sends a wrapped symmetric key. SIM card gets a PIN from the user and encrypts it with symmetric key.

`byte[] enrollPIN (byte[] wrappedKey, byte[] PIN)`

currentMsg: current APDU message number

totalMsgs: total APDU messages that will be sent for the request

wrappedKey: the wrapped symmetric key $(key(S))_w$

PIN: 4 byte pin to be encrypted

returns: Encrypted PIN

Usage from Mobile Handset:

`byte[] enrollPIN (byte currentMsg, byte totalMsgs, byte[] wrappedKey, byte[] PIN)`

This method is used to get PIN from the user, encrypt it and return to mobile handset. Following APDU needs to be sent to SIM card:

`0x80 0x33 0x00 0x00 <lengthOfData> < currentMsg + totalMsgs + wrappedKey + PIN >`

CLA	INS	P1	P2	Lc	Data
0x80	0x33	0x20	0x00	0x50	< currentMsg + totalMsgs + wrappedKey + PIN >

CLA represents the CLA of the applet and is same for all command APDUs sent to the KMS applet.
INS represents `enrollPIN` method.
P1 represents length of `wrappedKey`
Lc represents the length of the Data part.
Data represents `currentMsg + totalMsgs + wrappedKey + PIN`

Returns:
Encrypted PIN

p. PIN Verification

This method is used to verify a pin.

Mobile application sends the encrypted PIN. SIM card asks for a PIN code. SIM card decrypts the encrypted PIN. SIM card verifies the inputted PIN with the decrypted one.

`byte verifyPIN (byte[] wrappedKey, byte[] encryptedPIN, byte[] PIN)`

currentMsg: current APDU message number

totalMsgs: total APDU messages that will be sent for the request

wrappedKey: the wrapped symmetric key $(key(S))_w$

encryptedPIN: an encrypted PIN (which is the result of `enrollPIN` function)

PIN: 4 byte pin to be verified with

returns: returns false or true. 0x00 for false; 0x01 for true

Usage from Mobile Handset:

`byte[] verifyPIN (byte[] wrappedKey, byte[] encryptedPIN, byte[] PIN)`

This method is used to get PIN from the user, verify it with the encrypted one. Following APDU needs to be sent to SIM card:

`0x80 0x34 0x20 0x00 <lengthOfData> < currentMsg + totalMsgs + wrappedKey + encryptedPIN + PIN >`

CLA	INS	P1	P2	Lc	Data
0x80	0x34	0x20	0x00	0x50	< currentMsg + totalMsgs + wrappedKey + encryptedPIN + PIN >

CLA represents the CLA of the applet and is same for all command APDUs sent to the KMS applet.
INS represents `verifyPIN` method.
P1 represents length of `wrappedKey`

Lc represents the length of the Data part.
Data represents `currentMsg + totalMsgs + wrappedKey + encryptedPIN + PIN`

Returns:
01 for true
00 for false

q. Verify (New method in Phase 4)

This method is used to verify a signature.

```
byte verify (byte currentMsg, byte totalMsgs, byte[] publicKey,
byte[] data, byte[] signature)
```

currentMsg: current APDU message number

totalMsgs: total APDU messages that will be sent for the request

publicKey: the RSA public key $(key(AS)_{public})$

data: the data to verify

signature: the previously generated signature

Note: Mobile handset should use sequence number in this method as described in Section 2, because SIM card may need to send the response in more than one APDU message. The details on how to use sequence number in this method is given in method specification in Section 2.

Note: Mobile application generally needs to send more than 256 bytes for this method. Thus it needs to send more than one messages. In every message, it needs to send the current message number and total messages. Examples are given below.

returns: returns false or true. 0x00 for false; 0x01 for true

Usage from Mobile Handset:

```
byte verify (byte currentMsg, byte totalMsgs, byte[] publicKey,
  byte[] data, byte[] signature )
```

This method is used to verify a signature. Following APDU needs to be sent to SIM card:

```
0x80 0x29 0x20 0x02 <lengthOfData> < currentMsg + totalMsgs
+ publicKey + data + signature >
```

CLA	INS	P1	P2	Lc	Data
0x80	0x35	0x20	0x02	0x50	< currentMsg + totalMsgs + publicKey + data + signature >

CLA represents the CLA of the applet and is same for all command APDUs sent to the KMS

applet
INS represents `verify` method
P1 represents the (length of `publicKey`**`/4`**`)`
P2 represents the (length of `data`**`/4`**`)`
Lc represents the length of the Data part
Data represents `currentMsg + totalMsgs + publicKey + data + signature`

Returns:
0x01 (true) if the signatures match
0x00 (false) if the signatures do not match