PV181: Training 5 – Java Card technology

Javacard overview

Sun Microsystems published the Java Card Platform Specification and the Java Card Development Kit, which includes a reference implementation based on the specification. The aim is to provide the basis for cross-platform and cross-vendor applet interoperability. The current version of the JavaCard specification is 2.2.2 [JC222]. JavaCard applet is Java-like application that is uploaded to a smart card and is executed by the Java Virtual Machine on the smart card.

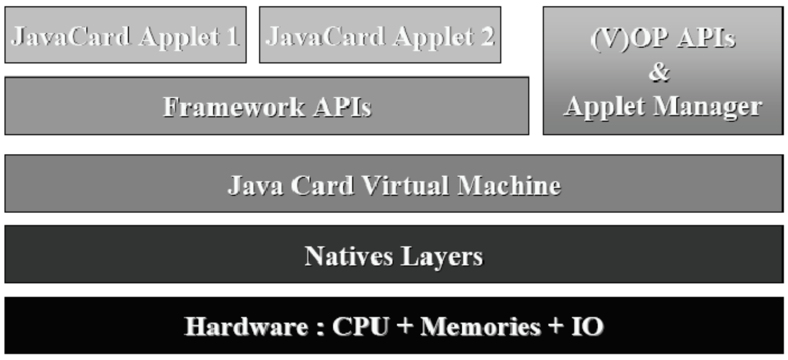

JavaCard Applet 1
JavaCard Applet 2
(V)OP APIs & Applet Manager
Framework APIs
Java Card Virtual Machine
Natives Layers
Hardware : CPU + Memories + IO

Figure 1 - JavaCard architecture overview

Java Card applications are compiled using common Java compilers. Due to limited memory resources and computing power, Java Cards do not support:

- dynamic class loading,
- security manager,
- threads and synchronization,
- garbage collection and object cloning,
- finalization,
- large primitive data types (float, double, long and char) and
- most of std. classes (most of the java.lang, Object and Throwable in limited form).

JavaCard security

The main security features of Java Card include:

- All the benefits of the Java language: data encapsulation, safe memory management, packages, etc.
- Applet isolation based on the Java Card firewall: applets cannot directly communicate with each other. Special interface for sharing objects Shareable must be implemented to allow cross applets interaction.
- Atomic operations using transaction mode: JCSystem.beginTransaction(), commitTransaction(), abortTransaction().
- Transient data, which guarantees that sensitive session data is wiped out: JCSystem.makeTransientByteArray().
- A rich cryptography API for encryption, digital signatures and message digests.
- Secure communication with the card reader, if the card is Open Platform compliant (secure messaging, security domains).

Creating a JavaCard applet (GemXpressoRADIII, JBuilder7)

The aim of this text is to create and compile a simple JavaCard applet. Detailed information can be found in JavaCard API documentation. Described techniques use GemXpresso RADIII and its plug-in into Borland JBuilder7.

Required development tools

- Development environment (IDE) for Java language (e.g. Borland JBuilder7)
- GemXpresso RADIII. Contains plug-in for JBilder7 and management environment JCardManager for loading applets onto smart card, sending APDU commands etc.
- JavaCard Development Kit 2.1.2

General information

Whole process consists of four logical steps:

1. Creation of the applet as a descendant of the class javacard.framework.Applet in arbitrarily development environment for Java language with respect to restrictions of the JavaCard platform (restricted primitive types, only basic classes from javacard.* package). Compilation using general Java compiler into *.class.
2. Conversion of *.class into *.jar using JavaCard converter. The code can be checked using JavaCard Off-Card Verifier for correctness (not necessary for testing purposes).
3. Loading file *.jar onto the smart card, installation of the included applet and registration in operating system of smart card. Done using appropriate interface like OpenPlatform.
4. Repeated usage of the installed applet. During the first step, applet with the given AID (unique identifier) is selected as the active one using SELECT command. All subsequent APDU commands are directly passed to the active applet.

Source code, compilation

Using wizard *File->New…->Gemplus->JavaCard applet*, a class with a selected name (*„Applet name“*) is created. Set element *type* to *„JavaCard Open Platform Applet“*, insert unique identification of package (*„Package AID“*) and unique identification of applet (*„Applet AID“*). In the next step, select the type of the card for which the conversion will be performed.

The result is a skeletal code of the applet that can be compiled and uploaded onto smart card immediately. The skeletal code contains basic functionality required for installation and registration of applet in the operating system of smart card.

Applet contains the following methods:

- *protected applet_name(byte[] buffer, short offset, byte length)* (constructor). It is called from the *install* method only once during installation of applet. It is preferable to initialize all needed structures, allocate all needed memory and objects that will be required by applet in the body of this method (memory can be already occupied later)
- *public static void install(byte[] bArray, short bOffset, byte bLength).* This method is called only once during the installation of the applet. Perform only calling of applet constructor.
- *public boolean select().* This method is called every time, when somebody tries to select this applet as the active one. Selection of an applet can be suppressed in this method by returning *false* value, for example because a required condition is not satisfied. Needed structures can be initialized or security conditions reset here. This method is called every time before applet can be used.
- *public void deselect().* This method is called when the terminal sends a command to set the applet as inactive (incoming commands will not be sent to this applet any

more until new select command is sent). It can be used for the cleanup of the code. WARNING: There is no guarantee that this method will be called! The method is called only during a correct deactivation of the applet, but not when smart card is suddenly removed from reader. That is why it is highly recommended to perform cleanup and initialization in the *select()* method.

- *public void process(APDU apdu)* . This method serves as the entrance gate for all APDU commands (except system-reserved) received by the card after SELECT command. Is properly to define own class of instructions (CLA), test the header of incoming APDU to this class value and return exception ISO7816.SW_CLA_NOT_SUPPORTED if it does not match. Branching based on the value of the INS parameter is performed using the switch statement and the APDU command (APDU object) is passed to the appropriate method. Actions inside *process()* method are under control of the applet programmer.

```
public class Foo extends javacard.framework.Applet
    protected Foo(byte[] buffer, short offset, byte length)   {
      // allocation of required memory and objects, initialization of structures
      m_byteArray = JCSystem.makeTransientByteArray((short) 128,  System.CLEAR_ON_DESELECT);
      // registration of instance
    }
    public static void install(byte[] bArray, short bOffset, byte bLength) throws ISOException  {
        new Foo (bArray, bOffset, (byte)bLength );
    }
    public boolean select()   {
        // <PUT YOUR SELECTION ACTION HERE>
        // return status of selection, if false, applet is not selected and cannot be used
        return true;
    }
     public void deselect()  {
        // <PUT YOUR DESELECTION ACTION HERE>
        return;
    }
    public void process(APDU apdu) throws ISOException  {
       // get the APDU buffer
       byte[] apduBuffer = apdu.getBuffer();

       // ignore the applet select command dispached to the process
       if (selectingApplet()) return;

        if (apduBuffer[ISO7816.OFFSET_CLA] == INS_CLA_FOO) {
          // APDU instruction parser
          switch ( apduBuffer[ISO7816.OFFSET_INS] ) {
              case INS_CARD_SET_KEY: SetKey(apdu); break;
              case INS_CARD_COPYINPUT: CopyInput(apdu); break;
              default: ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED); break;
            }
        }
        else ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
   }
```

Other methods can be defined as needed and are called from the *process()* method. If there is a need to work with incoming/outgoing data (usual), the following structure of the method is needed:

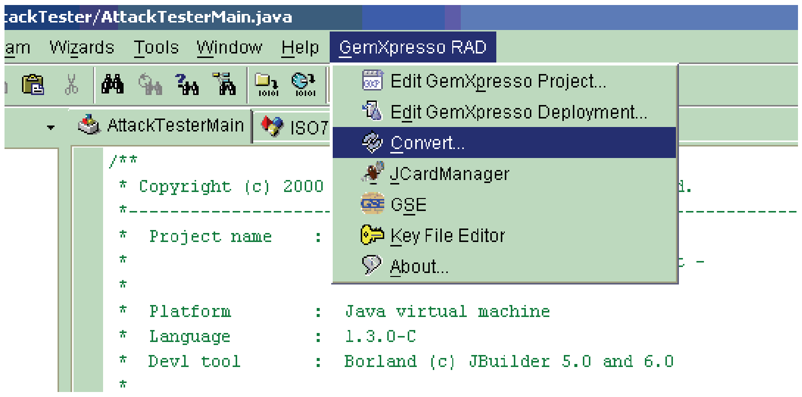
```
void Test(APDU apdu) {
        // reading of APDU header that contains CLA, INS, P1, P2, LC
        byte[]    apdubuf = apdu.getBuffer();

        // reading of incoming data (relevant only when LC > 0)
        short     dataLen = apdu.setIncomingAndReceive();

         // manipulating header (e.g. reading of CLA and P1)
         byte cla = apdubuf[ISO7816.OFFSET_CLA];
         byte p1 = apdubuf[ISO7816.OFFSET_P1];

         // sending the content of m_testArray with length ARRAY_LENGTH to SC output
         Util.arrayCopyNonAtomic(apdubuf, ISO7816.OFFSET_CDATA, m_testArray, (short) 0,
         ARRAY_LENGTH);
         apdu.setOutgoingAndSend(ISO7816.OFFSET_CDATA, ARRAY_LENGTH);
  }
```

The applet is compiled using Project->Make project and converted into a smart card compatible format. The conversion starts using the Convert command from the plug-in menu added by the GemXpresso RADIII into environment (viz. **Pict. A-2**). This menu can be also used for modification of settings of project, adding new types of card for conversion etc.



```
/**
 * Copyright (c) 2000
 *--------------------
 *  Project name    :
 *
 *
 *  Platform        :  Java virtual machine
 *  Language        :  1.3.0-C
 *  Devl tool       :  Borland (c) JBuilder 5.0 and 6.0
 *
```

Pict. A-2: Conversion of compiled code for smart card.

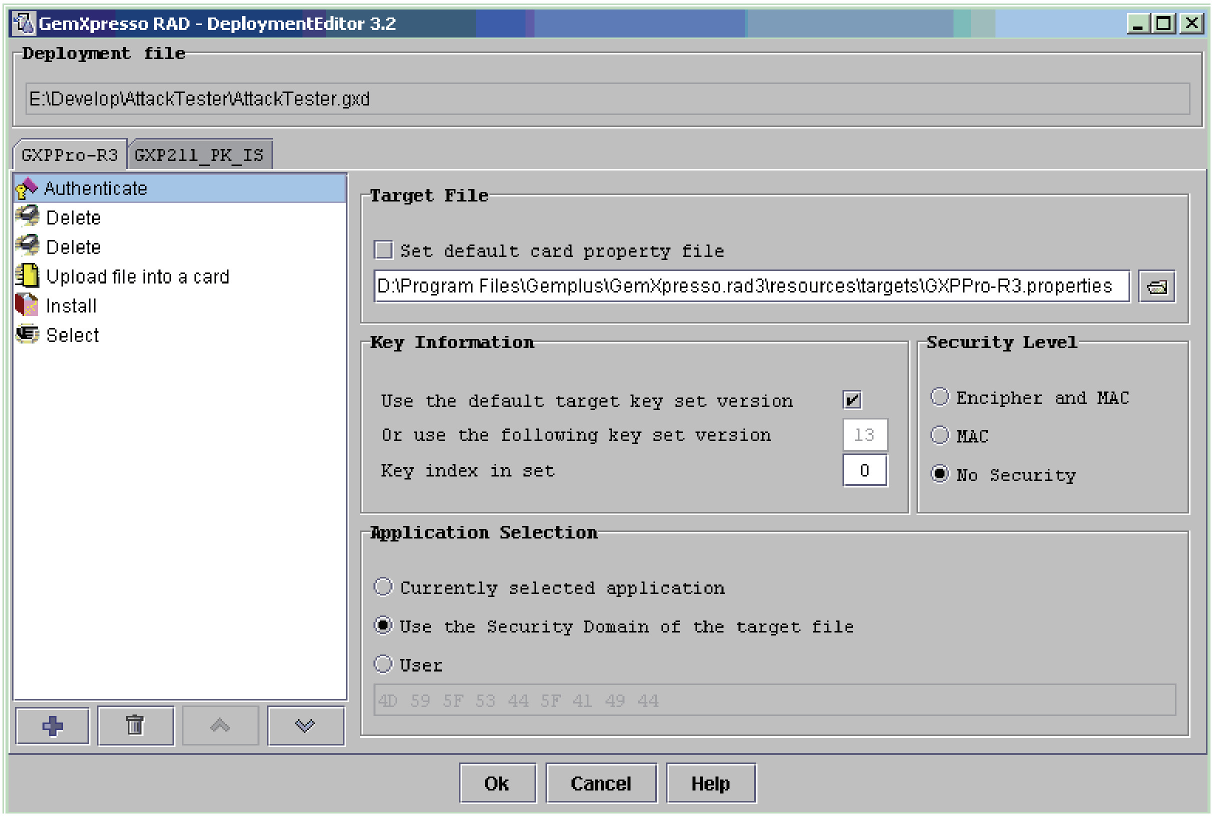
Applet uploading, installation

In this step we assume, that we already have compiled and converted the applet (files *.jar or *.sap (for smart card simulator)).

1. Run JCardManager environment
2. Insert the smart card or run the smart card simulator (Tools->Gse Gui)
3. Switch to tab OP 2.0.1

4. Authenticate against the smart card („Authenticate“ command). Smart cards used for testing purposes have the default file containing authentication information (diversification keys - e.g. for *GXP_E64PK* the file C*:\Program Files\Gemplus\RADIII\resources\targets\GXP_E64PK_MUNI.properties*)
5. If older version of applet with same AID is already present on the card, it must be removed using “Delete” command. Instances of the applet must also be removed. Remove instances of the applet FIRST and the package as SECOND. The package with active instance cannot be removed.
6. Upload the new package with your applet using “Upload file into a card“ command. Set the path to package (*.jar or *.sap) and unique identification of the package „Package AID“.
7. Install applet. „Package AID“ (see step 6.), „Applet AID“ and „Instance AID“ must be filled. Last two can be same. During installation, *install()* method of applet of your applet is invoked.

That steps will finish the installation of the applet and the applet is ready for use. The applet will resist on the smart card until it is explicitly removed. If the smart card simulator is used without saving the internal state (“clean” card every time), installation must be performed every time. Steps from 4 to 7 can be automated using GemXpresso Deployment (Deployment->New…).


GemXpresso RAD - DeploymentEditor 3.2

Deployment file: E:\Develop\AttackTester\AttackTester.gxd

Tabs: GXPPro-R3 | GXP211_PK_IS

Authenticate
Delete
Delete
Upload file into a card
Install
Select

Target File
[ ] Set default card property file
D:\Program Files\Gemplus\GemXpresso.rad3\resources\targets\GXPPro-R3.properties

Key Information
Use the default target key set version [✓]
Or use the following key set version: 13
Key index in set: 0

Security Level
( ) Encipher and MAC
( ) MAC
(●) No Security

Application Selection
( ) Currently selected application
(●) Use the Security Domain of the target file
( ) User
4D 59 5F 53 44 5F 41 49 44

Pict. A-3: GemXpresso Deployment.

Communication with applet

Installed applet with AID (e.g.. ‘41 74 74 61 63 6B 54 65 30 30 30 31‘) is assumed in this part. Switch to tab OP 2.0.1.

1. Send „Select“ command with AID of applet (41 74 74 61 63 6B 54 65 30 30 30 31). Return status <- 90 00 in log windows should be displayed as a confirmation of the successful selection of the applet as being active.
2. Send your commands using „Send APDU“ with parameters filled according to your needs. Card responses can be found in the log window.

Debugging using JBuilderu7 and Gemplus Simulator

The debugging process of a real smart card is difficult as there's no possibility to perform debug steps using the source code, no trace output can be displayed etc. GemXpresso RADIII environment contains a possibility to run smart card simulator for a particular type of a real smart card with debug features enabled. The debugging can be performed directly from Borland JBuilder7:

1. In project settings Project->Project Properties…->Run set item „Main class“ to 'com.gemplus.javacard.gse.Simulator' and „Application parameters“ to '-port 5000 -card GXPPro-R3' (GXPPro-R3 smart card will be simulated). For different type of card change string after *-card* statement and ALSO change type of linked library of simulator in Project->Project properties->Required libraries->GSE…
2. Compile and convert applet as usual.
3. Set breakpoints to the required positions in the code and run Run->Debug project. A simulator of the selected card is launched (GXPPro-R3 in our case). The type of the launched card is displayed in the log window of JBuilder7.
4. Upload and install the applet onto the simulator using the JCardManager and send *Select* command.
5. Send APDU that cause invocation of the method with our breakpoint.

JavaCard applet for PIN verification

The sample applet implements the following logical steps:

- Allocation of PIN object (OwnerPIN())
- Initial setting of the secret value of PIN (OwnerPIN.update())
- Verification of the correctness of the supplied PIN (OwnerPIN.check())
- Get remaining tries of PIN verification attempts (OwnerPIN.getTriesRemaining())
- Set tries counter to maximum value and unblock blocked PIN. (OwnerPIN.resetAndUnblock())

```
// CREATE PIN OBJECT (try limit == 5, max. PIN length == 4)
OwnerPIN m_pin = new OwnerPIN((byte) 5, (byte) 4);
// SET CORRECT PIN VALUE
m_pin.update(INIT_PIN, (short) 0, (byte) INIT_PIN.length);
// VERIFY CORRECTNESS OF SUPPLIED PIN
boolean correct = m_pin.check(array_with_pin, (short) 0, (byte) array_with_pin.length);
// GET REMAING PIN TRIES
byte j = m_pin.getTriesRemaining();
// RESET PIN RETRY COUNTER AND UNBLOCK IF BLOCKED
m_pin.resetAndUnblock();
```

JavaCard applet for encryption of the supplied data

The sample applet implements the following logical steps:

- Allocation and initialization of the key object (KeyBuilder.buildKey())
- Set key value (DESKey.setKey())
- Allocation and initialization of the object of cipher (Cipher. getInstance(), Cipher. init())
- Receive incoming data (APDU.setIncomingAndReceive())
- Encrypt or decrypt data (Cipher.update(), Cipher.doFinal())
- Send outgoing data (APDU. setOutgoingAndSend())

```
// …. INICIALIZATION SOMEWHERE (IN CONSTRUCT)
// CREATE DES KEY OBJECT
DESKey m_desKey = (DESKey) KeyBuilder.buildKey(KeyBuilder.TYPE_DES, KeyBuilder.LENGTH_DES,
false);
// SET KEY VALUE
m_desKey.setKey(array, (short) 0);

// CREATE OBJECTS FOR ECB CIPHERING
m_encryptCipher = Cipher.getInstance(Cipher.ALG_DES_ECB_NOPAD, false);
// INIT CIPHER WITH KEY FOR ENCRYPT DIRECTION
m_encryptCipher.init(m_desKey, Cipher.MODE_ENCRYPT);
//….

// ENCRYPT INCOMING BUFFER
void Encrypt(APDU apdu) {
    byte[]    apdubuf = apdu.getBuffer();
    short     dataLen = apdu.setIncomingAndReceive();

    // CHECK EXPECTED LENGTH (MULTIPLY OF 64 bites)
    if ((dataLen % 8) != 0) ISOException.throwIt(SW_CIPHER_DATA_LENGTH_BAD);

    // ENCRYPT INCOMING BUFFER
    m_encryptCipher.doFinal(apdubuf, ISO7816.OFFSET_CDATA, dataLen, m_ramArray, (short) 0);

    // COPY ENCRYPTED DATA INTO OUTGOING BUFFER
    Util.arrayCopyNonAtomic(m_ramArray, (short) 0, apdubuf, ISO7816.OFFSET_CDATA, dataLen);

    // SEND OUTGOING BUFFER
    apdu.setOutgoingAndSend(ISO7816.OFFSET_CDATA, dataLen);
}
```

JavaCard applet for hashing of the supplied data

JavaCard 2.2.2 standard describes hashing functions MD5, SHA-1, SHA-256 and RIPEMD160. Not all must be implemented by a particular smart card.

The sample applet implements the following logical steps:

- Allocation of the hashing object (MessageDigest.getInstance())

- Reset internal state of hash object (MessageDigest. reset ())
- Update intermediate hash value using incoming array (MessageDigest.update())
- Finalize and read hash value of data (MessageDigest.doFinal())

```
// CREATE SHA-1 OBJECT
MessageDigest m_sha1 = MessageDigest.getInstance(MessageDigest.ALG_SHA, false);

// RESET HASH ENGINE
m_sha1.reset();
// PROCESS ALL PARTS OF DATA
while (next_part_to_hash_available) {
 m_sha1.update(array_to_hash, (short) 0, (short) array_to_hash.length);
}
// FINALIZE HASH VALUE (WHEN LAST PART OF DATA IS AVAILABLE)
// AND OBTAIN RESULTING HASH VALUE
m_sha1.doFinal(array_to_hash, (short) 0, (short) array_to_hash.length, out_hash_array, (short) 0);
```

JavaCard applet for signing of the supplied data

JavaCard 2.2.2 standard describes various combination of signature functions based on asymmetric cryptography (RSA, DSA, ECDSA) and symmetric cryptography (MAC – DES, AES based). Again, not all must be implemented by a particular smart card.

The sample applet implements the following logical steps:

- Allocation of the key and signature objects (KeyBuilder.buildKey, new KeyPair, Signature.getInstance)
- On-card generation of key pair (KeyPair.genKeyPair())
- Obtaining references to private and public key (KeyPair.getPrivate/Public)
- Initialization of signature engine with private key (Signature.init)
- Performing signature operation (Signature.update, Signature.sign)

```
// CREATE RSA KEYS AND PAIR
m_privateKey = KeyBuilder.buildKey(KeyBuilder.TYPE_RSA_PRIVATE,KeyBuilder.LENGTH_RSA_1024,false);
m_publicKey = KeyBuilder.buildKey(KeyBuilder.TYPE_RSA_PUBLIC,KeyBuilder.LENGTH_RSA_1024,true);
m_keyPair = new KeyPair(KeyPair.ALG_RSA, (short) m_publicKey.getSize());

// STARTS ON-CARD KEY GENERATION PROCESS
m_keyPair.genKeyPair();
// OBTAIN KEY REFERENCES
m_publicKey = m_keyPair.getPublic();
m_privateKey = m_keyPair.getPrivate();

// CREATE SIGNATURE OBJECT
Signature m_sign = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1, false);
// INIT WITH PRIVATE KEY
m_sign.init(m_privateKey, Signature.MODE_SIGN);

// SIGN INCOMING BUFFER
signLen = m_sign.sign(apdubuf, ISO7816.OFFSET_CDATA, (byte) dataLen, m_ramArray, (byte) 0);
```

JavaCard applet for generating random data

JavaCard 2.2.2 standard defines two types of random generators within object RandomData: RandomData.ALG_SECURE_RANDOM and RandomData.ALG_PSEUDO_RANDOM. Sometimes, the ALG_PSEUDO_RANDOM is not implementated by the card.

The sample applet implements the following logical steps:

- Allocation of the random data object (RandomData.getInstance())
- Generation of random block with given length (RandomData.generateData())

```
private RandomData      m_rngRandom = null;

// CREATE RNG OBJECT
m_rngRandom = RandomData.getInstance(RandomData.ALG_SECURE_RANDOM);

// GENERATE RANDOM BLOCK WITH 16 BYTES
m_rngRandom.generateData(m_testArray1, (short) 0, ARRAY_ONE_BLOCK_16B);
```

Homework

Create a JavaCard applet and corresponding PC application capable to generate large amount of random data on SC (>> 240B) and sends it back to PC. The origin of random data will be authenticated – the applet also generates one RSA signature for the random data transferred within one session. Use ALG_RSA_SHA_PKCS1 as a signature algorithm. Process should consist:

1. PC initialize new gathering session of random data using command “GetRandomBlock()”. Signature engine in now reseted.
2. PC request random block with length signalized in P2 param (up to 240B) – command “GetRandomBlock()”.
3. SC internally generate random block and compute partial signature of transferred data (now only the hash, actually)
4. PC request next block using step 2.
5. When all blocks are generated and transferred to PC, data signature is requested using special command – GetRandomSign().

Notes: You may use example applications LabakApplet and LabakaAPDU. Generate new RSA keypair during applet installation (constructor). Use P1 of APDU header in step 1 and 2 to differentiate between first (initialization) call of GetRandomBlock() and remaining blocks. Use Signature.update() to partially sign actual block of random data except the last block. Use Signature.sign() to compute final signature. You don’t have to verify SC signature.

Note, that this exercise implementation still have a significant weakness – random data can be observed on communication link between SC and PC and then possibly misused by the attacker. If you are implementing the real application and data secrecy is needed, secure channel should be established forehand.

References

[JC222] JavaCard 2.2.2 http://java.sun.com/products/javacard/specs.html