View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Digital.CSIC

ECC programming in Java Card

V. Gayoso Martínez and L. Hernández Encinas
Institute of Physical and Information Technologies (ITEFI)
Spanish National Research Council (CSIC)
Madrid, Spain
Email: {victor.gayoso,luis}@iec.csic.es

***Abstract*—Elliptic Curve Cryptography (ECC) is a branch of public-key cryptography based on the arithmetic of elliptic curves. Given its mathematical characteristics, ECC is currently one of the best options for protecting sensitive information. The lastest version of the Java Card platform includes several classes related to elliptic curves. However, potential developers are discouraged by the peculiarities of its programming model and the scarce information available.**

In this work, we present an up to date and extensive review of the ECC support in Java Card. In addition to that, we offer to the reader the complete code of two applications that will allow programmers to understand and test the entire application development process in Java Card.

***Keywords*-elliptic curves; information security; Java Card; public key cryptography; smart cards;**

As it is well known, in 1985 Miller [1] and Koblitz [2] independently proposed a cryptosystem based on the ECDLP (Elliptic Curve Discrete Logarithm Problem). This field of cryptography is usually known as ECC (Elliptic Curve Cryptography). In comparison with other public-key cryptosystems, ECC uses significantly shorter keys to achieve the same level of security [3]. This makes ECC the perfect choice for devices with limited resources [4].

In 1996, the smart card sector witnessed the appearance of a new technology named Java Card. Java Card is the smallest of the Java platforms, and it allows to develop and install a specific type of Java-based application (called applet) in smart cards compliant with the Java Card specifications. This card technology is widely used in several sectors, for example in the cell phone and banking industries. In those sectors security is essential, so the integration of cryptographic capabilities is a typical application requirement.

Although Java Card is derived from the Java language, its programming model has several important particularities, so most Java programmers are not able to develop applets unless they are provided the proper training. Unfortunately, the number of learning resources about this technology is limited, which makes the development of Java Card applications a complex and resource-consuming operation for most software companies.

This contribution analyses the ECC capabilities in every Java Card version released so far, including all the classes and ECC functions implemented. In addition to that, we provide a complete code example that shows how to develop ECC applications in Java Card. In order to facilitate the understanding of the example, we have included the script needed to execute it and the console output obtained when running the applet in a simulator.

The rest of this paper is organized as follows: Section II presents a brief mathematical introduction to elliptic curves. In Section III, we review some important concepts about smart cards. Section IV describes the most relevant characteristics of Java Card, including the new features presented by each version. Section V details the ECC functionality included in the different Java Card releases. In Section VI, we offer two code examples which demonstrate how to create digital signatures and produce a shared secret using a key agreement procedure. Finally, Section VII summarizes our conclusions about this topic.

I. Elliptic Curve Cryptography

An elliptic curve E over the field $\mathbb{F}$ is a regular projective curve of genus 1 with at least one rational point ([5] and [6]). Every elliptic curve admits a canonical equation called the general Weierstrass form. That equation in homogeneous coordinates is

$$Y^2Z+a_1XYZ+a_3YZ^2 = X^3+a_2X^2Z+a_4XZ^2+a_6Z^3,$$

with $a_1, a_2, a_3, a_4, a_6 \in \mathbb{F}$ and $\Delta \neq 0$, where Δ is the discriminant of E.

The homogeneous Weierstrass equation defines a projective plane curve which has a special point, the point at infinity, which is denoted as $\mathcal{O} = [0 : 1 : 0]$. In principle that curve does not have to be elliptic, as it could have singular points. Due to that fact, the condition $\Delta \neq 0$ assures that the curve is regular, which is equivalent to stating that there are no curve points where the first derivatives of the function are cancelled [7].

In practice, instead of the general Weierstrass equation, two short Weierstrass forms depending on the characteristic of the finite field $\mathbb{F}_q$ are typically used:

- If the finite field is a prime field, i.e. $\mathbb{F} = \mathbb{F}_p$, where $p > 3$ is a prime number, the equation defining the (non-supersingular) elliptic curve becomes

$$y^2 = x^3 + ax + b.$$

Command APDU						
Header (mandatory)				Body (optional)		
CLA	INS	P1	P2	Lc	Data	Le

Figure 1. Command APDU

Response APDU		
Body (optional)	Result (mandatory)	
Data	SW1	SW2

Figure 2. Response APDU

- If the finite field is a binary field, i.e. $\mathbb{F} = \mathbb{F}_{2^m}$, where m is an integer number, then the equation of the (non-supersingular) elliptic curve is

$$y^2 + xy = x^3 + ax^2 + b.$$

II. SMART CARDS

A smart card is a plastic card with an embedded chip that controls the access to the stored data. The most widespread communication model for smart cards is composed by the byte oriented protocol T=0 and the APDU (Application Protocol Data Unit) elements.

APDUs, built according to the ISO/IEC 7816-3 [8] and 7816-4 [9] specifications, are the data packets exchanged between the external application and the card by means of a smart card reader. The card operating system is responsible for analysing any incoming APDU and redirect it to the application it is intended for. The operating system is also responsible for retrieving the response data from the card application and submit it to the external application using the card reader.

There are two types of APDUs: command and response. Figure 1 shows the format of command APDUs, which consist of a header and optionally a body with the following elements:

- CLA (1 byte): command class.
- INS (1 byte): specific instruction within the class.
- P1 (1 byte): first parameter associated to the instruction. It can be used to give more information about the instruction, or as input data.
- P2 (1 byte): second parameter associated to the instruction. As in the previous case, it can be used to give more information about the instruction, or as input data.
- Lc (1 byte, optional): number of bytes in the data field of the command. Since its highest value is `0xFF`, the maximum data length is 255 bytes, although some cards allow to send 256 bytes using the value `0x00`.
- Data (variable size, optional): information to be processed by the applet.
- Le (1 byte, optional): maximum number of bytes to be included in the data field of the response APDU.

In comparison, the format of any response APDU is simpler (see Figure 2), as it only includes the following items:

- Data (variable length, optional): information returned by the card application.
- SW1 (1 byte): first status byte, which provides general information about the result of the command execution.
- SW2 (1 byte): second status byte.

III. JAVA CARD

The first Java Card specification was presented in November 1996 by engineers working for the French company Schlumberger. Their goal was to create a technology that could combine the ease of development provided by the Java language and the security features associated to smart cards. Shortly after submitting the first draft of the Java Card API, Gemplus and Bull joined Schlumberger in order to constitute the Java Card Forum, a consortium created to evolve this technology.

After the presentation of Java Card 1.0, Sun Microsystems began to actively cooperate with those smart card manufacturers. The result was the announcement in November 1997 of Java Card 2.0. This new version represented a major milestone in Java Card, as it provided a mechanism for object-oriented programming and improved the level of detail in the specification of the application runtime environment.

Due to the need to adapt the capabilities of the Java language to the physical limitations of smart cards, since its inception it became clear that it was impossible to implement some of the Java features. For example, the `char`, `double`, `float`, and `long` types were not supported, multithreading was not allowed, and the dimension of data arrays was limited to one.

Java Card 2.1 was released in March 1999 and consisted of three specifications:

- Java Card API: defines the Java packages available for programmers.
- JCVM (Java Card Virtual Machine): specifies the subset of the Java language that can be used and the virtual machine needed for the execution of applets.
- JCRE (Java Card Runtime Environment): describes the applet runtime behaviour.

Java Card 2.2 was released in September 2002, and included the following novelties:

- JCRMI (Java Card Remote Method Invocation) implementation.

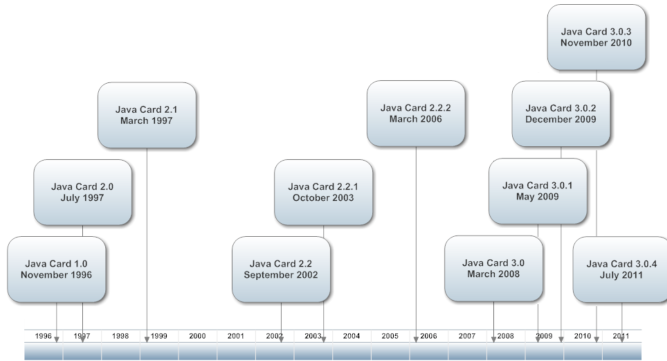

Figure 3. Java Card timeline

- Support of up to 4 logical channels.
- Improvement of the memory resource management.
- New classes for cryptographic algorithms.

In March 2006, Sun announced the availability of Java Card 2.2.2, which provided the following improvements:

- Support of up to 20 logical channels.
- Implementation of new cryptographic algorithms.
- New classes for biometric recognition technology.

Finally, in March 2008 the Java Card 3.0 specification was released. With the aim to adapt the Java Card technology to the needs of internet services, for the first time the specification was divided into two different editions:

- Connected Edition: introduces an enhanced runtime environment and a new virtual machine that provides network-oriented features such as the support for web applications using servlets.
- Classic Edition: represents an evolution of Java Card 2.2.2, including not only the correction of errors, but also several new features like the possibility to use some of the algorithms described in the NSA (National Security Agency) Suite B document [10]. In the remaining sections of this contribution, whenever we mention Java Card 3.0 we will implicitly refer to the Classic Edition.

As a summary of the previous information, Figure 3 shows the timeline of the different Java Card releases.

IV. ECC in Java Card

Java Card 2.2 was the first version that included ECC capabilities. More specifically, that version defined the following elements:

- New classes `ECKey`, `ECPrivateKey`, and `ECPublicKey` for the creation and management of public and private keys. Those classes can be used with elliptic curves defined over prime and binary fields.
- `KeyPair` class extension to allow the use of ECC key pairs.
- `KeyAgreement` class extension to include the key agreement functions ECDH (Elliptic Curve Diffie Hellman) and ECDHC (Elliptic Curve Diffie Hellman with Cofactor), with the peculiarity that the output of these functions is not the product $u \cdot V$ of the first user's private key u and the second user's public key V (and additionally the cofactor in the ECDHC case), but the result of feeding the first coordinate of the point $u \cdot V$ to the SHA-1 function [11].
- `KeyBuilder` class extension, which defines the permitted key lengths for ECC and other cryptosystems. In the case of elliptic curves over prime fields, the valid lengths in Java Card 2.2 were 112, 128, 160, and 192 bits, while in the case of curves defined over binary fields it was possible to use key lengths of 113, 131, 163, and 193 bits.
- `Signature` class extended with the implementation of ECDSA (Elliptic Curve Digital Signature Scheme) using the hash function SHA-1 [3].

Java Card versions 2.2.1 and 2.2.2 did not incorporate new ECC features. However, Java Card 3.0 included the following novelties:

- In elliptic curves over prime fields, key lengths of 224, 256, and 384 bits were available for the first time.
- The implementation of the functions ECDH and ECDHC was extended in order to accept new variants in which the output of the function is directly the first coordinate of the product $u \cdot V$ (optionally with the cofactor), eliminating the use of the SHA-1 function in the final step of the procedure.
- The implementation of ECDSA permitted to use that digital signature scheme in combination with the SHA-224, SHA -256, SHA-384, and SHA-512 functions.

Finally, the revision version 3.0.4 added the possibility to use 521-bit keys in curves defined over prime fields.

V. Code examples

The JCDK (Java Card Development Kit) is a software package that includes a complete development environment in which applications written for the Java Card platform can be developed and tested [12]. The JCDK includes a suite of tools along with a reference implementation written in C.

Before JCDK 3.0.2, the reference implementation did not include support for cryptographic functions. That was a significant drawback for developers interested in this technology. Fortunately, starting with JCDK 3.0.2, it was possible to use some of the cryptographic functions described in the Java Card API (more specifically, those listed in the *Development Kit User Guide* pertaining to the JCDK version in use).

Regarding ECC, the reference implementation in JCDK 3.0.2 allows to use the functions ECDH, ECDHC, and ECDSA only with curves defined over prime fields. From the

set of key lengths specified in the Java Card API, the ones available in the reference implementation are 112, 128, 160, and 192 bits. For each of those key lengths, the development kit implements one curve. The elliptic curves available in JCDK 3.0.2 are the following curves specified in the SECG SEC 2 standard [13]: secp112r1, secp128r1, secp160k1, and secp192k1.

Although the applet development and execution simulation steps can be performed using the command-line tools provided by the JCDK, Tim Boudreau has developed a connector for the NetBeans IDE (Integrated Development Environment). Interested readers can obtain the instructions for its installation at [14]. The minimum version requirements for installing this plugin are the following ones:

- NetBeans: 6.8.
- Java Development Kit: 6.
- Java Card Development Kit: 3.0.2.
- Java Card plugin for NetBeans: 1.3.

A. Key agreement example

Listing 1 contains the code that we have developed in order to demonstrate how to generate two pairs of 128-bit keys corresponding to users **U** and **V**, retrieve the most relevant information from them, and generate a shared secret using the key agreement function ECDH.

```
1   import javacard.framework.*;
2   import javacard.security.*;
3
4   public class JCDiffieHellman extends Applet
5   {
6    byte[] baTemp = new byte[255];
7    byte[] baPrivKeyU, baPrivKeyV, baPubKeyU, baPubKeyV;
8    short len;
9
10   KeyPair kpU, kpV;
11   ECPrivateKey privKeyU, privKeyV;
12   ECPublicKey pubKeyU, pubKeyV;
13
14   KeyAgreement ecdhU, ecdhV;
15
16   public static void install(byte[] bArray,
17    short bOffset, byte bLength)
18   {
19    new JCDiffieHellman();
20   }
21
22   protected JCDiffieHellman()
23   {
24    register();
25   }
26
27   public void process(APDU apdu)
28   {
29    byte[] buffer = apdu.getBuffer();
30
31    if (selectingApplet())
32     return;
33
34    if(buffer[ISO7816.OFFSET_CLA] != (byte)0x00)
35     ISOException.throwIt((short) 0x6660);
36
37    switch (buffer[ISO7816.OFFSET_INS])
38    {
39     case (byte)0xD1:
40      processINSD1(apdu);
41      return;
42     case (byte)0xD2:
43      processINSD2(apdu);
44      return;
45     case (byte)0xD3:
46      processINSD3(apdu);
47      return;
48     default:
49      ISOException.throwIt((short) 0x6661);
50    }
51   }
52
53   //////////////////////////////////////////////////
54   // INS D1 - KEY PAIR GENERATION                 //
55   // Generates a key pair for both users U and V //
56   // APDU EXAMPLE: 00D1000000                     //
57   //////////////////////////////////////////////////
58
59   private void processINSD1(APDU apdu)
60   {
61    try
62    {
63     kpU = new KeyPair(KeyPair.ALG_EC_FP,
64      KeyBuilder.LENGTH_EC_FP_128);
65     kpU.genKeyPair();
66     privKeyU = (ECPrivateKey) kpU.getPrivate();
67     pubKeyU = (ECPublicKey) kpU.getPublic();
68
69     kpV = new KeyPair(KeyPair.ALG_EC_FP,
70      KeyBuilder.LENGTH_EC_FP_128);
71     kpV.genKeyPair();
72     privKeyV = (ECPrivateKey) kpV.getPrivate();
73     pubKeyV = (ECPublicKey) kpV.getPublic();
74    }
75    catch(Exception exception)
76    {
77     ISOException.throwIt((short) 0xFFD1);
78    }
79   }
80
81   ///////////////////////////////////////
82   // INS D2 - PARAMETER DATA RETRIEVAL //
83   // P1: user                          //
84   // Values P1: 01: user U, 02: user V //
85   // P2: parameter to retrieve         //
86   // Values P2: 01: A, 02: B, 03: P,   //
87   //            04: public key,        //
88   //            05: private key        //
89   // APDU EXAMPLE: 00D2020300          //
90   ///////////////////////////////////////
91
92   private void processINSD2(APDU apdu)
93   {
94    byte buffer[] = apdu.getBuffer();
95
96    try
97    {
98     switch(buffer[3])
99     {
100     case 0x01: // Parameter A
101      if(buffer[2]==(byte)0x01)
102       len = pubKeyU.getA(baTemp,(short) 0);
103      else
104       len = pubKeyV.getA(baTemp,(short) 0);
105      apdu.setOutgoing();
106      apdu.setOutgoingLength((short) len);
107      apdu.sendBytesLong(baTemp,(short) 0, len);
108      break;
109
110     case 0x02: // Parameter B
111      if(buffer[2]==(byte)0x01)
112       len = pubKeyU.getB(baTemp,(short) 0);
113      else
114       len = pubKeyV.getB(baTemp,(short) 0);
115      apdu.setOutgoing();
116      apdu.setOutgoingLength((short) len);
117      apdu.sendBytesLong(baTemp,(short) 0, len);
118      break;
119
120     case 0x03: // Parameter P
121      if(buffer[2]==(byte)0x01)
```

```
      len = pubKeyU.getField(baTemp, (short) 0);
     else
      len = pubKeyV.getField(baTemp, (short) 0);
     apdu.setOutgoing();
     apdu.setOutgoingLength((short) len);
     apdu.sendBytesLong(baTemp,(short) 0,len);
     break;

    case 0x04: // Public key
     if(buffer[2]==(byte)0x01)
      len = pubKeyU.getW(baTemp,(short) 0);
     else
      len = pubKeyV.getW(baTemp,(short) 0);
     apdu.setOutgoing();
     apdu.setOutgoingLength((short) len);
     apdu.sendBytesLong(baTemp,(short) 0,len);
     break;

    case 0x05: // Private key
     if(buffer[2]==(byte)0x01)
      len = privKeyU.getS(baTemp,(short) 0);
     else
      len = privKeyV.getS(baTemp,(short) 0);
     apdu.setOutgoing();
     apdu.setOutgoingLength((short) len);
     apdu.sendBytesLong(baTemp,(short) 0,len);
     break;

    default:
     throw new IndexOutOfBoundsException();
   }
  }
  catch(Exception exception)
  {
   ISOException.throwIt((short) 0xFFD2);
  }
 }

 //////////////////////////////////////
 // INS D3 - SHARED SECRET GENERATION //
 // P1: user                         //
 // Values P1: 01: user U, 02: user V //
 // APDU EXAMPLE: 00D3010000         //
 //////////////////////////////////////

 private void processINSD3(APDU apdu)
 {
  byte buffer[] = apdu.getBuffer();

  try
  {
   switch(buffer[2])
   {
    case 0x01: // Process from U's standpoint
     len = privKeyU.getS(baTemp,(short) 0);
     baPrivKeyU = new byte[len];
     Util.arrayCopyNonAtomic(baTemp, (short)0,
      baPrivKeyU, (short)0, len);

     len = pubKeyV.getW(baTemp,(short) 0);
     baPubKeyV = new byte[len];
     Util.arrayCopyNonAtomic(baTemp,(short)0,
      baPubKeyV, (short)0, len);

     ecdhU = KeyAgreement.getInstance(KeyAgreement.
      ALG_EC_SVDP_DH, false);
     ecdhU.init(privKeyU);
     len = ecdhU.generateSecret(baPubKeyV,
      (short)0, len, baTemp, (short)0);

     apdu.setOutgoing();
     apdu.setOutgoingLength((short) len);
     apdu.sendBytesLong(baTemp,(short) 0, len);
     break;

    case 0x02: // Process from V's standpoint
     len = privKeyV.getS(baTemp,(short) 0);
     baPrivKeyV = new byte[len];
     Util.arrayCopyNonAtomic(baTemp,(short)0,
      baPrivKeyV, (short)0, len);

     len = pubKeyU.getW(baTemp,(short) 0);
     baPubKeyU = new byte[len];
     Util.arrayCopyNonAtomic(baTemp,(short)0,
      baPubKeyU, (short)0, len);

     ecdhV = KeyAgreement.getInstance(KeyAgreement.
      ALG_EC_SVDP_DH,false);
     ecdhV.init(privKeyV);
     len = ecdhV.generateSecret(baPubKeyU,
      (short)0, len, baTemp, (short)0);

     apdu.setOutgoing();
     apdu.setOutgoingLength((short) len);
     apdu.sendBytesLong(baTemp,(short) 0, len);
     break;

    default:
     throw new IndexOutOfBoundsException();
   }
  }
  catch(Exception exception)
  {
   ISOException.throwIt((short) 0xFFD2);
  }
 }
}
```

Listing 1. Java Card ECDH code example.

The most important classes and methods that appear in Listing 1 are the following [15]:

- `install()`: method called by the JCRE to create an instance of the applet.
- `register()`: method used by the application to register this applet instance with the JCRE and to assign the Java Card platform name of the applet as its instance AID (Application Identifier) bytes.
- `process()`: method called by the JCRE to process an incoming APDU command.
- `apdu.getBuffer()`: returns the APDU content as a byte array.
- `apdu.setOutgoing()`: method used to set the data transfer direction to outbound and to obtain the expected length of the response (APDU field Le).
- `apdu.sendBytesLong()`: sends the specified bytes from the output byte array.
- `KeyPair`: container for a key pair.
- `ECPrivateKey`: interface that allows the implementation of ECC private keys.
- `ECPublicKey`: interface that allows the implementation of ECC public keys.
- `KeyAgreement`: base class for key agreement algorithms.
- `genKeyPair()`: method that initializes the key objects encapsulated in the `KeyPair` instance in use with new key values.
- `getPrivate()`: returns a reference to the private key component of the `KeyPair` object in use.
- `getPublic()`: returns a reference to the public key component of the `KeyPair` object in use.

In order to simulate the applet execution, it is necessary to use a script with the command APDUs to be sent to the

smart card. Listing 2 shows the content of our script file. The `powerup` command prepares the APDUTool utility for reading command APDUs, so it must be executed before any command APDU is sent. In comparison, the `powerdown` command ends the APDUTool processing. For every command APDU in the script, the six bytes that compose the APDU correspond to the CLA, INS, P1, P2, Lc, and Le fields mentioned in Section III [16].

```
powerup;

// Applet selection (AID: C9AA4E15B3F6)
0x00 0xA4 0x04 0x00 0x06 0xC9 0xAA 0x4E 0x15 0xB3 0xF6
     0x7F;

//Command APDUs

0x00 0xD1 0x00 0x00 0x00 0x7F;

0x00 0xD2 0x01 0x01 0x00 0x7F;
0x00 0xD2 0x01 0x02 0x00 0x7F;
0x00 0xD2 0x01 0x03 0x00 0x7F;
0x00 0xD2 0x01 0x04 0x00 0x7F;
0x00 0xD2 0x01 0x05 0x00 0x7F;

0x00 0xD2 0x02 0x01 0x00 0x7F;
0x00 0xD2 0x02 0x02 0x00 0x7F;
0x00 0xD2 0x02 0x03 0x00 0x7F;
0x00 0xD2 0x02 0x04 0x00 0x7F;
0x00 0xD2 0x02 0x05 0x00 0x7F;

0x00 0xD3 0x01 0x00 0x00 0x7F;
0x00 0xD3 0x02 0x00 0x00 0x7F;

powerdown;
```

Listing 2. Applet execution script.

The result of running the script is the following sequence of command and response APDUs:

```
→  00A4040006C9AA4E15B3F6
←  9000
→  00D1000000
←  9000
→  00D2010100
←  FFFFFFFDFFFFFFFFFFFFFFFFFFFFFFFC9000
→  00D2010200
←  E87579C11079F43DD824993C2CEE5ED39000
→  00D2010300
←  FFFFFFFDFFFFFFFFFFFFFFFFFFFFFFFF9000
→  00D2010400
←  0479D69944F614C7AC9C6B5DF66C391AE2F77F
   04CBF17257CE92F5D791B9B7533C9000
→  00D2010500
←  595DA05E618DA5A664EF6A931272F5039000
→  00D2020100
←  FFFFFFFDFFFFFFFFFFFFFFFFFFFFFFFC9000
→  00D2020200
←  E87579C11079F43DD824993C2CEE5ED39000
→  00D2020300
←  FFFFFFFDFFFFFFFFFFFFFFFFFFFFFFFF9000
→  00D2020400
←  04620044FA3892038A9C3ADB194916E31F0112
   9E2429B92B75037979D17C1D6CD79000
→  00D2020500
←  835DC74BEB36D19C28E6474A4D400E0E9000
→  00D3010000
←  248B7E259095F53613641F1DD27DB61768D946
   D79000
→  00D3020000
←  248B7E259095F53613641F1DD27DB61768D946
   D79000
```

As it can be observed, **U**'s private key is `0x595DA05E618DA5A664EF6A931272F503`, while the serialization of his public key is `0x0479D69944F614C7AC9C6B5DF66C391AE2F77F04CBF17257CE92F5D791B9B7533C`. Besides, **V**'s private and public key are `0x835DC74BEB36D19C28E6474A4D400E0E` and `0x04620044FA3892038A9C3ADB194916E31F01129E2429B92B75037979D17C1D6CD7`, respectively. Finally, the value of the shared secret is `0x248B7E259095F53613641F1DD27DB61768D946D7`.

B. Digital signature example

The code of Listing 3 allows to generate a digital signature associated to the text "ECDSA example", and to validate the validity of that signature, using a 192-bit key pair.

The goal of using a predefined key pair in the example consists in simulating the data persistence of real smart cards. Besides, instead of codifying the text to be signed, we have developed the applet so it signs the string that is sent to the card using one of the valid APDUs for this applet. Regarding the verification process, it is necessary to send to the card both the digital signature and the original text.

```
1  import javacard.framework.*;
2  import javacard.security.*;
3
4  public class JCECDSA extends Applet
5  {
6
7   byte[] baTemp = new byte[255];
8   byte[] baSignature = new byte[255];
9   byte[] baText = new byte[255];
10  byte[] baPrivKey = {(byte)0x33, (byte)0x4a,
11    (byte)0x6a, (byte)0xa1, (byte)0xd5, (byte)0x42,
12    (byte)0xc3, (byte)0x12, (byte)0xbd, (byte)0xfa,
13    (byte)0x70, (byte)0x61, (byte)0x99, (byte)0xb4,
14    (byte)0x11, (byte)0xf7, (byte)0xa8, (byte)0xdd,
15    (byte)0xcf, (byte)0xaf, (byte)0x56, (byte)0x3a,
16    (byte)0x7c, (byte)0xb8};
17  byte[] baPubKey =  {(byte)0x04, (byte)0x4e,
18    (byte)0x0d, (byte)0xb7, (byte)0xd8, (byte)0x81,
19    (byte)0x39, (byte)0xee, (byte)0x2a, (byte)0x4c,
20    (byte)0xd4, (byte)0x75, (byte)0x47, (byte)0x6b,
21    (byte)0x62, (byte)0x9c, (byte)0x10, (byte)0x41,
22    (byte)0x9e, (byte)0x3d, (byte)0xa8, (byte)0x35,
23    (byte)0x44, (byte)0x5f, (byte)0x50, (byte)0x4c,
24    (byte)0x55, (byte)0x54, (byte)0x40, (byte)0xc4,
25    (byte)0x16, (byte)0xfa, (byte)0x2d, (byte)0xde,
26    (byte)0xd7, (byte)0x67, (byte)0xf5, (byte)0xea,
27    (byte)0x0d, (byte)0xbc, (byte)0x98, (byte)0x49,
28    (byte)0x7e, (byte)0x95, (byte)0x47, (byte)0xb0,
29    (byte)0xb8, (byte)0x09, (byte)0x63};
30
31  short len, lenText, lenSignature;
32  boolean result = false;
33
34  KeyPair kp;
35  ECPublicKey pubKey;
36  ECPrivateKey privKey;
37  Signature ecdsa;
38
```

```
39  public static void install(byte[] bArray,
40   short bOffset, byte bLength)
41  {
42   new JCECDSA();
43  }
44
45  protected JCECDSA()
46  {
47   register();
48  }
49
50  public void process(APDU apdu)
51  {
52   byte[] buffer = apdu.getBuffer();
53
54   if (selectingApplet())
55    return;
56
57   if(buffer[ISO7816.OFFSET_CLA] != (byte)0x00)
58    ISOException.throwIt((short) 0x6660 );
59
60   switch (buffer[ISO7816.OFFSET_INS])
61   {
62    case (byte)0xD1:
63     processINSD1(apdu);
64     return;
65    case (byte)0xD2:
66     processINSD2(apdu);
67     return;
68    case (byte)0xD3:
69     processINSD3(apdu);
70     return;
71    case (byte)0xD4:
72     processINSD4(apdu);
73     return;
74    default:
75     ISOException.throwIt((short) 0x6661 );
76   }
77  }
78
79  /////////////////////////////////////////
80  // INS D1 - KEY PAIR GENERATION         //
81  // Generates a new key pair of 192 bits //
82  // APDU EXAMPLE: 00D1000000             //
83  /////////////////////////////////////////
84
85  private void processINSD1(APDU apdu)
86  {
87   try
88   {
89    kp = new KeyPair(KeyPair.ALG_EC_FP,
90     KeyBuilder.LENGTH_EC_FP_192);
91    kp.genKeyPair();
92    privKey = (ECPrivateKey) kp.getPrivate();
93    privKey.setS(baPrivKey, (short)0,
94     (short)baPrivKey.length);
95    pubKey = (ECPublicKey) kp.getPublic();
96    pubKey.setW(baPubKey, (short)0,
97     (short)baPubKey.length);
98    ecdsa = Signature.getInstance(
99     Signature.ALG_ECDSA_SHA, false);
100  }
101  catch(Exception exception)
102  {
103   ISOException.throwIt((short) 0xFFD1);
104  }
105 }
106
107 /////////////////////////////////////////////
108 // INS D2 - SIGNATURE                      //
109 // DATA: string to be signed               //
110 // APDU EXAMPLE: 00D20000080102030405060708 //
111 /////////////////////////////////////////////
112
113 private void processINSD2(APDU apdu)
114 {
115  byte buffer[] = apdu.getBuffer();
116  short numBytesInput =
117   apdu.setIncomingAndReceive();
118  lenText = 0;
119
120  while (numBytesInput > 0)
121  {
122   Util.arrayCopyNonAtomic(buffer,
123    ISO7816.OFFSET_CDATA, baText,
124    lenText, numBytesInput);
125   lenText += numBytesInput;
126   numBytesInput =
127    apdu.receiveBytes(ISO7816.OFFSET_CDATA);
128    }
129
130  try
131  {
132   ecdsa.init(privKey, Signature.MODE_SIGN);
133   len = ecdsa.sign(baText, (short)0, lenText,
134    baSignature, (short)0);
135
136   apdu.setOutgoing();
137   apdu.setOutgoingLength((short) len);
138   apdu.sendBytesLong(baSignature, (short) 0, len);
139  }
140  catch(Exception exception)
141  {
142   ISOException.throwIt((short) 0xFFD2);
143  }
144 }
145
146 /////////////////////////////////////////////
147 // INS D3 - SIGNATURE VERIFICATION         //
148 // P1: operation                           //
149 // Values P1: 01: text load                //
150 //            02: signature load           //
151 // APDU EXAMPLE: 00D301000401020304        //
152 // APDU EXAMPLE: 00D3020006010203040506    //
153 /////////////////////////////////////////////
154
155 private void processINSD3(APDU apdu)
156 {
157
158  byte buffer[] = apdu.getBuffer();
159  short numBytesInput =
160   apdu.setIncomingAndReceive();
161
162  if(buffer[2]==(byte)0x01) // Text load
163  {
164   lenText = 0;
165   while (numBytesInput > 0)
166   {
167    Util.arrayCopyNonAtomic(buffer,
168     ISO7816.OFFSET_CDATA, baText,
169     lenText, numBytesInput);
170    lenText += numBytesInput;
171    numBytesInput =
172     apdu.receiveBytes(ISO7816.OFFSET_CDATA);
173    return;
174   }
175  }
176
177  if(buffer[2]==(byte)0x02) // Signature load
178  {
179   lenSignature = 0;
180   while (numBytesInput > 0)
181   {
182    Util.arrayCopyNonAtomic(buffer,
183     ISO7816.OFFSET_CDATA, baSignature,
184     lenSignature, numBytesInput);
185    lenSignature += numBytesInput;
186    numBytesInput =
187     apdu.receiveBytes(ISO7816.OFFSET_CDATA);
188    return;
189   }
190  }
191  ISOException.throwIt((short) 0xFFD3);
192
193 }
194
195 ///////////////////////////////////
196 // INS D4 - SIGNATURE VERIFICATION //
197 // APDU EXAMPLE: 00D4000000        //
198 ///////////////////////////////////
```

```
199
200  private void processINSD4(APDU apdu)
201  {
202   try
203   {
204    ecdsa.init(pubKey, Signature.MODE_VERIFY);
205    result = ecdsa.verify(baText, (short)0,
206      lenText, baSignature, (short)0, lenSignature);
207   }
208   catch(Exception exception)
209   {
210    ISOException.throwIt((short) 0xFFD4);
211   }
212
213   if(result)
214    return;
215   else
216    ISOException.throwIt((short) 0xFFD5);
217  }
218 }
```

Listing 3. Java Card ECDSA code example.

In addition to the classes and methods described regarding the key agreement example, the new elements that appear in Listing 3 are the following [15]:

- `Signature`: base class for digital signature algorithms.
- `init()`: method that initializes the `Signature` object with the appropriate key and mode (sign or verify).
- `sign()`: generates the signature of the input data.
- `verify()`: verifies the signature of the input data against the supplied signature.

Listing 4 contains the test script for the JCECDSA applet.

```
powerup;

// Applet selection (AID: C9AA4E15B3A2)
0x00 0xA4 0x04 0x00 0x06 0xC9 0xAA 0x4E 0x15 0xB3 0xA2
     0x7F;

0x00 0xD1 0x00 0x00 0x00 0x7F;

0x00 0xD2 0x00 0x00 0x0D 0x45 0x43 0x44 0x53 0x41 0x20
     0x65 0x78 0x61 0x6D 0x70 0x6C 0x65 0x7F;

0x00 0xD3 0x01 0x00 0x0D 0x45 0x43 0x44 0x53 0x41 0x20
     0x65 0x78 0x61 0x6D 0x70 0x6C 0x65 0x7F;

0x00 0xD3 0x02 0x00 0x37 0x30 0x35 0x02 0x18 0x0b 0x7c
     0x83 0x11 0x0c 0xe0 0xb7 0xba 0xc5 0x98 0xc9 0x73
     0x0a 0x11 0x3b 0x04 0x49 0xeb 0xb6 0x35 0x80 0x20
     0x6b 0x2f 0x02 0x19 0x00 0xe5 0x66 0x9c 0x57 0x97
     0xa2 0xd6 0x1d 0x2f 0x3e 0xdd 0x29 0x29 0x93 0x85
     0x7e 0xd4 0x8b 0xfa 0xec 0xb4 0x7c 0x02 0x70 0x7F;

0x00 0xD4 0x00 0x00 0x00 0x7F;

powerdown;
```

Listing 4. Applet execution script.

Below is included the command and response APDU sequence obtained when executing the previous script:

```
→  00A4040006C9AA4E15B3A2
←  9000
→  00D1000000
←  9000
→  00D200000D4543445341206578616D706C65
←  37303502180B7C83110CE0B7BAC598C9730A11
   3B0449EBB63580206B2F021900E5669C5797A2
   D61D2F3EDD292993857ED48BFAECB47C027090
   00
→  00D301000D4543445341206578616D706C65
←  9000
→  00D3020037303502180B7C83110CE0B7BAC598
   C9730A113B0449EBB63580206B2F021900E566
   9C5797A2D61D2F3EDD292993857ED48BFAECB4
   7C0270
←  9000
→  00D4000000
←  9000
```

In the previous example, the user's private key is `0x334A6AA1D542C312BDFA706199B411F7A8DDCFAF563A7CB8`, while the serialization of the public key is `0x044E0DB7D88139EE2A4CD475476B629C10419E3DA835445F504C555440C416FA2DDED767F5EA0DBC98497E9547B0B80963`. Given the input message "ECDSA example", coded as `0x4543445341206578616D706C65`, the signature generated by the procedure is `0x37303502180B7C83110CE0B7BAC598C9730A113B0449EBB63580206B2F021900E5669C5797A2D61D2F3EDD292993857ED48BFAECB47C0270`.

VI. Conclusion

Java Card is a technology that has benefited from the success of the Java language. Its object-oriented model allows smart card programmers to develop interoperable applets that can be deployed on smart cards independently of their manufacturer. However, Java Card's learning curve is steeper than Java's, so only a minority of Java programmers are attracted to Java Card.

Regarding ECC, Java Card 2.2 was the first version that included classes and functions supporting elliptic curves. The latest release, Java Card 3.0, has incremented the support for ECC, offering a range of key lengths suitable for any commercial deployment.

In this contribution, we have provided all the information needed by any Java programmer to start developing ECC applets, including two working examples that allow users to perform key exchanges with the ECDH algorithm, and digital signatures with the ECDSA procedure.

Given the current trends in information security, we believe that implementing ECC applications in smart cards will be an attractive option for many companies willing to create new security services.

Acknowledgment

This work has been partially supported by Ministerio de Ciencia e Innovación (Spain) under the grant TIN2011-22668.

REFERENCES

[1] V. S. Miller, "Use of elliptic curves in cryptography," *Lecture Notes Comput. Sci.*, vol. 218, pp. 417–426, 1986.

[2] N. Koblitz, "Elliptic curve cryptosytems," *Math. Comp.*, vol. 48, no. 177, pp. 203–209, 1987.

[3] *Digital Signature Standard (DSS)*, NIST Std. FIPS 186-3, 2009.

[4] V. Gayoso Martínez, F. Hernández Álvarez, L. Hernández Encinas, and C. Sánchez Ávila, "Analysis of ECIES and other cryptosystems based on elliptic curves," *J. Inform. Assurance and Security*, vol. 6, no. 4, pp. 285–293, 2011.

[5] N. Koblitz, *Algebraic Aspects of Cryptography*. New York, NY, USA: Springer-Verlag, 1998.

[6] J. H. Silverman, *The Arithmetic of Elliptic Curves*. New York, NY, USA: Springer-Verlag, 2nd ed., 2009.

[7] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. New York, NY, USA: Springer-Verlag, 2004.

[8] *Identification cards – Integrated circuit cards – Part 3: Cards with contacts – Electrical interface and transmission protocols*, ISO/IEC Std. 7816-3, 3rd ed., 2006.

[9] *Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange*, ISO/IEC Std. 7816-4, 3rd ed., 2013.

[10] National Security Agency. (2005) NSA Suite B cryptography. http://www.nsa.gov/ia/programs/suiteb_cryptography/index.shtml.

[11] *Secure Hash Standard*, NIST Std. FIPS 180-4, 2012.

[12] Oracle Corp. (2011) Java Card SDK. http://www.oracle.com/technetwork/java/javame/javacard/download/devkit/index.html.

[13] *Recommended elliptic curve domain parameters*, SECG Std. SEC 2 version 1.0, 2000, http://www.secg.org/download/aid-784/sec2-v2.pdf.

[14] Oracle Corp. (2013) Java Card development quick start guide. https://netbeans.org/kb/docs/javame/java-card.html.

[15] ——. (2011) Java Card Classic platform specification 3.0.4. http://www.oracle.com/technetwork/java/javacard/specs-jsp-136430.html.

[16] ——. (2001) Developing a Java Card applet. http://www.oracle.com/technetwork/java/javacard/applet-136808.html.