Developing ECC applications in Java Card

Conference Paper · December 2013

DOI: 10.1109/ISIAS.2013.6947743

Víctor Gayoso Martínez

Spanish National Research Council

Developing ECC applications 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 an application 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**

I. INTRODUCTION

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 a complete code example which demonstrates how to generate a key pair and create a shared secret using a key agreement procedure. Finally, Section VII summarizes our conclusions about this topic.

II. 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

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

- 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

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

III. 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. Command APDUs 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, 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.

IV. 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.
- 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.

V. 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.

VI. CODE EXAMPLE

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: 3.02.
- Java Card plugin for NetBeans: 1.3.

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.

```
import javacard.framework.*;
import javacard.security.*;

public class JCDiffieHellman extends Applet
{
 byte[] baTemp = new byte[255];
 byte[] baPrivKeyU, baPrivKeyV, baPubKeyU, baPubKeyV;
 short len;

 KeyPair kpU, kpV;
 ECPrivateKey privKeyU, privKeyV;
 ECPublicKey pubKeyU, pubKeyV;

 KeyPair kp;
 ECPublicKey pubKey;
 ECPrivateKey privKey;

 KeyAgreement ecdhU, ecdhV;

 public static void install(byte[] bArray,
  short bOffset, byte bLength)
 {
  new JCDiffieHellman();
```

```
}

protected JCDiffieHellman()
{
 register();
}

public void process(APDU apdu)
{
 byte[] buffer = apdu.getBuffer();

 if (selectingApplet())
  return;

 if(buffer[ISO7816.OFFSET_CLA] != (byte)0x00)
  ISOException.throwIt((short) 0x6660);

 switch (buffer[ISO7816.OFFSET_INS])
 {
  case (byte)0xD1:
   processINSD1(apdu);
   return;
  case (byte)0xD2:
   processINSD2(apdu);
   return;
  case (byte)0xD3:
   processINSD3(apdu);
   return;
  default:
   ISOException.throwIt((short) 0x6661);
 }
}

//////////////////////////////////////////////////
// INS D1 - KEY PAIR GENERATION                 //
// Generates a key pair for both users U and V //
// APDU EXAMPLE: 00D1000000                     //
//////////////////////////////////////////////////

private void processINSD1(APDU apdu)
{
 try
 {
  kpU = new KeyPair(KeyPair.ALG_EC_FP,
   KeyBuilder.LENGTH_EC_FP_128);
  kpU.genKeyPair();
  privKeyU = (ECPrivateKey) kpU.getPrivate();
  pubKeyU = (ECPublicKey) kpU.getPublic();

  kpV = new KeyPair(KeyPair.ALG_EC_FP,
   KeyBuilder.LENGTH_EC_FP_128);
  kpV.genKeyPair();
  privKeyV = (ECPrivateKey) kpV.getPrivate();
  pubKeyV = (ECPublicKey) kpV.getPublic();
 }
 catch(Exception exception)
 {
  ISOException.throwIt((short) 0xFFD1);
 }
}

///////////////////////////////////////
// INS D2 - PARAMETER DATA RETRIEVAL //
// P1: user                          //
// Values P1: 01: user U, 02: user V //
// P2: parameter to retrieve         //
// Valores P2: 01: A, 02: B, 03: P,  //
//             04: public key,       //
//             05: private key       //
// APDU EXAMPLE: 00D2020300          //
///////////////////////////////////////

private void processINSD2(APDU apdu)
{
 byte buffer[] = apdu.getBuffer();

 try
 {
  switch(buffer[3])
  {
   case 0x01: // Parameter A
    if(buffer[2]==(byte)0x01)
     len = pubKeyU.getA(baTemp,(short) 0);
    else
     len = pubKeyV.getA(baTemp,(short) 0);
    apdu.setOutgoing();
    apdu.setOutgoingLength((short) len);
    apdu.sendBytesLong(baTemp,(short) 0, len);
    break;

   case 0x02: // Parameter B
    if(buffer[2]==(byte)0x01)
     len = pubKeyU.getB(baTemp,(short) 0);
    else
     len = pubKeyV.getB(baTemp,(short) 0);
    apdu.setOutgoing();
    apdu.setOutgoingLength((short) len);
    apdu.sendBytesLong(baTemp,(short) 0, len);
    break;

   case 0x03: // Parameter P
    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);
```

```java
    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 code example.

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.

```
//Test script

powerup;

// Applet selection (instance ID: 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`.

VII. 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. 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 secure 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] National Institute of Standards and Technology, *Digital Signature Standard (DSS)*, NIST, Federal Information Processing Standard Publication, 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, 2^{nd} ed., 2009.

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

[8] Institute of Electrical and Electronics Engineers, *Identification cards – Integrated circuit cards – Part 3: Cards with contacts – Electrical interface and transmission protocols*, ISO/IEC 7816-3, 3^{rd} ed., 2006.

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

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

[11] National Institute of Standards and Technology, *Secure Hash Standard*, NIST, Federal Information Processing Standard Publication, FIPS 180-4, 2012. [Online]. Available: http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

[12] Oracle Corp., *Java Card SDK*, 2013, http://www.oracle.com/technetwork/java/javame/javacard/download/devkit/index.html.

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

[14] Oracle Corp., *Java Card Development Quick Start Guide*, 2013, https://netbeans.org/kb/docs/javame/java-card.html.