Implementing ECC with Java Standard Edition 7

V. Gayoso Martínez[1] and L. Hernández Encinas[2]

[1,2]Information Security Institute (ISI), Spanish National Research Council (CSIC), Madrid, Spain

[1]victor.gayoso@iec.csic.es; [2]luis@iec.csic.es

***Abstract*- Elliptic Curve Cryptography is one of the best options for protecting sensitive information. The lastest version of the Java platform includes a cryptographic provider, named SunEC, that implements some elliptic curve operations and protocols. However, potential users of this provider are limited by the lack of information available.**

In this work, we present an extensive review of the SunEC provider and, in addition to that, we offer to the reader the complete code of three applications that will allow programmers to generate key pairs, perform key exchanges, and produce digital signatures with elliptic curves in Java.

Keywords-Elliptic Curves; Information Security; Java; Public Key Cryptography

I. INTRODUCTION

The development of public key cryptography by Diffie and Hellman in 1976 [1] represented a major milestone in the history of modern security, opening the door to new and revolutionary cryptosystems. In 1985, Miller [2] and Koblitz [3] 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).

Almost in parallel, the technology sector witnessed the emergence of a promising programming language: Java. Before the version known as J2SE (Java 2 Standard Edition) 5.0, this programming language did not include specific classes for ECC. Developers willing to use those algorithms were forced to use software from third parties that was not compatible with code from other vendors. In J2SE 5.0 and the next version, Java SE 6, some classes and interfaces were included in order to facilitate the implementation of ECC applications. However, it was still necessary to use third party cryptographic engines in order to access all the power that ECC can provide to Java applications.

With the release of Java SE 7, Oracle Corporation provided a cryptographic engine, called SunEC, which supports ECC functionality off the shelf. However, the existing documentation about this cryptographic provider is quite limited, and no sample code is currently available, which makes it difficult to be used by novel programmers.

This contribution analyses the cryptographic features of SunEC, including aspects not available in the public documentation, such as the complete list of elliptic curves supported by SunEC. In addition to that, we provide several code examples to demonstrate the possibilities of ECC applications developed with Java. In order to facilitate the comprehension of the code, we have included the console screenshots obtained when running the applications.

The rest of this paper is organized as follows: Section II presents a brief mathematical introduction to elliptic curves. Section III describes the most important characteristics of Java, including its security model. Section IV analyses the distinctive features of SunEC. In Section V, we offer several code examples which implement three basic ECC operations: generating a key pair, making a key exchange, and creating a digital signature. Finally, Section VI summarizes the most relevant conclusions about this topic.

II. ELLIPTIC CURVE CRYPTOGRAPHY

A. Basic Definition

An elliptic curve E over the field $\mathbb{F}$ is a regular projective curve of genus 1 with at least one rational point [4, 5]. Every elliptic curve admits a canonical equation called the general Weierstrass form. That equation in homogeneous coordinates is

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

DOI: 10.5963/IJCSAI0304002

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 and can be computed in the following way [6]:

$$\begin{aligned}
\Delta &= -d_2^2 d_8 - 8d_4^3 - 27d_6^2 + 9d_2 d_4 d_6,\\
d_2 &= a_1^2 + 4a_2,\\
d_4 &= 2a_4 + a_1 a_3,\\
d_6 &= a_3^2 + 4a_6,\\
d_8 &= a_1^2 a_6 + 4a_2 a_6 - a_1 a_3 a_4 + a_2 a_3^2 - a_4^2.
\end{aligned}$$

The general Weirstrass equation is usually expressed in non-homogeneous form, where the relationship between both equations is given by $f(x, y) = F(x, y, 1)$ and $F(X, Y, Z) = f(X/Z, Y/Z) \cdot Z^3$, which produces the following affine equation:

$$E : y^2 + a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6.$$

The general homogeneous Weierstrass equation defines a projective plane curve which has a special point, called the point at infinity and 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 imposed 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].

B. Elliptic Curves Over Finite Fields

The order of a finite field $\mathbb{F}$ is the number of elements of that field. If the order of a finite field is q, then $q = p^m$, where p is a prime number called the characteristic of the field, and m is a positive integer [7]. In general, ECC protocols use two types of finite fields $\mathbb{F}_q$: $\mathbb{F}_p$ (prime fields) and $\mathbb{F}_{2^m}$ (binary fields).

In prime fields, the elements of $\mathbb{F}_p$ are $\{0, 1, 2, \ldots, p - 1\}$, and the operations are always performed modulo p [8].

In comparison, in finite fields of type $\mathbb{F}_{2^m}$ the elements are represented as bit strings of length m. If $f(x)$ is an irreducible polynomial of degree m with coefficients in $\mathbb{F}_2$, then the field $\mathbb{F}_{2^m}$ can be interpreted as the set of polynomials with coefficients in $\mathbb{F}_2$ of degree less than the degree of $f(x)$ [8].

Depending on the characteristic of the finite field $\mathbb{F}$ that defines the elliptic curve, instead of the general Weierstrass equation two alternate forms, known as the short Weierstrass equations, can be used.

- If $\mathbb{F} = \mathbb{F}_p$, where $p > 3$ is a prime number, the equation defining the (non-supersingular) elliptic curve becomes
$$y^2 = x^3 + ax + b. \tag{1}$$
- If $\mathbb{F} = \mathbb{F}_{2^m}$, where m is an integer number, then the equation of the (non-supersingular) elliptic curve is
$$y^2 + xy = x^3 + ax^2 + b. \tag{2}$$

C. Elliptic Curve Parameters

For elliptic curves defined over prime fields, the set of parameters that must be used in relation to a specific elliptic curve is $\mathcal{P}= (p, a, b, G, n, h)$, where:

- p is the prime number that specifies the field $\mathbb{F}_p$.
- a and b are the elements of $\mathbb{F}_p$ that define the curve E given by (1).
- $G = (x_G, y_G)$ is the generator of a cyclic subgroup of the curve.
- n is the prime number that indicates the order of G.
- h is the cofactor, computed as $\#E/n$.

In comparison, if the curve is defined over a binary field, the set of parameters is $\mathcal{P}= (m, f(x), a, b, G, n, h)$, where:

- m is the positive integer that specifies the field $\mathbb{F}_{2^m}$.
- $f(x)$ is an irreducible polynomial of degree m.
- a and b are the elements of $\mathbb{F}_{2^m}$ that define the curve E given by (2).
- G, n, and h have the same meaning as in the previous case.

DOI: 10.5963/IJCSAI0304002

D. ECC Standards

Some basic applications in security are key exchange, digital signatures, and data encryption. For each of those applications, there are numerous standards, some of which are implemented with ECC.

The best known ECC schemes and protocols are ECDH (Elliptic Curve Diffie Hellman), a key agreement protocol [3, 2]; ECDSA (Elliptic Curve Digital Signature Algorithm), equivalent to the DSA algorithm [9]; and ECIES (Elliptic Curve Integrated Encryption Scheme), the most extended ECC encryption scheme, defined in ANSI X9.63 [10], IEEE 1363a [11], ISO/IEC 18033-2 [12], and SECG SEC 1 [13].

E. Compared Security

The ECDLP is considered to be more difficult to solve than the IFP (Integer Factorization Problem) and the DLP (Discrete Logarithm Problem), which are used in other well-known cryptosystems such as RSA and ElGamal [3, 14]. This is the reason why the key length in ECC is significantly smaller than the key length in other cryptosystems.

Table I provides a comparison between RSA and ECC key lengths, with data taken from [7] and [15]. The security level must be interpreted as the cryptographic strength provided by a symmetric encryption algorithm using a key of n bits. In RSA the key length is the bit length of the modulus, while in ECC it is the number of bits needed to represent the prime number p (in prime fields) or it is the value m (in binary fields).

As it can be observed, the ratio between the key length in RSA and ECC clearly increases, which means that ECC is not only best adapted for devices with limited resources such as the smart cards, but it is also a good option for desktop and server applications where higher security levels are needed.

TABLE I KEY LENGTH COMPARISON MEASURED IN BITS BETWEEN RSA AND ECC

Security level	RSA key length	ECC key length	Ratio
80	1024	160-223	4.6:6.4
112	2048	224-255	8.0:9.1
128	3072	256-283	10.8:12.0
192	7680	384-511	15.0:20.0
256	15360	512-571	26.9:30.0

III. JAVA

The Java programming language was originated in 1990 when a team at Sun Microsystems was working in the design and development of software for small electronic devices.

Despite their efforts, the projects where Java was initially applied to did not succeed. As a second opportunity, Sun decided to use Java in the emerging market of Internet browsing. Then, after the first official version of Java was launched in 1996, its popularity started to increase.

Currently there are more than 9 million Java developers and, according to [16], the figure of Java enabled devices (mainly personal computers, mobile phones, and smart cards) is numbered in the thousands of millions. Between November 2006 and May 2007, Sun Microsystems released most of the Java components under the GNU (*GNU's Not Unix!*) GPL (*General Public License*) model through the OpenJDK project [17], so virtually all the pieces of the Java language are currently free open source software. On January 2010, Oracle Corporation completed the acquisition of Sun Microsystems [18], so at this moment the Java technology is managed by Oracle.

To avoid misunderstandings with the nomenclature, we provide below the complete list of Java versions up to the moment of writing this contribution:

- JDK 1.0 (1996) was the initial version. The name JDK (Java Development Kit) refers to the software development package.
- JDK 1.1 (1997) restructured the GUI (Graphic User Interface) model and introduced the RMI (Remote Method Invocation) concept.
- J2SE 1.2 (1998) modified the name to J2SE (Java 2 Standard Edition) to distinguish this version from J2EE (Java 2 Enterprise Edition). In this release, the Swing graphic API (Application Programming Interface) was integrated in the core of the distribution.
- J2SE 1.3 (2000) included a new virtual machine called Hot Spot.

DOI: 10.5963/IJCSAI0304002

- J2SE 1.4 (2002) was the first release of the Java platform managed by the JCP (Java Community Process) as the project JSR (Java Specification Request) 59. From a programmer point of view, its main novelty was the inclusion of an integrated security model which allowed third-party cryptographic extensions.
- J2SE 5.0 (2004) represented the change from the numbering scheme 1.X to X.0.
- Java SE 6 (2006) modified once more the platform's name, from J2SE to Java SE, and removed the final ".0".
- Java SE 7 (2011) presented changes in the Java language coming from several JSR projects.

In addition to the Standard and Enterprise editions, designed for personal computers and servers, Java can be used also in Android devices and smart cards.

In Java, security is built around two elements: the JCA (Java Cryptography Architecture) and the JCE (Java Cryptography Extension). While the JCA deals with digital signatures and message digests, the JCE manages the key agreement, encryption, key generation and message authentication algorithms.

Algorithm independence is achieved by means of specific cryptographic engines that implement the security functionality. Before J2SE 5.0, the JCA/JCE architecture did not include specific classes for ECC. Developers willing to use ECC algorithms were forced to acquire software from third parties that, in most cases, was not compatible with software from other vendors. In J2SE 5.0 and Java SE 6, some classes and interfaces were included in order to facilitate a standard ECC support. However, it was still necessary to use third party engines in order to access all the power that ECC can provide to Java applications.

With the release of Java SE 7 the situation changed again, since in that version Oracle included a new cryptographic engine called SunEC [19]. Using key lengths from 112 to 571 bits, SunEC allows to generate key pairs, complete key agreement exchanges, and produce digital signatures.

More specifically, the SunEC provider implements the ECDH key agreement protocol and the ECDSA digital signature procedure. Whilst the SunEC implementation of the ECDH protocol produces a shared value which is the first coordinate of the elliptic curve point computed as the product as one user's public key and the other user's private key, the ECDSA implementation allows to use the following hash functions: SHA-1, SHA-256, SHA-384, and SHA-512.

IV.ELLIPTIC CURVES IN THE SunEC PROVIDER

Tables II and III show the Java identifiers of the elliptic curves implemented in SunEC over prime and binary fields, respectively. These curves were defined in the first versions of SECG SEC 2 [20] and ANSI X9.62 [21], and in the second version of NIST FIPS 186 [22].

TABLE II ELLIPTIC CURVES OVER $\mathbb{F}_p$ IN SunEC

SECG SEC 2	ANSI X9.62	NIST FIPS 186-2
secp112r1		
secp112r2		
secp128r1		
secp128r2		
secp160k1		
secp160r1		
secp160r2		
secp192k1		
secp192r1	X9.62 prime192v1	NIST P-192
	X9.62 prime192v2	
	X9.62 prime192v3	
secp224k1		
secp224r1		NIST P-224
	X9.62 prime239v1	
	X9.62 prime239v2	
	X9.62 prime239v3	
secp256k1		
secp256r1	X9.62 prime256v1	NIST P-256
secp384r1		NIST P-384
secp521r1		NIST P-521

DOI: 10.5963/IJCSAI0304002

TABLE III ELLIPTIC CURVES OVER $\mathbb{F}_{2^m}$ IN SunEC

SECG SEC 2	ANSI X9.62	NIST FIPS 186-2
sect113r1		
sect113r2		
sect131r1		
sect131r2		
sect163k1		NIST K-163
sect163r1		
sect163r2		NIST B-163
	X9.62 c2tnb191v1	
	X9.62 c2tnb191v2	
	X9.62 c2tnb191v3	
sect193r1		
sect193r2		
sect233k1		NIST K-233
sect233r1		NIST B-233
sect239k1		
	X9.62 c2tnb239v1	
	X9.62 c2tnb239v2	
	X9.62 c2tnb239v3	
sect283k1		NIST K-283
sect283r1		NIST B-283
	X9.62 c2tnb359v1	
sect409k1		NIST K-409
sect409r1		NIST B-409
	X9.62 c2tnb431r1	
sect571k1		NIST K-571
sect571r1		NIST B-571

The identifiers, which are part of the class `sun.security.ec.SunECEntries`, have been distributed along Tables II and III so that the identifiers located in the same row represent the same curve. Therefore, if a cell is empty, that means that the elliptic curve referred to by that row is not defined in the standard in question.

The meaning of the elements used as part of the identifiers can be found in their parent specifications. As a summary, the identifiers are composed of a string that informs about the original specification and the type of elliptic curve (e.g. secp, sect, X9.62 prime, X9.62 c2tnb, NIST P, NIST B, and NIST K), followed by the size in bits of the finite field and, depending on the standard, a string that allows to differentiate between curves where the rest of the parameters used by the same specification have the same value.

It is important to note that the curves published in the 2005 version of the X9.62 standard [23] do not match those included in the 1998 edition [21], since as described in [23], some were eliminated as they ceased to be considered secure. Additionally, the 2005 edition changed the identifiers of the remaining curves (e.g. the ansix9p192r1 curve of the 2005 edition is the prime192v1 curve of the 1998 edition).

Similarly, the second version of SECG SEC 2 [24] removed some of the curves defined in the first version [20] due to security reasons. SECG is planning to revise the specification every five years in order to ensure the integrity of the curves against new attacks, which means that the next version will probably be published in 2015.

Given that [24] is the most recent standard dealing with this topic, and that the curves of [9] are a subset of those defined in [24], we recommend to use the following curves when developing ECC applications with Java.

- Curves over $\mathbb{F}_p$
  - 192 bits: secp192k1 and secp192r1.
  - 224 bits: secp224k1 and secp224r1.
  - 256 bits: secp256k1 and secp256r1.
  - 384 bits: secp384r1.
  - 521 bits: secp521r1.
- Curves over $\mathbb{F}_{2^m}$
  - 163 bits: sect163k1, sect163r1, and sect163r2.
  - 233 bits: sect233k1 and sect233r1.
  - 239 bits: sect239k1.
  - 283 bits: sect283k1 and sect283r1.
  - 409 bits: sect409k1 and sect409r1.
  - 571 bits: sect571k1 and sect571r1.

DOI: 10.5963/IJCSAI0304002

V. CODE EXAMPLES

In this section we will show three examples of Java applications using the SunEC cryptographic engine.

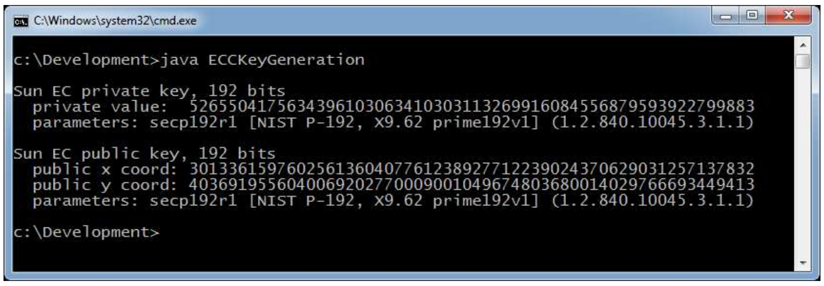
A. Key Pair Generation

The code shown in Listing A. generates a key pair using the secp192r1 curve. Additionally, it displays on the screen the content of the public and private keys.

```java
import java.security.*;
import java.security.spec.*;

public class ECCKeyGeneration {
  public static void main(String[] args) throws Exception {
    KeyPairGenerator kpg;
    kpg = KeyPairGenerator.getInstance("EC","SunEC");
    ECGenParameterSpec ecsp;
    ecsp = new ECGenParameterSpec("secp192r1");
    kpg.initialize(ecsp);

    KeyPair kp = kpg.genKeyPair();
    PrivateKey privKey = kp.getPrivate();
    PublicKey pubKey = kp.getPublic();

    System.out.println(privKey.toString());
    System.out.println(pubKey.toString());
  }
}
```

Listing 1 ECC key generation example code

Figure 1 shows the output produced when executing the program ECCKeyGeneration.



```
c:\Development>java ECCKeyGeneration

Sun EC private key, 192 bits
  private value:  5265504175634396103063410303113269916084556879593922799883
  parameters: secp192r1 [NIST P-192, X9.62 prime192v1] (1.2.840.10045.3.1.1)

Sun EC public key, 192 bits
  public x coord: 3013361597602561360407761238927712239024370629031257137832
  public y coord: 4036919556040069202770009001049674803680014029766693449413
  parameters: secp192r1 [NIST P-192, X9.62 prime192v1] (1.2.840.10045.3.1.1)

c:\Development>
```

Fig. 1 ECC key generation example output

B. Key Agreement

The code displayed in Listing B. allows two users, U and V, to complete the ECDH key agreement protocol. The private keys of those users are u and v, respectively, while their public keys are denoted as $U = u \cdot G$ and $V = v \cdot G$, respectively. The SunEC implementation of the ECDH protocol produces a shared value which is the first coordinate of the elliptic curve point computed as $u \cdot V = u \cdot v \cdot G = v \cdot u \cdot G = v \cdot U$.

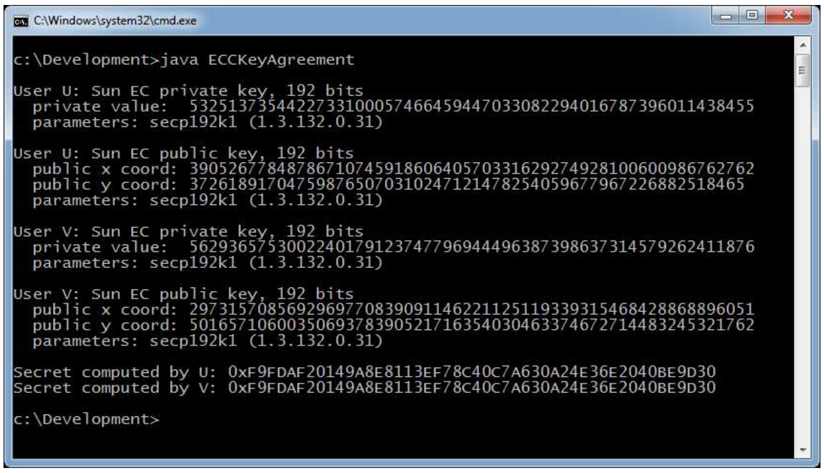
```java
import java.math.BigInteger;
import java.security.*;
import java.security.spec.*;
import javax.crypto.KeyAgreement;

public class ECCKeyAgreement
{
  public static void main(String[] args) throws Exception
  {
    KeyPairGenerator kpg;
    kpg = KeyPairGenerator.getInstance("EC","SunEC");
    ECGenParameterSpec ecsp;

    ecsp = new ECGenParameterSpec("secp192k1");
    kpg.initialize(ecsp);

    KeyPair kpU = kpg.genKeyPair();
```

DOI: 10.5963/IJCSAI0304002

```java
    PrivateKey privKeyU = kpU.getPrivate();
    PublicKey pubKeyU = kpU.getPublic();
    System.out.println("User U: " + privKeyU.toString());
    System.out.println("User U: " + pubKeyU.toString());

    KeyPair kpV = kpg.genKeyPair();
    PrivateKey privKeyV = kpV.getPrivate();
    PublicKey pubKeyV = kpV.getPublic();
    System.out.println("User V: " + privKeyV.toString());
    System.out.println("User V: " + pubKeyV.toString());

    KeyAgreement ecdhU = KeyAgreement.getInstance("ECDH");
    ecdhU.init(privKeyU);
    ecdhU.doPhase(pubKeyV,true);

    KeyAgreement ecdhV = KeyAgreement.getInstance("ECDH");
    ecdhV.init(privKeyV);
    ecdhV.doPhase(pubKeyU,true);

    System.out.println("Secret computed by U: 0x" + (new BigInteger(1, ecdhU.generateSecret())
      .toString(16)).toUpperCase());
    System.out.println("Secret computed by V: 0x" + (new BigInteger(1, ecdhV.generateSecret())
      .toString(16)).toUpperCase());
  }
}
```

Listing 2 ECC key agreement example code

The output of the application called ECCKeyAgreement is displayed in Figure 2.



```
c:\Development>java ECCKeyAgreement

User U: Sun EC private key, 192 bits
  private value:  5325137354422733100057466459447033082294016787396011438455
  parameters: secp192k1 (1.3.132.0.31)

User U: Sun EC public key, 192 bits
  public x coord: 3905267784878671074591860640570331629274928100600986762762
  public y coord: 372618917047598765070310247121478254059677967226882518465
  parameters: secp192k1 (1.3.132.0.31)

User V: Sun EC private key, 192 bits
  private value:  5629365753002240179123747796944496387398637314579262411876
  parameters: secp192k1 (1.3.132.0.31)

User V: Sun EC public key, 192 bits
  public x coord: 2973157085692969770839091146221125119339315468428868896051
  public y coord: 5016571060035069378390521716354030463374672714483245321762
  parameters: secp192k1 (1.3.132.0.31)

Secret computed by U: 0xF9FDAF20149A8E8113EF78C40C7A630A24E36E2040BE9D30
Secret computed by V: 0xF9FDAF20149A8E8113EF78C40C7A630A24E36E2040BE9D30

c:\Development>
```

Fig. 2 ECC key agreement example output

C. Digital Signature

The code shown in Listing C. allows to generate the digital signature of the text “In teaching others we teach ourselves” using the ECDSA scheme. In addition to that, the application validates the resulting signature, simulating the action of the user who receives both the text and the signature.

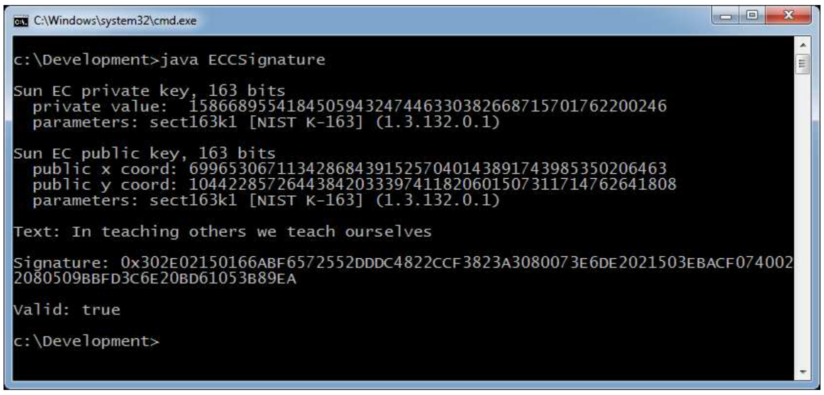
```java
import java.math.BigInteger;
import java.security.*;
import java.security.spec.*;

public class ECCSignature
{
  public static void main(String[] args) throws Exception
  {
    KeyPairGenerator kpg;
    kpg = KeyPairGenerator.getInstance("EC","SunEC");

    ECGenParameterSpec ecsp;
    ecsp = new ECGenParameterSpec("sect163k1");
    kpg.initialize(ecsp);

    KeyPair kp = kpg.genKeyPair();
    PrivateKey privKey = kp.getPrivate();
```

DOI: 10.5963/IJCSAI0304002

```java
    PublicKey pubKey = kp.getPublic();
    System.out.println(privKey.toString());
    System.out.println(pubKey.toString());

    Signature ecdsa;
    ecdsa = Signature.getInstance("SHA1withECDSA","SunEC");
    ecdsa.initSign(privKey);

    String text = "In teaching others we teach ourselves";
    System.out.println("Text: " + text);
    byte[] baText = text.getBytes("UTF-8");

    ecdsa.update(baText);
    byte[] baSignature = ecdsa.sign();
    System.out.println("Signature: 0x" + (new BigInteger(1, baSignature).toString(16)).toUpperCase());

    Signature signature;
    signature = Signature.getInstance("SHA1withECDSA","SunEC");
    signature.initVerify(pubKey);
    signature.update(baText);
    boolean result = signature.verify(baSignature);
    System.out.println("Valid: " + result);
  }
}
```

Listing 3 ECC digital signature example code

Figure 3 shows the output produced when running the application presented in Listing C..


C:\Windows\system32\cmd.exe

```
c:\Development>java ECCSignature

Sun EC private key, 163 bits
  private value:  1586689554184505943247446330382668715701762200246
  parameters: sect163k1 [NIST K-163] (1.3.132.0.1)

Sun EC public key, 163 bits
  public x coord: 6996530671134286843915257040143891743985350206463
  public y coord: 10442285726443842033397411820601507311714762641808
  parameters: sect163k1 [NIST K-163] (1.3.132.0.1)

Text: In teaching others we teach ourselves

Signature: 0x302E02150166ABF6572552DDDC4822CCF3823A308007 3E6DE2021503EBACF074002
2080509BBFD3C6E20BD61053B89EA

Valid: true

c:\Development>
```

Fig. 3 ECC digital signature example output

VI. CONCLUSIONS

During the past years, Java has been one of the technologies with a fastest growth. Since its version Java SE 5.0, it is possible to use ECC implementations adapted to the JCA/JCE framework. With the new SunEC cryptographic provider included by Oracle in the Java SE 7 distribution, programmers are able to develop ECC applications easily without the need to manage third party libraries.

However, there is scarce information about this cryptographic provider. For example, the list of supported elliptic curves included in this document has been taken from the source code of one of the SunEC classes, as they are not documented. Another limiting factor is the lack of code samples that use the SunEC provider.

In this contribution, we have tried to facilitate the usage of ECC in Java by analysing the capabilities of the SunEC provider and offering three complete examples dealing with key generation, key exchange, and digital signatures. With that information, we believe that any interested reader can start developing powerful ECC applications very quickly.

Although the issuing of the SunEC provider has facilitated a lot the adoption of ECC by Java programmers, it must be noted that there is still room for improvement. In this sense, the actual elliptic curves implemented by the provider should be updated in accordance with the latest standards. In addition to that, it would be highly desirable if Oracle implemented ECIES, the most widely used ECC encryption procedure. With the implementation of this encryption scheme, developers would have access to the three security primitives typically needed in security projects: key exchanges, digital signatures, and data encryption.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, 1976.

[2] V. S. Miller. Use of elliptic curves in cryptography. *Lecture Notes in Computer Science*, 218:417–426, 1986.

[3] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

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

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

[6] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, Boston, MA, USA, 1993.

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

[8] E. Bach and J. Shallit. *Algorithmic Number Theory. Volume I: Efficient Algorithms*. The MIT Press, Cambridge, MA, USA, 1996.

[9] National Institute of Standards and Technology. *Digital Signature Standard (DSS)*. NIST FIPS 186-3, 2009.

[10] American National Standards Institute. *Public Key Cryptography for the Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography*. ANSI X9.63, 2001.

[11] Institute of Electrical and Electronics Engineers. *Standard Specifications for Public Key Cryptography - Amendment 1: Additional Techniques*. IEEE 1363a, 2004.

[12] International Organization for Standardization / International Electrotechnical Commission. *Information Technology – Security Techniques – Encryption Algorithms – Part 2: Asymmetric Ciphers*. ISO/IEC 18033-2, 2006.

[13] Standards for Efficient Cryptography Group. *Recommended Elliptic Curve Domain Parameters*. SECG SEC 1 version 2.0, 2009.

[14] Bundesamt für Sicherheit in der Informationstechnik. *Elliptic Curve Cryptography*, 2009. `http://www.bsi.de/literat/tr/tr03111/BSI-TR-03111.pdf`.

[15] National Institute of Standards and Technology. *Recommendation for Key Management. Part 1: General*. NIST SP 800-57, 2007.

[16] Oracle Corp. *Java Technology*, 2013. `http://java.com/en/about`.

[17] Oracle Corp. *OpenJDK*, 2010. `http://openjdk.java.net`.

[18] Oracle Corp. *Oracle Completes Acquisition of Sun*, 2010. `http://www.oracle.com/us/corporate/press/044428`.

[19] Oracle Corp. *The SunEC Provider*, 2013. `http://docs.oracle.com/javase/7/docs/technotes/guides/security/SunProviders.html#SunEC`.

[20] Standards for Efficient Cryptography Group. *Recommended Elliptic Curve Domain Parameters*. SECG SEC 2 version 1.0, 2000.

[21] American National Standards Institute. *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*. ANSI X9.62, 1998.

[22] National Institute of Standards and Technology. *Digital Signature Standard (DSS)*. NIST FIPS 186-2, 2000.

[23] American National Standards Institute. *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*. ANSI X9.62, 2005.

[24] Standards for Efficient Cryptography Group. *Recommended Elliptic Curve Domain Parameters*. SECG SEC 2 version 2.0, 2010.

DOI: 10.5963/IJCSAI0304002