6 Mật mã trong Java Card

6.1 Tính toán tóm lược thông điệp (hàm băm)

Tóm lược thông điệp là một hàm băm duy nhất và đáng tin cậy của thông điệp cho phép nhận biết rằng thông điệp nhận được cũng chính là thông điệp đã được gửi.

Để tính toán tóm lược thông điệp cần thực hiện theo các bước sau:

Bước 1: Tạo đối tượng tóm lược thông điệp. Để làm như vậy, chúng ta sẽ gọi phương thức xuất xưởng `getInstance` của lớp `MessageDigest.`

```
public static final MessageDigest
getInstance(byte algorithm, boolean externalAccess)
```

Tham số `algorithm` chỉ định một thuật toán mong muốn bằng cách sử dụng một trong các tham số chọn thuật toán.

Tham số thứ hai, `externalAccess`, nếu là `true` chỉ ra rằng cá thể sẽ được chia sẻ giữa nhiều phiên bản applet và một Applet được định nghĩa trong một gói khác có thể truy cập vào đối tượng tóm lược thông điệp thông qua một phương thức giao diện có thể chia sẻ. Nếu tham số của `externalAccess` được đặt thành `false` (có nghĩa là đối tượng tóm lược thông điệp được trả lại không dành cho truy cập bên ngoài), đối tượng tóm lược thông điệp chỉ có thể được truy cập bởi các Applet trong ngữ cảnh sở hữu của nó và khi một trong các Applet này là Applet hiện được chọn.

Bước 2: Cung cấp dữ liệu đầu vào và tính toán tóm lược thông điệp.

Giả sử rằng chúng ta có ba mảng byte `m1`, `m2` và `m3`, tạo thành tổng đầu vào có tóm lược thông điệp mà chúng ta muốn tính toán. Chúng ta nên tính toán thông qua các hàm sau:

```
// cung cấp toàn bộ dữ liệu trong mảng byte m1 cho tóm lược thông
điệp
```

```
sha.update(m1, (short)0, (short)(m1.length));

// cung cấp thêm 8 bytes trong mảng byte m2 bắt đầu từ offset 0
sha.update(m2, (short)0, (short)8);

// gửi toàn bộ dữ liệu theo byte byte m3 như đợt cuối cùng và lưu
trữ giá trị băm trong mảng byte digest bắt đầu từ offset 0
sha.doFinal(m3, (short)0, (short)(m3.length), digest, (short)0);
```

Phương thức `update` cho phép chúng ta cung cấp dữ liệu đầu vào một cách linh hoạt để tạo ra bản tóm lược. Nhưng khi chúng ta đến lô dữ liệu cuối cùng, chúng ta nên gọi phương thức `doFinal`. Điều này thông báo cho tóm lược thông điệp rằng nó là phần cuối của dữ liệu đầu vào và hệ thống sẽ thực hiện các hoạt động cuối cùng, chẳng hạn như phần đệm. Phương thức `doFinal` hoàn thành và trả về phép tính băm trong mảng đầu ra được chỉ định. Trong ví dụ này, nó ghi giá trị băm vào đầu mảng byte `digest`. Trong phương thức `doFinal`, dữ liệu đầu ra có thể trùng với dữ liệu đầu vào, do đó chúng ta có thể sử dụng lại mảng byte `m3` cho đầu ra.

```
sha.doFinal(m3, (short)0, (short)(m3.length), m3, (short)0);
```

Sau khi phương thức `doFinal` được gọi, đối tượng tóm lược thông điệp sẽ tự động được đặt lại. Do đó, nó có thể được sử dụng để tính toán các giá trị mới.

Nếu tất cả các dữ liệu đầu vào đặt vào trong một mảng byte, chúng ta nên bỏ qua phương thức `update` và chỉ gọi phương thức `doFinal`. Vì phương thức `update` sử dụng lưu trữ tạm thời cho kết quả băm trung gian, nên chỉ được sử dụng nếu tất cả dữ liệu đầu vào cần thiết cho hàm băm có thể được chứa trong một mảng byte. Vì phương thức `update` có thể dẫn đến tiêu thụ tài nguyên bổ sung hoặc hiệu suất chậm, chỉ sử dụng phương thức `doFinal` bất cứ khi nào có thể.

Tại bất kỳ thời điểm nào trong quá trình tính toán băm và trước khi phương thức `doFinal` được gọi, chúng ta có thể gọi phương thức `reset` để bỏ qua đầu vào trước đó và đặt lại trạng thái ban đầu cho một tính toán mới.

Sau đây chúng ta xem xét một ví dụ về cách tính toán tóm lược thông điệp:

```
/*Tom luoc thong diep*/
package bai6_Crypto;

import javacard.framework.*;
import javacard.security.*;
import javacardx.crypto.*;

public class Applet1 extends Applet
{
     private MessageDigest sha;
     private byte[] m1, m2, m3;

     private Applet1()
     {
          sha = MessageDigest.getInstance(MessageDigest.ALG_SHA,
false);
          m1 = new byte[] {0x01, 0x02};
          m2 = new byte[] {0x03, 0x04, 0x05};
          m3 =  new byte[] {0x06, 0x07};
     }

     public static void install(byte[] bArray, short bOffset,
byte bLength)
     {
          new Applet1().register(bArray, (short) (bOffset + 1),
bArray[bOffset]);
     }

     public void process(APDU apdu)
     {
```

```
            if (selectingApplet())
            {
                 return;
            }

            byte[] buf = apdu.getBuffer();
            apdu.setIncomingAndReceive();
            switch (buf[ISO7816.OFFSET_INS])
            {
            case (byte)0x00:
                 sha.update(m1, (short)0, (short)(m1.length));
                 sha.update(m2, (short)0, (short)(m2.length));
                 short ret = sha.doFinal(m3, (short)0,
(short)(m3.length), buf, (short)0);
                 apdu.setOutgoingAndSend((short)0, ret);
                 break;
            default:

     ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
            }
     }

}
```

Kết quả thực thi Applet trên như sau:

```
>> /select 11223344550601
>> 00 A4 04 00 07 11 22 33 44 55 06 01 00
<< 90 00

>> /send 00000102
>> 00 00 01 02
<< 8C 1F 28 FC 2F 48 C2 71 D6 C4 98 F0 F2 49 CD DE 36 5C 54 C5
90 00
```

Từ phiên bản Java Card 2.2.2, lớp `MesseageDigest` được bổ sung thêm lớp con `InitializedMessageDigest` với các phương thức khởi tạo bổ sung, ví dụ như phương thức `setInitialDigest` cho phép khởi tạo giá trị băm ban đầu cho hàm băm. Khi đó để tạo đối tượng `InitializedMessageDigest` chúng ta gọi phương thức

```
public static final InitializedMessageDigest
getInitializedMessageDigestInstance(byte algorithm, boolean
externalAccess)
```

Các tham số `algorithm` và `externalAccess` được sử dụng tương tự như trong phương thức `getInstance.`

Khi đó ví dụ trên có thể viết lại như sau:

```
package bai6_Crypto;
import javacard.framework.*;
import javacard.security.*;

public class Applet2 extends Applet
{
     private InitializedMessageDigest sha;
     private byte[] m1, m2, m3;
     private Applet2()
     {
          sha =
MessageDigest.getInitializedMessageDigestInstance(MessageDigest.A
LG_SHA, false);
          m1 = new byte[] {0x01, 0x02};
          m2 = new byte[] {0x03, 0x04, 0x05};
          m3 =  new byte[] {0x06, 0x07};
     }
```

```
     public static void install(byte[] bArray, short bOffset,
byte bLength)
     {
          new Applet2().register(bArray, (short) (bOffset + 1),
bArray[bOffset]);
     }

     public void process(APDU apdu)
     {
          if (selectingApplet())
          {
               return;
          }

          byte[] buf = apdu.getBuffer();
          apdu.setIncomingAndReceive();
          switch (buf[ISO7816.OFFSET_INS])
          {
          case (byte)0x00:
               sha.update(m1, (short)0, (short)(m1.length));
               sha.update(m2, (short)0, (short)(m2.length));
               short ret = sha.doFinal(m3, (short)0,
(short)(m3.length), buf, (short)0);
               apdu.setOutgoingAndSend((short)0, ret);
               break;
          default:

     ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
          }
     }
}
```

6.2 Ký và kiểm tra chữ ký

Chữ ký (chữ ký số) cung cấp hai dịch vụ bảo mật: xác thực và toàn vẹn. Lớp `javacard.security.Signature` được thiết kế để được sử dụng theo kiểu tương tự như lớp `MessageDigest`. Để tạo một đối tượng `Signature`, chúng ta gọi phương thức nhà máy `getInstance` của lớp `Signature`:

```
Signature rsaSig;
rsaSig = Signature.getInstance(Signature.ALG_RSA_SHA_PCKS1,
false);
```

Để chỉ định một thuật toán, người ta có thể sử dụng một trong các tham số lựa chọn thuật toán được định nghĩa trong lớp `Signature`. Một tham số lựa chọn thuật toán chỉ định cả tóm lược thông điệp và thuật toán mã hóa. Lớp `Signature` hỗ trợ một tập hợp rộng lớn các thuật toán chữ ký khả dĩ. Tham số thứ hai, `externalAccess`, cho biết liệu đối tượng `Signature` được trả về có thể được truy cập bên ngoài bởi một bối cảnh khác với bối cảnh sở hữu của nó hay không.

Để có thể ký, đầu tiên chúng ta cần có khóa. Khóa mật mã là một giá trị bí mật được cung cấp cho thuật toán mật mã để mã hóa và giải mã dữ liệu. API mã hóa Java Card xác định một bộ giao diện mở rộng để triển khai cả khóa đối xứng và khóa bất đối xứng. Để xây dựng một khóa, chúng ta sẽ gọi phương thức `buildKey` của lớp `KeyBuilder`:

```
public static Key buildKey
(byte keyType, short keyLength, boolean keyEncryption);
```

Lớp `KeyBuilder` định nghĩa một tập các tham số lựa chọn mà chúng ta có thể chọn để chọn loại khóa và độ dài khóa. Ví dụ: để tạo khóa riêng RSA có độ dài 128 byte (128 * 8 = 1024 bit), chúng ta gọi phương thức `buildKey` như sau:

```
Key rsaPrivKey;
rsaPrivKey = KeyBuilder.buildKey(KeyBuilder.TYPE_RSA_PRIVATE,
KeyBuilder.LENGTH_RSA_1024, false);
```

Phương thức buildKey trả về một đối tượng của loại giao diện Key. Kiểu thực của đối tượng là một lớp thực thi giao diện khóa của loại khóa được yêu cầu. Trong ví dụ này, lớp triển khai khóa sẽ triển khai giao diện RSAPrivateKey. Để có thể gọi các phương thức trong giao diện RSAPrivateKey, chúng ta chuyển đối tượng chính thành RSAPrivateKey:

```
RSAPrivateKey rsaPrivKey;
rsaPrivKey = (RSAPrivateKey)KeyBuilder.buildKey
(KeyBuilder.TYPE_RSA_PRIVATE,
KeyBuilder.LENGTH_RSA_1024, false);
```

Tương tự để tạo khóa công khai chúng ta gọi phương thức buildKey như sau:

```
RSAPublicKey rsaPubKey;
rsaPubKey = (RSAPublicKey)KeyBuilder.buildKey
(KeyBuilder.TYPE_RSA_PUBLIC,
KeyBuilder.LENGTH_RSA_1024, false);
```

Phương thức buildKey trả về một đối tượng khóa với loại khóa được yêu cầu, nhưng đối tượng khóa không được khởi tạo. Để khởi tạo khóa, chúng ta có thể sử dụng phương thức genKeyPair trong lớp KeyPair để tạo cặp khóa RSA như sau:

```
KeyPair keyPair = new KeyPair(KeyPair.ALG_RSA,
KeyBuilder.LENGTH_RSA_1024);
keyPair.genKeyPair();
```

Khi đó khóa bí mật và khóa công khai sẽ được khởi tạo từ cặp khóa vừa tạo:

```
rsaPrivKey = (RSAPrivateKey)keyPair.getPrivate();
rsaPubKey = (RSAPublicKey)keyPair.getPublic();
```

Ngoài ra, khóa bí mật và khóa công khai RSA cũng có thể được khởi tạo thông qua các phương thức `setModulus` và `setExponent` trong giao diện `RSAPrivateKey`:

```
public void setExponent(byte[] buffer, short offset, short
length)
public void setModulus(byte[] buffer, short offset, short length)
```

Để có thể ký và xác minh chữ ký, chúng ta cần chúng ta cần khởi tạo đối tượng `Signature`. Để làm như vậy, chúng ta gọi một trong hai phương thức `init`.

```
public void init (Key theKey, byte theMode);
public void init (Key theKey, byte theMode,
byte[] bArray, short bOff, short bLen);
```

Trong thuật toán bất đối xứng, ký và xác minh không sử dụng cùng một khóa. Do đó, chúng ta cần xác định cách sử dụng khóa trong tham số thứ hai `theMode`. Có hai chế độ, như được định nghĩa trong lớp `Signature.`

- MODE_SIGN - biểu thị chế độ ký
- MODE_VERIFY - biểu thị chế độ xác minh

Phương thức `init` thứ hai cũng cho phép chúng ta chỉ định dữ liệu khởi tạo thuật toán trong mảng byte `bArray`. Một ví dụ về dữ liệu khởi tạo là bộ khởi tạo (IV) cho DES và triple DES ở chế độ CBC.

Để tính toán chữ ký, trước tiên chúng ta cung cấp dữ liệu bằng phương thức `update`. Khi chúng ta cung cấp lô dữ liệu cuối cùng, hãy gọi phương thức `sign`. Dưới đây là ví dụ tính toán chữ ký từ dữ liệu trong các mảng `s1`, `s2` và `s3.`

```
rsaSig.update(s1, (short)0, (short)(s1.length));
rsaSig.update(s2, (short)0, (short)(s2.length));
```

```
    rsaSig.sign(s3, (short)0, (short)(s3.length), sig_buffer,
(short)0);
```

Để xác minh chữ ký, trước tiên chúng ta cung cấp cùng một dữ liệu đầu vào bằng phương thức `update`. Khi chúng ta cung cấp lô dữ liệu cuối cùng, hãy gọi phương thức `verify`. Phương thức `verify` xác minh chữ ký được tính toán từ dữ liệu đầu vào so với dữ liệu được cung cấp. Nếu cả hai khớp, nó trả về `true`. Ví dụ sau đây cho thấy cách xác minh chữ ký được tính trong ví dụ trước.

```
    rsaSig.update(s1, (short)0, (short)(s1.length));
    rsaSig.update(s2, (short)0, (short)(s2.length));
    boolean ret = rsaSig.verify(s3, (short)0,
(short)(s3.length), sig_buffer, (short)0, sigLen);
```

Sau đây là ví dụ một Applet ký và kiểm tra chữ ký:

```
package bai6_Crypto;

import javacard.framework.*;
import javacardx.crypto.*;
import javacard.security.*;
import javacard.security.KeyBuilder;

public class Applet3_RSA extends Applet
{
    private static final byte INS_SIGN  = (byte)0x00;
    private static final byte INS_VERIFY = (byte)0x01;

    private RSAPrivateKey rsaPrivKey;
    private RSAPublicKey rsaPubKey;
    private Signature rsaSig;

    private byte[] s1, s2, s3, sig_buffer;
```

```
    private short sigLen;

    private Applet3_RSA()
    {
        s1 = new byte[]{0x01, 0x02, 0x03};
        s2 = new byte[]{0x04, 0x05};
        s3 = new byte[]{0x06, 0x07, 0x08};
        sigLen = (short)(KeyBuilder.LENGTH_RSA_1024/8);
        sig_buffer = new byte[sigLen];

        rsaSig =
Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1,false);
        rsaPrivKey =
(RSAPrivateKey)KeyBuilder.buildKey(KeyBuilder.TYPE_RSA_PRIVATE,
(short)(8*sigLen),false);
        rsaPubKey =
(RSAPublicKey)KeyBuilder.buildKey(KeyBuilder.TYPE_RSA_PUBLIC,
(short)(8*sigLen), false);

        KeyPair keyPair = new KeyPair(KeyPair.ALG_RSA,
(short)(8*sigLen));
        keyPair.genKeyPair();
        rsaPrivKey = (RSAPrivateKey)keyPair.getPrivate();
        rsaPubKey = (RSAPublicKey)keyPair.getPublic();
    }
    public static void install(byte[] bArray, short bOffset,
byte bLength)
    {
        new Applet3_RSA().register(bArray, (short) (bOffset +
1), bArray[bOffset]);
    }
```

```
	public void process(APDU apdu)
	{
		if (selectingApplet())
		{
			return;
		}

		byte[] buf = apdu.getBuffer();
		apdu.setIncomingAndReceive();
		switch (buf[ISO7816.OFFSET_INS])
		{
		case INS_SIGN:
			rsaSign(apdu);
			break;
		case INS_VERIFY:
			rsaVerify(apdu);
			break;
		default:

	ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
		}
	}

	private void rsaSign(APDU apdu)
	{
		rsaSig.init(rsaPrivKey, Signature.MODE_SIGN);
		rsaSig.update(s1, (short)0, (short)(s1.length));
		rsaSig.update(s2, (short)0, (short)(s2.length));
		rsaSig.sign(s3, (short)0, (short)(s3.length),
sig_buffer, (short)0);
		apdu.setOutgoing();
		apdu.setOutgoingLength(sigLen);
```

```
            apdu.sendBytesLong(sig_buffer, (short)0, sigLen);
      }
      private void rsaVerify(APDU apdu)
      {
            byte [] buf = apdu.getBuffer();
            rsaSig.init(rsaPubKey, Signature.MODE_VERIFY);
            rsaSig.update(s1, (short)0, (short)(s1.length));
            rsaSig.update(s2, (short)0, (short)(s2.length));
            boolean ret = rsaSig.verify(s3, (short)0,
(short)(s3.length), sig_buffer, (short)0, sigLen);

            buf[(short)0] = ret ? (byte)1 : (byte)0;
            apdu.setOutgoingAndSend((short)0, (short)1);
      }
}
```

Kết quả thực thi Applet trên như sau:

```
>> /select 11223344550603
>> 00 A4 04 00 07 11 22 33 44 55 06 03 00
<< 90 00

>> /send 00000102
>> 00 00 01 02
<< 40 DF 50 77 CD B2 88 AE CD 71 B0 C6 17 43 21 74 50 A9 9D 94
6A 1D 11 B0 34 1E 18 D2 E5 48 74 1D 0F 66 76 50 3E BE 76 A9 C3
7A 1A EB 1D 34 1C 22 DF AA 97 5C A9 C3 70 AF 8B 4B 1C 9E EB 5D
99 BB 85 1C B4 55 46 C3 E2 BD 37 3B E8 4E 1E 80 26 05 4C 40 D5
FF AE 0F AB EF 63 05 87 15 38 ED 12 70 2B F5 EF 33 50 23 95 A1
D9 06 D9 F1 33 79 C1 41 B7 A9 C1 1F B3 6F 83 43 49 39 2C F6 89
6A 20 47 90 00

>> /send 00010102
>> 00 01 01 02
<< 01 90 00
```

Lưu ý: Tương tự như các phương thức trong lớp `MessageDigest`, chỉ nên sử dụng phương thức `update` nếu tất cả dữ liệu đầu vào không vừa trong một mảng

byte. Sau khi phương thức `sign` hoặc `verify` được gọi đến, đối tượng `Signature` được đặt lại về trạng thái sau khi được khởi tạo trước đó thông qua phương thức `init`. Cũng trong phương thức `sign`, dữ liệu đầu vào và dữ liệu chữ ký đầu ra có thể sử dụng cùng một mảng và dữ liệu có thể trùng nhau.

6.3 Mã hóa và giải mã

Mã hóa là một công cụ để bảo vệ sự riêng tư dữ liệu. Mã hóa làm xáo trộn dữ liệu tồn tại trong bản rõ và biến chúng thành bản mã. Giải mã phục hồi bản rõ gốc từ bản mã.

Lớp `javacardx.crypto.Cipher` cung cấp cả dịch vụ mã hóa và giải mã với các thuật toán đối xứng hoặc bất đối xứng. Tương tự như các lớp `MessageDigest` và `Signature`, chúng ta gọi phương thức nhà máy `getInstance` và cung cấp hai tham số để tạo một đối tượng `Cipher`. Tham số đầu tiên chỉ định một thuật toán mật mã. Tham số thứ hai, `externalAccess`, cho biết liệu `Cipher`, đối tượng được trả về có thể được truy cập bên ngoài bởi một bối cảnh khác với bối cảnh sở hữu của nó hay không (xem lớp `keyBuilder`).

Sau đó, chúng ta khởi tạo đối tượng `Cipher` bằng một khóa thích hợp và chỉ định xem khóa đó được sử dụng để mã hóa hay để giải mã. Để làm như vậy, chúng ta gọi một trong hai phương thức `init`:

```
public void init (Key theKey, byte theMode);
public void init (Key theKey, byte theMode,
          byte[] bArray, short bOff, short bLen);
```

Ví dụ sau đây tạo một đối tượng `Cipher` với thuật toán AES 128 ở chế độ ECB. Dữ liệu đầu vào sẽ không được đệm. Đối tượng `Cipher` được khởi tạo bằng khóa `aesKey` để mã hóa:

```
Cipher cipher;
```

```
cipher = Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_ECB_NOPAD,
false);
cipher.init(aesKey, Cipher.MODE_ENCRYPT);
```

Chúng ta cũng có thể khởi tạo đối tượng `Cipher` để giải mã bằng cách chọn tham số lựa chọn là `MODE_DECRYPT`, chỉ định khóa được cung cấp trong phương thức `init` là để giải mã. Tiếp theo, để mã hóa dữ liệu, hãy sử dụng phương thức `update` và phương thức `doFinal`:

```
public short update(byte[] inBuf, short inOffset, short inLength,
           byte[] outBuff, short outOffset);
public short doFinal (byte[] inBuf, short inOffset,
          short inLength, byte[] outBuff, short outOffset);
```

Cả hai phương thức đều lấy bản rõ trong bộ đệm đầu vào và ghi ra bản mã được tính toán vào bộ đệm đầu ra. Chúng ta nên gọi phương thức `update` để cung cấp dữ liệu đầu vào tích lũy và gọi phương thức `doFinal` khi chúng ta đưa vào lô dữ liệu cuối cùng. Phiên bản cuối cùng của bản mã được tính toán trong bộ đệm đầu ra từ phương thức `doFinal`. Sau khi phương thức `doFinal` được gọi đến, đối tượng `Cipher` được đặt lại về trạng thái khi nó được khởi tạo trước đó thông qua phương thức `init`. Để giải mã dữ liệu, cũng gọi phương thức `update` và `doFinal`. Nhưng lưu ý rằng để giải mã, bản mã được đưa vào bộ đệm đầu vào và bản rõ được ghi vào bộ đệm đầu ra. Một lần nữa, vì phương thức `update` liên quan đến chi phí lưu trữ kết quả nội bộ, chúng ta chỉ nên gọi nó nếu toàn bộ dữ liệu đầu vào không thể vừa trong một mảng byte.

Dưới đây là ví dụ Applet dùng để mã hóa và giải mã:

```
package bai6_Crypto;

import javacard.framework.*;
import javacardx.crypto.Cipher;
```

```
import javacard.security.*;


public class Applet4_Cipher extends Applet
{
     private static final byte INS_ENCRYPT   = (byte)0x00;
     private static final byte INS_DECRYPT   = (byte)0x01;
     private byte[] in, enc_buffer, dec_buffer, keyData;

     private AESKey aesKey;
     private Cipher cipher;
     private short keyLen;

     private Applet4_Cipher()
     {

          keyLen         = (short)(KeyBuilder.LENGTH_AES_128/8);
          in             = new byte[keyLen];
          enc_buffer     = new byte[keyLen];
          dec_buffer= new byte[keyLen];
          keyData        = new byte[keyLen];

          for (byte i = 0; i < (byte)keyLen; i++){
               keyData[i]     = (byte)i;
               in[i]     = (byte)(i+1);
          }

          cipher =
Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_ECB_NOPAD, false);
          aesKey =
(AESKey)KeyBuilder.buildKey(KeyBuilder.TYPE_AES,
(short)(8*keyLen), false);
```

```
          aesKey.setKey(keyData, (short)0);
     }

     public static void install(byte[] bArray, short bOffset,
byte bLength)
     {
          new Applet4_Cipher().register(bArray, (short) (bOffset
+ 1), bArray[bOffset]);
     }

     public void process(APDU apdu)
     {
          if (selectingApplet())
          {
               return;
          }

          byte[] buf = apdu.getBuffer();
          apdu.setIncomingAndReceive();
          switch (buf[ISO7816.OFFSET_INS])
          {
          case INS_ENCRYPT:
               encrypt(apdu);
               break;
          case INS_DECRYPT:
               decrypt(apdu);
               break;
          default:

     ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
          }
```

```
    }
    private void encrypt(APDU apdu)
    {
        byte[] buf = apdu.getBuffer();
        cipher.init(aesKey, Cipher.MODE_ENCRYPT);
        cipher.doFinal(in, (short)0, keyLen, enc_buffer,
(short)0);
        Util.arrayCopy(enc_buffer, (short)0, buf, (short)0,
keyLen);
        apdu.setOutgoingAndSend((short)0, keyLen);
    }

    private void decrypt(APDU apdu)
    {
        byte[] buf = apdu.getBuffer();
        cipher.init(aesKey, Cipher.MODE_DECRYPT);
        cipher.doFinal(enc_buffer, (short)0, keyLen,
dec_buffer, (short)0);
        Util.arrayCopy(dec_buffer, (short)0, buf, (short)0,
keyLen);
        apdu.setOutgoingAndSend((short)0, keyLen);
    }

}
```

Kết quả thực thi Applet này như sau:

```
>> /select 11223344550604
>> 00 A4 04 00 07 11 22 33 44 55 06 04 00
<< 90 00

>> /send 00000102
>> 00 00 01 02
<< 08 92 08 56 05 BE 8F 34 9F 58 4A F9 93 DF 11 F8 90 00
```

```
>> /send 00010102
>> 00 01 01 02
<< 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 90 00
```

6.4 Tạo dữ liệu ngẫu nhiên

Số ngẫu nhiên là thường xuyên cần thiết cho các thủ tục mật mã. Để tạo một trình tạo số ngẫu nhiên, chúng ta gọi phương thức `getInstance` trong lớp `javacard.security.RandomData` và chỉ định một thuật toán. Tham số lựa chọn thuật toán có thể là `RandomData.ALG_PSEUDO_RANDOM` cho thuật toán tạo số giả ngẫu nhiên hoặc `RandomData.ALG_SECURE_RANDOM` cho thuật toán tạo số ngẫu nhiên an toàn mạnh về mật mã.

Giống như các lớp dựa trên thuật toán khác trong API mã hóa Java Card, lớp `javacard.security.RandomData` là một lớp cơ sở trừu tượng. Do đó, nó phải được mở rộng bởi một lớp thực thi. Đối tượng `RandomData` được trả về từ phương thức `getInstance` là một đối tượng của lớp thực thi như vậy, thực hiện thuật toán mong muốn.

Mầm giả ngẫu nhiên của đối tượng `RandomData` được khởi tạo thành một giá trị mặc định bên trong, khi đối tượng `RandomData` an toàn cố gắng khởi tạo mầm với một giá trị hoàn toàn ngẫu nhiên. Chúng ta có thể tạo thế hệ số ngẫu nhiên thông qua phương thức `setSeed`.

Cuối cùng, để có được một số ngẫu nhiên, chúng ta gọi phương thức `GenerateData`. Kết quả chúng ta có thể xây dựng Applet tạo dữ liệu ngẫu nhiên như sau:

```
package bai6_Crypto;

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

public class Applet5_Random extends Applet
```

```
{
     private byte[] seed;
     private RandomData ranData;

     public static void install(byte[] bArray, short bOffset,
byte bLength)
     {
          new Applet5_Random().register(bArray, (short) (bOffset
+ 1), bArray[bOffset]);
     }

     public void process(APDU apdu)
     {
          if (selectingApplet())
          {
               return;
          }

          byte[] buf = apdu.getBuffer();
          apdu.setIncomingAndReceive();
          switch (buf[ISO7816.OFFSET_INS])
          {
          case (byte)0x00:
               seed = new byte[] {0x01, 0x02, 0x03}; //mam ngau
nhien
               ranData                                          =
RandomData.getInstance(RandomData.ALG_PSEUDO_RANDOM);
               //cung cap mam ngau nhien
```

```
                ranData.setSeed(seed,                    (short)0,
(short)(seed.length));
                short ranLen = (short)10;
                // sinh du lieu ngau nhien
                ranData.generateData(buf, (short)0, ranLen);
                apdu.setOutgoingAndSend((short)0, ranLen);

                break;
           default:

     ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
           }
     }
}
```

Kết quả thực thi Applet trên như sau:

```
>> /select 11223344550605
>> 00 A4 04 00 07 11 22 33 44 55 06 05 00
<< 90 00

>> /send 00000102
>> 00 00 01 02
<< 5E D2 D9 F9 74 3B 72 46 9B 2A 90 00

>> /send 00000102
>> 00 00 01 02
<< 67 3C 43 E6 52 BB 00 89 68 C7 90 00
```