Instituto Superior Técnico, Universidade de Lisboa

**Highly Dependable Systems**

# Java Cryptographic Mechanisms

## Important premise

This laboratory assignment covers the basic Java cryptographic mechanisms that will be used throughout the course and the project. This assignment is based on a similar assignment from the SIRS course and extended here with SEC-specific exercises. This serves as an introduction to cryptography for students who did not take the SIRS course and as a refresher for those students who did. It is strongly recommended that all students get comfortable with the Java Crypto library before starting working on the project.

## Goals

- Refresh/learn how to use the cryptographic mechanisms available in the Java platform.
- Perform attacks exploiting vulnerabilities introduced by the bad use of cryptography.
- Use cryptography in a client-server environment.

## 1. Introduction

On the laboratory page there are two versions of the source code, one that works with Java Development Kit (JDK) version 12 and another that works with an older version of the JDK (version 7 and higher) running over Linux. It is recommended that students use the latest version, the other one is provided for compatibility with older installations.

The Java platform strongly emphasizes security, including language safety, cryptography, public key infrastructure, secure communication, authentication and access control. The Java Cryptography Architecture (JCA), which is a major piece of the Java platform, includes a large set of application programming interfaces (APIs), tools, and implementations of commonly-used security algorithms, mechanisms, and protocols. It provides a comprehensive security framework for writing applications and also provides a set of tools to securely manage applications.

The JCA APIs include abstractions for secure random number generation, key generation and management, certificates and certificate validation, encryption (symmetric/asymmetric block/stream ciphers), message digests (hashes), and digital signatures. Some examples are the `MessageDigest`, `Signature`, `KeyFactory`, `KeyPairGenerator`, and `Cipher` classes. In the laboratory page there is a guide that covers the common APIs and their usage. Implementation independence, in the Java

platform, is achieved using a *provider*-based architecture. The term Cryptographic Service Provider (CSP) refers to a package or set of packages that implement one or more cryptographic services, such as digital signature algorithms, message digest algorithms, and key conversion services. A program may simply request a particular type of object, e.g., a `Signature` object, implementing a particular service, e.g., the RSA signature algorithm, and get an implementation from one of the installed providers. A program may instead request, if necessary, an implementation from a specific provider. In addition, for example, providers may be updated transparently to the application when faster or more secure versions are available. In the Java platform, the `java.security.Provider` class is the base class for all security providers. Each CSP contains an instance of this class, which contains the provider's name and lists all of the security services/algorithms it implements. Multiple providers may be configured at the same time, and are listed in order of preference. The highest priority provider that implements that service is selected when a security service is requested.

To obtain a security service from an underlying provider, applications rely on the relevant `getInstance()` method. The message digest creation, for example, represents one type of service available from providers. To obtain an implementation of a specific message digest algorithm such as SHA1, an application invokes the `getInstance()` method in the `java.security.MessageDigest` class.

```
MessageDigest md = MessageDigest.getInstance("SHA1");
```

Optionally, by indicating the provider name, the program may request an implementation from a specific provider as in the following piece of code:

```
MessageDigest md = MessageDigest.getInstance("SHA1", "MyProvider");
```

## 2. Cryptographic mechanisms

Extract the `lab1-javaX.zip` package where X corresponds to the version of your JDK into `/tmp/sirs`. Please note that all steps that follow expect that this was done, so you must change commands according to an alternative location if used.

Open the JavaCrypto directory inside the extracted directory and compile all java programs into bytecodes (.class files):

```
$ javac -d out src/pt/ulisboa/tecnico/meic/sirs/*.java
```

This will generate an out directory with all the .class bytecodes.

**Note**: For every java command, please write `$java -cp out pt.ulisboa.tecnico.meic.sirs .RandomImageGenerator` instead of just `$java RandomImageGenerator`. The package name is omitted for brevity.

In the directory `intro/inputs`, you can find 3 different images:

- Tecnico: `*.png`, the IST logo
- Tux: `*.png`, Tux, the Linux penguin
- Glider: `*.png`, the hacker emblem (http://www.catb.org/hacker-emblem/)

Each one is presented with three different dimensions: 480x480, 960x960, and 2400x2400. The `ImageMixer` class is available to facilitate the operations on images. Many examples are available, such as the `RandomImageGenerator`, `ImageXor`, and `ImageAESCipher` classes.

### 2.1 One-Time Pads (Symmetric stream cipher)

When correctly used, one-time pads (OTPs) provide perfect security. One of the constraints to make them work as expected is that they must never be reused. The following steps visually illustrate what happens if they are reused, even if just once:

1. Generate a new 480x480 random image

```
$ java RandomImageGenerator /tmp/sirs/intro/outputs/otp.png 480 480
```

2. Perform the bitwise eXclusive OR operation (XOR) with the generated key

```
$ java ImageXor /tmp/sirs/intro/inputs/tecnico-0480.png
/tmp/sirs/intro/outputs/otp.png /tmp/sirs/intro/outputs/encrypted-
tecnico.png
```

3. XOR tux-0480.png with the same generated key

```
$ java ImageXor /tmp/sirs/intro/inputs/tux-0480.png
/tmp/sirs/intro/outputs/otp.png /tmp/sirs/intro/outputs/encrypted-
tux.png
```

4. Watch the images `encrypted-tecnico.png` and `encrypted-tux.png`. Switch between them, and see the differences.
5. Make the differences obvious: XOR them together:

```
$ java ImageXor /tmp/sirs/intro/outputs/encrypted-tecnico.png
/tmp/sirs/intro/outputs/encrypted-tux.png
/tmp/sirs/intro/outputs/tecnico-tux.png
```

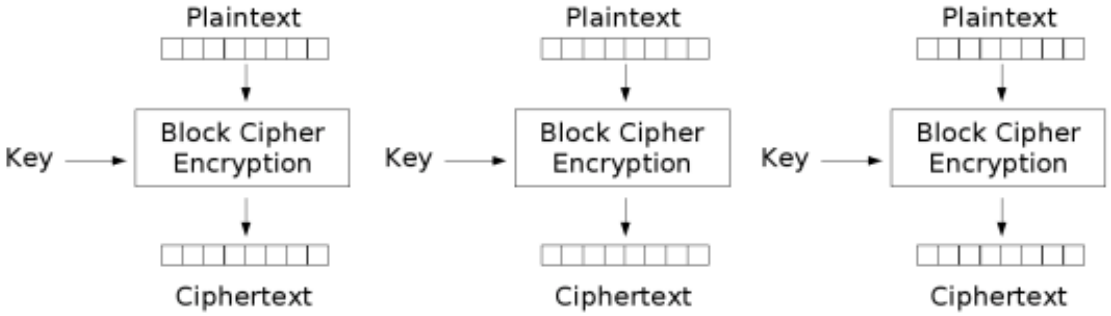
You can see that the reuse of an OTP (or any stream cipher key at all) considerably weakens (or completely breaks) the security of the information. Feel free to experiment with other images and sizes.

## 2.2 Block cipher modes

Now that you know that keys should never be reused, remember that the way you use them is also important. You are about to use a symmetric-key encryption algorithm in modes ECB (Electronic Code Book), CBC (Cipher Block Chaining) and OFB (Output Feedback), to encrypt the pixels from an image.

### 2.2.1 ECB

In the ECB mode, each block $m[i]$ is encrypted with key $k$ independently: $c[i] = E_k(m[i])$




1. Begin by generating a new AES Key.

```
$ java AESKeyGenerator w /tmp/sirs/intro/outputs/aes.key
```

2. Then, encrypt the glider image with it:

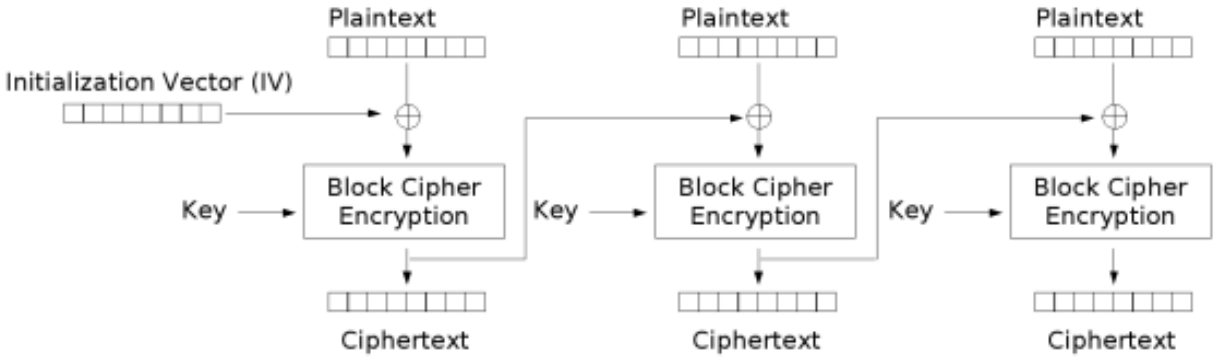
```
$ java ImageAESCipher /tmp/sirs/intro/inputs/glider-0480.png
/tmp/sirs/intro/outputs/aes.key ECB /tmp/sirs/intro/outputs/glider-aes-
ecb.png
```

3. Watch the output image. Remember what you have just done: encrypted the image with AES, using ECB mode, and a key you generated yourself.
4. Feel free to try the same thing with the other images (especially with other sizes).

5. Try using Java providers to generate a new DES key, based on the `AESKeyGenerator` class. What is necessary to change in the code for that to happen?
6. Repeat all the previous steps for the new key.
7. Compare the results obtained using ECB mode with DES with the previous ones. What are the differences between them?

### 2.2.2 CBC

In the CBC mode, each block $m[i]$ is XORed with the ciphertext from the previous block, and then encrypted with key $k$: $c[i] = E_k(m[i] \oplus c[i-1])$.




2.2.2.1 The encryption of the first block can be performed by means of a random and unique value known as the Initialization Vector (IV).

a. The AES key will be the same from the previous step.
b. Encrypt the glider image with it, this time replacing ECB for CBC:

```
$ java ImageAESCipher /tmp/sirs/intro/inputs/glider-0480.png
/tmp/sirs/intro/outputs/aes.key CBC /tmp/sirs/intro/outputs/glider-aes-
cbc.png
```

c. Notice how the file sizes are different between `glider-aes-ecb.png` and `glider-aes-cbc.png`. Can you find a reason why?
d. Watch the file `glider-aes-cbc.png`. See the difference it made, changing only the mode of operation.

2.2.2.2 Still in the CBC mode, you might have wondered why the IV is needed in the first block. Consider what happens when you encrypt two different images with similar beginnings, and with the same key $k$: the initial cipher text blocks will also be similar!

The `ImageAESCipher` class provided has been deliberately weakened: instead of randomizing the IV, it is always the same.

a. This time, encrypt the other two images with AES/CBC, still using the same AES key:

```
$ java ImageAESCipher /tmp/sirs/intro/inputs/tux-0480.png
/tmp/sirs/intro/outputs/aes.key CBC /tmp/sirs/intro/outputs/tux-aes-
cbc.png
```

```
$ java ImageAESCipher /tmp/sirs/intro/inputs/tecnico-0480.png
/tmp/sirs/intro/outputs/aes.key CBC /tmp/sirs/intro/outputs/tecnico-aes-
cbc.png
```

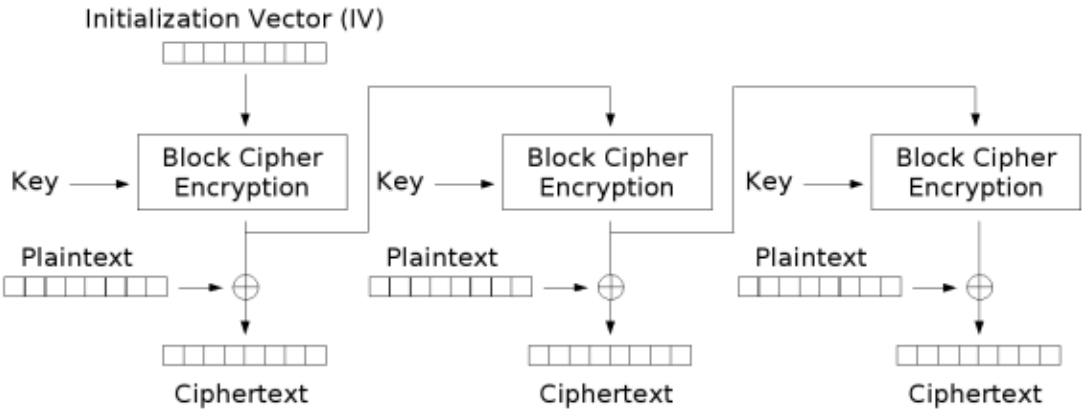
b. Now watch the images `glider-aes-cbc.png`, `tux-aes-cbc.png`, and `tecnico-aes-cbc.png`

Look to the first lines of pixels. Can you see what is going on?

c. Generate a new file by concatenating two png image files. Cipher this new image file. What did you obtain? Now, try doing the previous with two jpeg image files.

### 2.2.3 **OFB**

In the OFB mode, the IV is encrypted with the key to make a keystream that is then XORed with the plaintext to make the cipher text.




In practice, the keystream of the OFB mode can be seen as the one-time pad that is used to encrypt a message. This implies that in OFB mode, if the key and the IV are both reused, there is no security.

1. Encrypt the images with OFB:

```
$ java ImageAESCipher /tmp/sirs/intro/inputs/glider-0480.png
/tmp/sirs/intro/outputs/aes.key OFB /tmp/sirs/intro/outputs/glider-aes-
ofb.png
$ java ImageAESCipher /tmp/sirs/intro/inputs/tux-0480.png
/tmp/sirs/intro/outputs/aes.key OFB /tmp/sirs/intro/outputs/tux-aes-
ofb.png
$ java ImageAESCipher /tmp/sirs/intro/inputs/tecnico-0480.png
/tmp/sirs/intro/outputs/aes.key OFB /tmp/sirs/intro/outputs/tecnico-aes-
ofb.png
```

2. Remember that the `ImageAESCipher` implementation has been weakened, by having a null IV, and you are reusing the same AES key. Watch the generated images, and switch quickly between them.
3. Take 2 images (e.g., image1 and image2) and cipher them both. XOR ciphered image1 with the ciphered image2. What did you obtain? Why?
4. What is more secure to use: CBC or OFB?

## 2.3 Asymmetric ciphers

### 2.3.1 Generating a pair of keys with OpenSSL

1. Private key

```
$ openssl genrsa -out server.key
```

2. Public key:

```
$ openssl rsa -in server.key –pubout > public.key
```

3. Generating a self-signed certificate with these keys:

- Certificate Signing Request, using same key:

```
$ openssl req -new -key server.key -out server.csr
```

- Self-sign:

```
$ openssl x509 -req -days 365 -in server.csr -signkey server.key -out
server.crt
```

- In order for our certificate to be able to sign other certificates, OpenSSL requires that a database exists (a .srl file). Create it:

```
$ echo 01 > server.srl
```

- Then, generating a key to a user is basically repeating the same steps, except that the self-sign no longer happens and is replaced by:

```
$ openssl x509 -req -days 365 -in user.csr -CA server.crt -CAkey
server.key -out user.crt
```

- Sign the file grades.txt with the user certificate:

```
$ openssl dgst -sha256 /tmp/sirs/grades/inputs/grades.txt >
grades.sha256
$ openssl rsautl -sign -inkey server.key -keyform PEM -in grades.sha256
> grades.sig
```

- Verify the signature:

```
$ openssl rsautl -verify -in grades.sig -inkey server.key
```

This output for me is:

```
SHA256(/tmp/sirs/grades/inputs/grades.txt)=
770ddfe97cd0e6d279b9ce780ff060554d8ccbe4b8eccaed364a8fc6e89fd34d
```

And should always match this:

```
$ openssl dgst -sha256 /tmp/sirs/grades/inputs/grades.txt
   SHA256(/tmp/sirs/grades/inputs/grades.txt)=
770ddfe97cd0e6d279b9ce780ff060554d8ccbe4b8eccaed364a8fc6e89fd34d
```

- Verify the user certificate:

```
$ openssl verify -CAfile server.crt user.crt
```

user.crt: OK

### 2.3.2 Generating a pair of keys with Java

1. Generate a new pair of RSA Keys.

```
$ java RSAKeyGenerator w /tmp/sirs/intro/outputs/priv.key
/tmp/sirs/intro/outputs/pub.key
```

2. Based on the `ImageAESCipher` class, create `ImageRSACipher` and `ImageRSADecipher` classes.
3. Encrypted the image with the public key and then decrypt it with the private key.

Feel free to try the same thing with the other images - especially with other sizes, like 2400x2400.

## 3. Additional exercise (tampering with a file)

In the directory `grades/inputs`, you can find the file `grades.txt`, the plaintext of a file with the grades of a course. This flat-file database has a rigid structure: 64 bytes for name, and 16 bytes for each of the other fields, number, age and grade. Unfortunately, you happen to be *Mr. Thomas S. Cook*, and your grade was not on par with the rest of your class because you forgot to study...

1. Begin by encrypting this file into `grades.ecb.aes`, `grades.cbc.aes` and `grades.ofb.aes`. For this example, we'll still reuse the AES key generated above.

```
$java FileAESCipher /tmp/sirs/grades/inputs/grades.txt
/tmp/sirs/intro/outputs/aes.key ECB
/tmp/sirs/grades/outputs/grades.ecb.aes
$java FileAESCipher /tmp/sirs/grades/inputs/grades.txt
/tmp/sirs/intro/outputs/aes.key CBC
/tmp/sirs/grades/outputs/grades.cbc.aes
$java FileAESCipher /tmp/sirs/grades/inputs/grades.txt
/tmp/sirs/intro/outputs/aes.key OFB
/tmp/sirs/grades/outputs/grades.ofb.aes
```

2. Keeping in mind how the mode operations work, and without using the secret key, try to change your grade to 21 in the encrypted files or give everyone in class a 20. Your goal here is to show that the system is vulnerable, and not to perform actual academic cheating. Did you succeed? Did your changes have side effects? If so, which ones, and in which block modes?
3. Since the inputs and outputs of cryptographic mechanisms are byte arrays, in many occasions it is necessary to represent encrypted data in text files. A possibility is to use Base64 encoding that, for every binary sequence of 6 bits, assigns a predefined ASCII character. Execute the following to create a Base64 representation of files previously generated.

```
$java Base64Encode /tmp/sirs/grades/outputs/grades.cbc.aes
/tmp/sirs/grades/outputs/grades.cbc.aes.b64
```

4. Decode them:

```
$java Base64Decode /tmp/sirs/grades/outputs/grades.cbc.aes.b64
/tmp/sirs/grades/outputs/grades.cbc.aes.b64.decoded
```

5. Check if they are similar using the `diff` command (or `fc /b` command on Windows).

```
$diff /tmp/sirs/grades/outputs/grades.cbc.aes
/tmp/sirs/grades/outputs/grades.cbc.aes.b64.decoded
```

It should not return anything.

6. Check the difference on the file sizes. Can you explain it? In percent, how much is it? Does Base64 provide any kind of security? If so, how?

7. Use Java to generate the message authentication code (MAC) and digital signature of the file. By performing these operations which security requirements are guaranteed?

## 4. Cryptography in a client-server environment

To conclude this introduction to cryptography, you will apply the knowledge acquired throughout this guide in a client-server application.

1. Construct a simple client-server application in Java. The client should send messages to the server, and the server should echo them back to the client. You should implement this in Java and you can use any communication library you want (examples include java.net, Spring Boot, Java RMI, JAX-WS, etc).
2. Generate a key-pair for the server using the `RSAKeyGenerator` tool (explained in Section 2.3). The server should digitally sign the messages sent to the client. The client should verify if the server responses are correctly signed (assume that the server's public key was previously distributed in a secure manner).
3. Repeat the previous step for the client-side. Now the messages sent by the client should also be signed, and these signatures should be verified by the server.
   At this stage, which security properties does your communication protocol ensure? (Confidentiality? Authentication? Integrity? Non-repudiation?)
4. Generate an AES key using the `AESKeyGenerator` tool provided (shown in Section 2.2). Encrypt both the requests and responses using AES in an encryption mode of your choice (be careful with the correct algorithms relative to the key generated). Messages should still be signed. Why is this important? With this step, did you change the security properties provided by your communication protocol?

# Acknowledgments

Part of this assignment was prepared by Professors Ricardo Chaves and Miguel Pardal, in the context of the SIRS course. Some of the exercises are based on the work developed by the student Valmiky Arquissandas within the scope of his project work for the SIRS course, and later refined and adapted by the teaching staff of the SIRS course. The assignment was revised and extended for the SEC course by João Gonçalves and João Martinho.