SE448: Information and Cyber Security **Recitation 6**
Semester 2 5785 **8 May 2025**
Lecturer: Michael J. May **Kinneret College**

Using RSA

In this recitation we will write code in Java that performs the encryptions and decryptions for us.

1 Using RSA in Java

We will build a tool that lets you perform RSA encryptions on files. All of the work for the algorithm takes place under the hood, so there isn't a lot to see here. Still, it's important for you to recognize the classes that offer the RSA cryptographic services for later use in this class (and outside).

1.1 User Interface

We're going to make a command line tool as before. The tool has three operations that it can perform:

1. Generate a new RSA key pair with a given key length and store the key in an output file
2. Encrypt a file using a provided RSA key pair
3. Decrypt a file using a provided RSA key pair

To support the operations, the tool must support the following parameters:

outfile The output file path (after encryption or decryption)

infile The file to process (encrypt or decrypt)

keyfile The file containing the key (for encrypt/decrypt) or to write the key out to (key generation)

keylength The length of the key to be generated (in bits)

suite The suite to use - RSA/ECB/OAEPWithSHA-256AndMGF1Padding or RSA/ECB/PKCS1Padding

op Can be encrypt, decrypt, generatekey

When asking to generate a key, the following parameters are required:

1. operation
2. keylength
3. keyfile

When asking to encrypt or decrypt a file, the following parameters are required:

1. outfile
2. infile
3. keyfile
4. suite
5. op

1.2 The Key Objects

There are a few key classes that are crucial for Java support for the RSA algorithm:

KeyPairGenerator Generates RSA key pairs for you. You just need to give it the algorithm that you want it to create key pairs for. In this case, we'll use "RSA". It also supports other algorithms that we are not going to get to in this course (*e.g.* El Gamal and DSA). Before you can use the generator, you'll need to initialize it using the `initialize` method along with the key pair length that you want.

KeyPair Represents a public/private key pair. It's generated by the KeyPairGenerator using the `generateKeyPair` method.

RSAPrivateKey The private part of the KeyPair. You can extract it from the KeyPair using the `getPrivate` method.

RSAPublicKey The public part of the KeyPair. You can extract it from the KeyPair using the `getPublic` method.

1.3 Saving and Loading Keys

There are a few ways to store private and public keys on a computer. One option is to use a PEM file. Another is to use an X.509 digital certificate. Both options work, but involve significant additional work in the tool. To save time, I implemented a trivial save and restore mechanism for the private and public keys in hexadecimal format in a text file. The methods are already given to you in the starter code to save time. You can have a look at them to see what they do.

2 Encrypting and Decrypting with Java

We'll use the `Cipher` object to encrypt and decrypt, just like before. Remember the following:

- When encrypting, use the public key
- When decrypting, use the private key

Remember to read the file in a byte[]. You can then use either the `update(byte[])` or `doFinal(byte[])` methods of Cipher to encrypt or decrypt.

2.1 About Padding

We will use the `Cipher` object for encrypting and decrypting. This time we'll ask it to use one of two encryption/decryption cipher suites:

1. RSA/ECB/OAEPWithSHA-256AndMGF1Padding
2. RSA/ECB/PKCS1Padding

The difference between the two has to do with the padding algorithm that the cipher function uses. Both padding functions are designed to overcome the RSA homomorphism property mentioned in class - thereby making trivial changes to ciphertext not work. Any such naive change would mess up the padding, making the message not decrypt properly.

OAEP adds another level of protection on the encryption by adding a random number to the end of the message and using a hash algorithm to ensure completeness. A benefit to using OAEP is that due to its use of a random number, if you encrypt the same message multiple times using OAEP, the ciphertext will appear different each time. This means that an attacker will not be able to discern whether two encrypted messages contain the same message inside or not. It's a step toward IND-CPA and IND-CCA that we mentioned before.

As with all padding algorithm, the padding makes the ciphertext longer than the plaintext. We'll come back to this issue in one of the experiments below.

3 What to do: Fill in the code

Use the code and tool provided to perform the following tasks:

1. Connect the code to create and save the RSA key pair information.
2. Fill in the functions to initialize the KeyPairGenerator, `encrypt` for encryption, and `decrypt` for decryption using the classes mentioned above. Encryption and Decryption work very similarly to what we have seen before with DES and AES.
3. Make sure the tool supports both padding algorithms mentioned above.
4. As with AES, `Cipher` works only over `byte[]`, so we must read all file data as bytes[]. To make it easier, also output encrypted or decrypted file contents as bytes.
5. Check your implementation for correctness. Due to the randomness included in encryption, you can't compare encrypted outputs to see if they are correct. The best you can do is to attempt to decrypt the ciphertext you got with a particular key pair. I put a key pair on Moodle that you can use to test. It should work as follows:
   - TestFile1.txt decrypts to "Hello world!" NOT using OAEP with key1.txt
   - TestFile2.txt decrypts to "Hello world!" using OAEP with key1.txt
   - TestFile3.txt decrypts to "It is a truth universally acknowledged, that a single man in possession of a good fortune, must be in want of a wife." NOT using OAEP with key2.txt.
   - TestFile4.txt decrypts to "It is a truth universally acknowledged, that a single man in possession of a good fortune, must be in want of a wife." using OAEP with key2.txt.

4 Experiments

4.1 Experiment 1: Encryptions

1. Generate a new 1,024 bit key. Store it in a file called Experiment1-Key.txt
2. Create a file with the text: "It was a picture of a boa constrictor". Encrypt it using RSA/ECB/PKCS1Padding and Experiment1-Key.txt key. Store the result in Experiment1-Step2.txt
3. Encrypt the file again using RSA/ECB/PKCS1Padding. Store the result in Experiment1-Step3.txt. Compare the contents of Experiment1-Step2.txt and Experiment1-Step3.txt. Are they the same?
4. Decrypt Experiment1-Step2.txt using Experiment1-Key.txt key. Store the result in a file. What is the result?
5. Decrypt Experiment1-Step3.txt using Experiment1-Key.txt key. Store the result in a file. What is the result?

What is going on here?

Perform the encryption again with RSA/ECB/OAEPWithSHA-256AndMGF1Padding. What is going on here?

4.2 Experiment 2: File Length (PKCS)

1. Generate a new 1,024 bit key. Store it in a file called Experiment2-Key.txt.
2. Create a file with the text: "I pondered deeply, then, over the adventures of the jungle." Encrypt it using RSA/ECB/PKCS1Padding and Experiment2-Key.txt key. Store the result in Experiment2-Step2.txt.
3. Create a new file with the text: "I pondered deeply, then, over the adventures of the jungle. And after some work with a colored pencil I succeeded in making my first drawing." Encrypt it using RSA/ECB/PKCS1Padding and Experiment2-Key.txt key. Store the result in Experiment2-Step3.txt. What happens? Why?
4. Figure out the maximum length text that you can successfully encrypt.
5. Generate a new 2,048 bit key. Store it in a file called Experiment2-Key-Longer.txt.
6. Attempt to encrypt the second file ("I pondered deeply, then, over the adventures of the jungle. And after some work with a colored pencil I succeeded in making my first drawing.") with Experiment2-Key-Longer. What happens? Figure out the maximum length text you can successfully encrypt.

What can you tell about the way Java limits the length of text inputs with PKCS?

4.3 Experiment 3: File Length (OAEP)

1. Generate a new 1,024 bit key. Store it in a file called Experiment3-Key.txt.
2. Create a file with the text: "I pondered deeply, then, over the adventures of the jungle." Encrypt it using RSA/ECB/RSA/ECB/OAEPWithSHA-256AndMGF1Padding and Experiment3-Key.txt key. Store the result in Experiment3-Step2.txt.
3. Create a new file with the text: "I pondered deeply, then, over the adventures of the jungle. And after some work with a colored pencil I succeeded in making my first drawing." Encrypt it using RSA/ECB/RSA/ECB/OAEPWithSHA-256AndMGF1Padding and Experiment3-Key.txt key. Store the result in Experiment3-Step3.txt. What happens? Why?
4. Figure out the maximum length text that you can successfully encrypt.
5. Generate a new 2,048 bit key. Store it in a file called Experiment2-Key-Longer.txt.
6. Attempt to encrypt the second file ("I pondered deeply, then, over the adventures of the jungle. And after some work with a colored pencil I succeeded in making my first drawing.") with Experiment2-Key-Longer. What happens? Figure out the maximum length text you can successfully encrypt.

What can you tell about the way Java limits the length of text inputs with OAEP?

4.4 Experiment 4: Extending the RSA Testing Application

In the previous experiment you saw that Java limits the length of the input text for encryption. Adapt the code provided to permit the encryption and decryption of unlimited length files.

Note: You will need to do some work in the parsing functions as well.

If you want to enable unlimited length files, you'll need to do some block reduction akin to ECB. To make it work, you'll need to consider the following limitations of RSA:

1. The output size of encryption is equal to the length of the key (*ex.* 4096 bit key = 512 Byte key $\rightarrow$ 512 Byte ciphertext)
2. The maximum input size is equal to:
   (a) Under PKCS: The key length - 11 B (*ex.* 4096 bit key = 512 Byte key - 11 B (padding) = 501 Bytes input maximum)
   (b) Under OAEP with SHA-256: The key length - 66 B (*ex.* 4096 bit key = 512 Byte key - 66 B (padding) = 446 Bytes input maximum)

You can use those equations to properly cut your input blocks into chunks for encryption. You'll need to do a bit of a different algorithm for decryption (cut the ciphertext into key size (in bytes) blocks, then decrypt).