# Practical 1 – Cryptography

In this practical we will explore key characteristics and pitfalls of modern stream and block ciphers.

## Stream ciphers

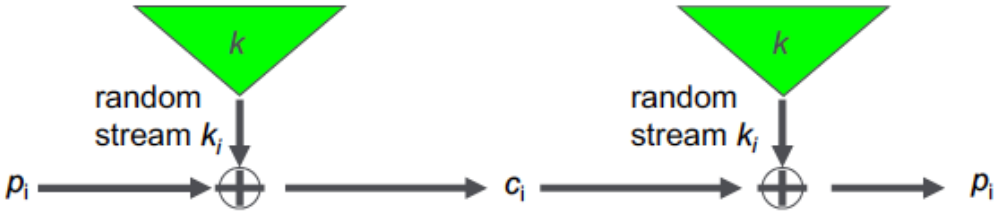


**Fig. 1: Stream cipher encryption and decryption**

Stream ciphers perform their encryption by combining the plaintext bits with the bits of a *key stream*. As shown in Fig. 1, the *i*-th bit of ciphertext, $c_i$, is calculated as the *exclusive or* between the *i*-th bit of plaintext, $p_i$, and the *i*-th bit of the key stream, $k_i$:

$$c_i = p_i \oplus k_i$$

The decryption is done using the same operation, with the *same key stream*:

$$p_i = c_i \oplus k_i$$

This works because $c_i \oplus k_i = (p_i \oplus k_i) \oplus k_i = p_i \oplus (k_i \oplus k_i) = p_i \oplus 0 = p_i$. Table 1 reminds you the truth table(s) for the exclusive or operation.

**Table 1: Exclusive or truth tables**

| ⊕ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

| ⊕ | false | true |
|---|---|---|
| false | false | true |
| true | true | false |

### Exercise 1

The first exercise requires you to complete the implementation of a Java emulator for the Trivium stream cipher [1]. Trivium is one of the three hardware-oriented ciphers that made it into the eSTREAM stream cipher portfolio [2] following a major international competition to design new efficient stream ciphers.

Fig. 2 shows the three shift registers of Trivium, how their elements are connected to their outputs (labelled *resultA*, *resultB* and *resultC*), and how these outputs are used to produce the next bit of the key stream (labelled $k_i$), and the next leftmost bits of the registers (i.e. A[0], B[0] and C[0]).

As you learnt in the lectures, Trivium obtains its key stream by repeatedly right-shifting its registers in successive steps in which:

(i) The next leftmost bit of each register is computed as a combination of several existing bits, e.g. A[0] = A[68] ⊕ *resultC*.

(ii) The next bit of the key stream, $k_i$, is also computed as a combination of several register bits.

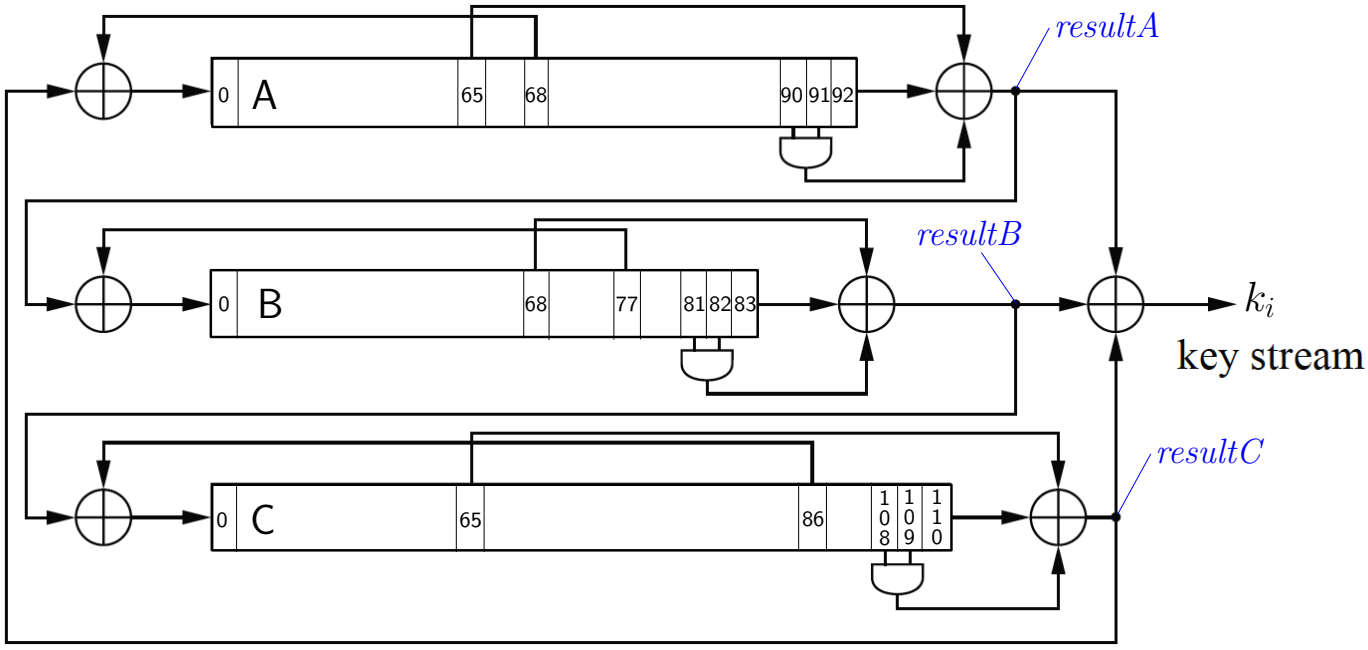


**Fig. 2: Architecture of the Trivium cipher (adapted from [3])**

Download the files `Trivium.java` and `TriviumTest.java` from the 'Practical 1' area of the module VLE, and organise them into an Eclipse Java project.

`Trivium.java` is a Trivium emulator from which the implementation of the `step` method for updating the content of the three registers and producing the next key stream bit is partly missing. Your task is to complete the implementation of this method.

For simplicity, the emulator uses Boolean arrays for the three registers. This is inefficient, but greatly simplifies the implementation, e.g. the AND gate below register C can be encoded in Java as

```
C[108] && C[109]
```

and the next leftmost bit of register A can be calculated as:

```
boolean nextLeftmostBitA = A[68] ^ resultC;
```

Before implementing the missing code, see how the cipher is initialised and "warmed up" in the Trivium class constructor (the two operations are described on slide 36 from the Cryptography lecture slides).

Have a look at the simple test from `TriviumTest.java`, which simulates the transmission of a Trivium-encrypted "secret" message between a sender and a receiver (you should not modify this file). Run the test to check that your implementation of the cipher produces the correct output below:

```
Original message:    This is a secret message
Plaintext:           546869732069732061207365637265742064657373616765
Ciphertext:          C3C4E092CDA750732EE5136F6D51DC6CD43C4FA106ECBC46
Decrypted plaintext: 546869732069732061207365637265742064657373616765
Decrypted message:   This is a secret message
```

## Exercise 2

This exercise shows you how the incorrect use of a strong (stream) cipher enables simple attacks on the ciphertext. Your task is to break the Trivium encryption of two messages that were encoded using *the same key stream* – key streams must not be reused!

Download the file `TriviumAttack.java`, which contains two ciphertexts (`ciphertext1` and `ciphertext2`) encrypted using our Trivium stream cipher emulator *with the same key and initialisation vector*. Although the key and initialisation vector (and therefore the key stream) are unknown, we know that

$$\texttt{ciphertext1} = \texttt{plaintext1} \oplus \texttt{unknownKeyStream}$$
$$\texttt{ciphertext2} = \texttt{plaintext2} \oplus \texttt{unknownKeyStream}$$

Therefore, we can calculate

$$\begin{aligned}\texttt{ciphertext1} \oplus \texttt{ciphertext2} &= (\texttt{plaintext1} \oplus \texttt{unknownKeyStream}) \oplus (\texttt{plaintext2} \oplus \texttt{unknownKeyStream}) = \\ &= (\texttt{plaintext1} \oplus \texttt{plaintext2}) \oplus (\texttt{unknownKeyStream} \oplus \texttt{unknownKeyStream}) = \\ &= (\texttt{plaintext1} \oplus \texttt{plaintext2}) \oplus 0 = \\ &= \texttt{plaintext1} \oplus \texttt{plaintext2}.\end{aligned}$$

A "brute force" dictionary attack can be carried out on $\texttt{plaintext1} \oplus \texttt{plaintext2}$ by computing the exclusive or of different word pairs until a match is obtained.

For simplicity, each of the two ciphertexts from `TriviumAttack.java` encodes a seven-character word.[1] Therefore, the method `bruteForceAttack` that you are required to implement in `TriviumAttack.java` must iterate through all pairs (`word1, word2`) of seven-character dictionary words until

$$\texttt{word1} \oplus \texttt{word2} = \texttt{plaintext1} \oplus \texttt{plaintext2}.$$

The following preliminary steps of the attack are already implemented for you:

1. The calculation of $\texttt{ciphertext1} \oplus \texttt{ciphertext2}$ (or, equivalently, $\texttt{plaintext1} \oplus \texttt{plaintext2}$). The first parameter of `bruteForceAttack` (i.e. `byte[] expectedPlaintextXOR`) is the result of this calculation.

2. The extraction of all seven-character words from a suitable dictionary. The second parameter of `bruteForceAttack` (i.e., `String[] words`) is an array containing these words.

For the second step to work, you need to download `dictionary.tex` from the module VLE, and to set the path from the first line of the `readDictionary` method to the actual location of the file on your computer.

Finally, two functions that will help with your implementation of the attack are:

- The `getBytes` method from the String class, which converts a string into a byte array.
- The `Arrays.equals` method, which returns true if and only if its two array parameters are identical.

Check the two words revealed by your "attack" with the lecturer or a teaching assistant (or compare them to the words produced by the model solution provided on the VLE after the practical session).

Most importantly, remember that using a strong cipher is not enough to ensure the confidentiality of sensitive data. It is also essential that the cipher is used correctly.

## Block ciphers

Block ciphers encrypt a block of plaintext units (e.g. bits or characters) at a time. As with stream ciphers, they fail to provide the expected protection when used incorrectly. The next exercise illustrates this for the encryption of images with the widely-used AES (Advanced Encryption Standard) cipher.

[1] Modern computers can carry out this attack very efficiently even when the two plaintexts contain multiple words that do not overlap perfectly and/or have different lengths.

### Exercise 3

Download `AESEncryptImage.java` and the image files `tree.png`, `house.jpg` and `elephant.png` from the practical area on the VLE.

Examine how the `javax.crypto` package can be used to generate an AES `SecretKey`, to instantiate and initialise an AES `Cipher` based on this key, and to perform encryption and decryption. Notice that the instantiation of the cipher, e.g.

```
Cipher aesCipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
```

requires the specification of the cipher type (`AES` in this example), mode of operation (Electronic Code Book, or `ECB`, in the example) and the data padding to use for the plaintext being encrypted (if the plaintext size is not a multiple of the block size) or that was used to encode the ciphertext being decrypted.

Remind yourselves about ECB and other modes of operation for block ciphers by having a quick read through the relevant section from the '*Cryptography and Security Protocols*' lecture slides – this section starts on slide 46.

Setting the `path` and `imageFileName` fields from `AESEncryptImage.java` appropriately, run the Java program to encrypt and decrypt each of the three images. Have a look at the encrypted and decrypted image files (created in the same folder as the original images, and named `encrypted-tree.png`, `decrypted-tree.png`, etc.).

Unsurprisingly, the decrypted images match the original images. What do you observe in the encrypted image files though? How do you explain this? Is AES not such a strong cipher after all?

### Exercise 4

Modify `AESEncryptImage.java` from the previous exercise to enforce the use of the Cipher Block Chaining (`CBC`) mode of operation for the AES `Cipher`. Make sure you change the mode of operation both in the encryption and in the decryption function.

Look again at the lecture slide describing the CBC mode of operation for block ciphers (slide 48). Notice that this mode of operation requires an *initialisation vector* (IV). Therefore, you will need to create a `javax.crypto.spec.IvParameterSpec` initialisation vector,[2] and to use it as a third argument for the `init` method of the AES cipher. Make sure you use the same IV for both encryption and decryption.

Run the modified Java program to encrypt and decrypt again the three images from the previous exercise. How do the encrypted images look like this time?

We conclude that – similarly to stream ciphers – the level of protection provided by strong block ciphers depends on how well they are used!

## Extension: Symmetric versus public ciphers

In this (optional) part of the practical, you will compare the efficiency of symmetric and public key ciphers.

### Exercise 5

Download the Java class `AESvsRSA.java` from the module VLE, and complete the implementation of its `initAES` method, which must return an AES cipher initialised for encryption. Your new code should resemble

[2] Use one of the IvParameterSpec constructors described in the javax.crypto.spec.IvParameterSpec documentation at https://docs.oracle.com/javase/7/docs/api/javax/crypto/spec/IvParameterSpec.html, remembering that the IV size for our AES cipher must be 128 bits (i.e. 16 bytes).

the code used to create AES ciphers in the previous two exercises. A similar method (i.e. `initRSA`) that returns an RSA public-key cipher initialised for encryption is already implemented for you.

To compare the encryption times of AES and RSA, generate a large (e.g. 500KB or 1MB) `plaintext` byte array and fill it with random values[3] at the beginning of the `main` method. Run the Java program to see the time required to encrypt your large `plaintext` using each of the ciphers.

What is the difference in encryption time between the two ciphers? Is the established approach of using AES to encrypt large datasets, and RSA to encrypt the AES "session" key used for the encryption justified?

[3] You can do this using the `nextBytes` method from the `java.util.Random` class.