## American University of Beirut

## Faculty of Engineering and Architecture

## Electrical and Computer Engineering Department

## **EECE 455 Cryptography and Network Security**

## Project Report

## Fall 2020-2021

## Project Title: Streaming Ciphers

## Marwan Ghamlouch

## Youssef Jaafar

## Project Summary

For this project we intend to implement two streaming ciphers, both commonly used in streaming media. The first is the A5 encryption scheme popularized in GSM networks and designed to be implemented in hardware (processors were still slow at that time). The other encryption scheme is RC4 which is widely used for its simplicity and ease of implementation.

To start the project just open the EECE\_455.pro file using Qt Creator and click on run (Ctrl+R).

A demo video is available at : <https://mailaub-my.sharepoint.com/:v:/g/personal/ymj06_mail_aub_edu/EXrjX-l3gQ5LiVYU3MeE0wUBg1JIPLZsNAm-nIa2GyF1zw?e=LeiFMm>

## Main Menu

When you first start the program you are greeted with the menu presented in the figure below.

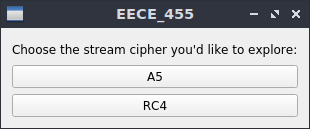


Fig 1: Main Menu when the application starts

The menu offers you which encryption scheme you are interested in trying.

## A5 Scheme

The A5 encryption scheme, like most stream ciphers, relies on creating a stream key to fit the length of the data being encrypted. Given A5 was built for GSM networks and concerned with voice data, it takes in two parameters: a 64-bit *Session Key* provided to both parties by the telecommunications company, and a 22-bit *Frame Number* concerned with data frames being sent.  
It is worth mentioning as well that the A5 scheme generates a 228-bit stream key for every frame, however only 114 of them are meant to encrypt our data, while the remaining 114 bits will be used to decrypt incoming messages.  
After generating the stream key, data is encrypted by a simple XOR operation between the data and the stream key.

When you select the A5 scheme from the main menu, you are greeted with the following window below.

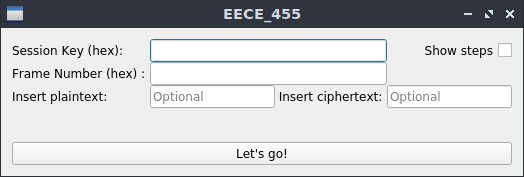
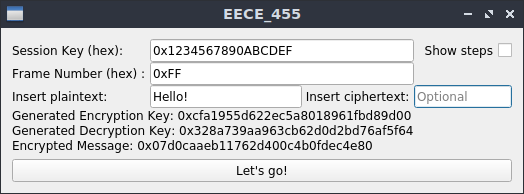
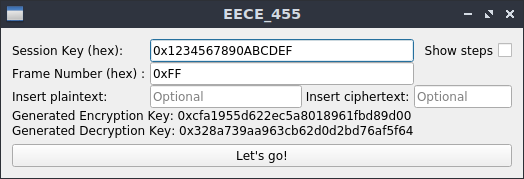


Fig 2: A5 Window ready to receive inputs

The tool allows you to generate the corresponding stream key for a given *Session Key* and *Frame Number,* optionally you can add plaintext to be encrypted or ciphertext to be decrypted.

The Session Key and Frame Number are expected as hex values (the 0x prefix is not necessary). If the provided inputs are shorter than the expected lengths (64-bit for the session and 22-bits for the frame number), the unspecified bits are assumed to be zero.

Fig 3: A5 Window in operation, showing the inputs and expected outputs

In case of an error in the inputs, the application will tell what error its facing as shown below:

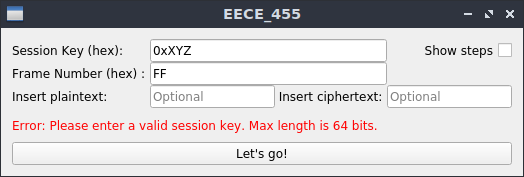
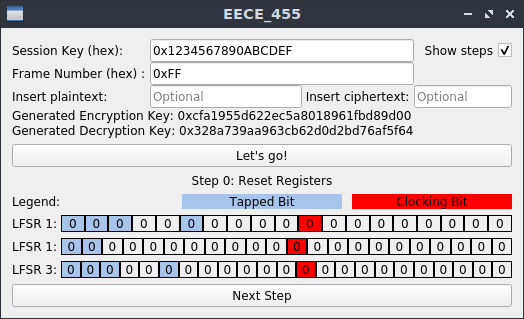


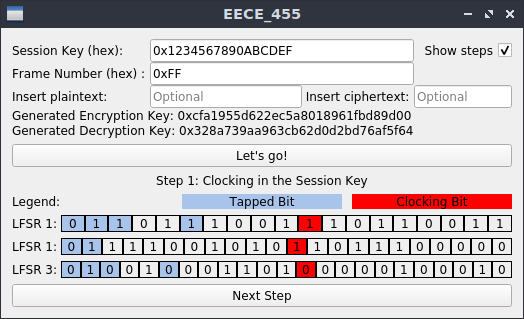
Fig 4: A5 Window showing an error in the session key field. XYZ are not valid hex values

The application also allows you to see intermediate steps to generating the stream key. To do so you need to check the “Show steps” check box. This will show the state of the Linear Feedback Shift Registers (LFSRs) and allow you to step through them at every stage using the “Next Step” button.

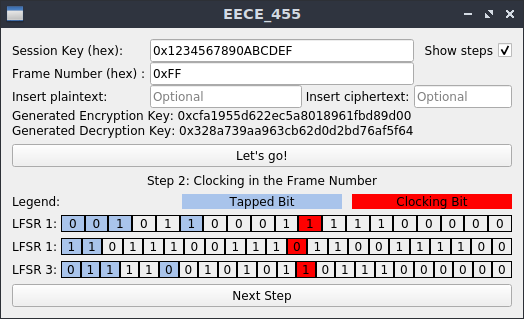


The “zeroth” step is a simple register reset to zero. As the legend mentions, the bits in blue are “tapped” and XORed with each other with the incoming bit (LSB) on each cycle. The bits in red are “Clocking Bits” and determine if a register should be clocked or not, this matters in later steps but not for the first two. If a register’s clocoking bit is part of the majority, the register is clocked, else it is not.

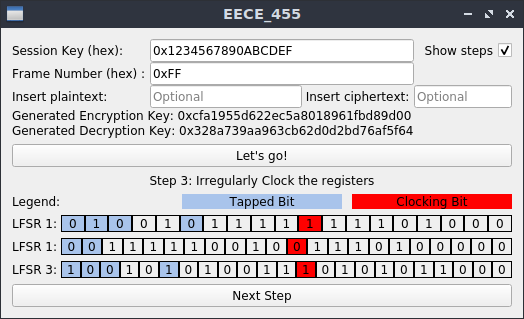
Fig 5: A5 Window showing intermediate steps Fig 6: A5 Window showing the first step



The first step is “Clocking In” the session key into the registers. This operation and all that follow use the tapped bits to determine the new bit that is inserted into the LSB (the rightmost bit in the application).



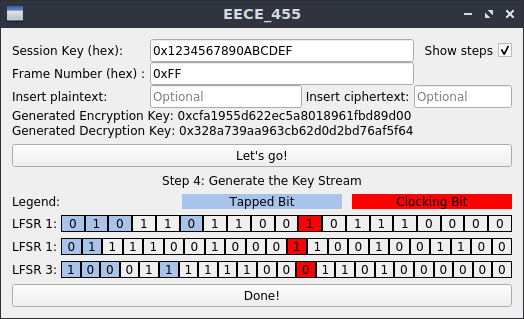
Similar to the previous step, but this time the scheme “clocks in” the frame number.

Fig 7: A5 Window Showing Step 2 Fig 8: A5 Window Showing Step 3

In the third step, we clock 100 cycles but we use the “Clocking Bits” to determine if a register should be clocked on a cycle or not.

As mentioned above, if the clocking bit’s value is from the majority of the three registers’ clocking bits, then the register is clocked.

Lastly we get the key stream. At this stage you can restart the process by clicking “Let’s go!” again, or unchecking the “Show steps” checkbox and using the tool normally.

  
Fig 9: A5 Window Showing the final step of A5 key stream generation

## RC4 Scheme

RC4 was designed by Ron Rivest of RSA Security in 1987. While it is officially termed "Rivest Cipher 4", the RC acronym is alternatively understood to stand for "Ron's Code".

RSA Security has never officially released the algorithm; Rivest has, however, linked to the [English Wikipedia](https://en.wikipedia.org/wiki/English_Wikipedia) article on RC4 in his own course notes in 2008 and confirmed the history of RC4 and its code in a 2014 paper by him.

RC4 generates a pseudorandom stream of bits (a keystream). As with any stream cipher, these can be used for encryption by combining it with the plaintext using bitwise exclusive-or; decryption is performed the same way (since exclusive-or with given data is an involution). This is similar to the one-time pad except that generated *pseudo random bits*, rather than a prepared stream, are used.

To generate the keystream, the cipher makes use of a secret internal state which consists of two parts:

1. A permutation of all 256 possible bytes (denoted "S" below).
2. Two 8-bit index-pointers (denoted "i" and "j").

The permutation is initialized with a variable length key, typically between 40 and 2048 bits, using the *key-scheduling* algorithm (KSA). Once this has been completed, the stream of bits is generated using the pseudo-random generation algorithm (PRGA).

Key Scheduling Algorithm:

The key-scheduling algorithm is used to initialize the permutation in the array "S". "keylength" is defined as the number of bytes in the key and can be in the range 1 ≤ keylength ≤ 256, typically between 5 and 16, corresponding to a key length of 40 – 128 bits. First, the array "S" is initialized to the identity permutation. S is then processed for 256 iterations in a similar way to the main PRGA, but also mixes in bytes of the key at the same time.

Pseudo-random generation algorithm (PRGA)

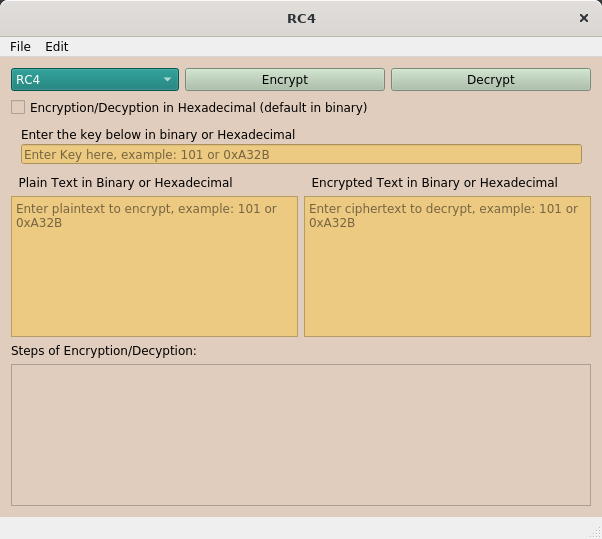
For as many iterations as are needed, the PRGA modifies the state and outputs a byte of the keystream. In each iteration, the PRGA:

* increments *i*
* looks up the *i*th element of S, S[*i*], and adds that to *j*
* exchanges the values of S[*i*] and S[*j*] then uses the sum S[*i*] + S[*j*] (modulo 256) as an index to fetch a third element of S (the keystream value K below)
* then bitwise exclusive ORed (XORed) with the next byte of the message to produce the next byte of either ciphertext or plaintext.

Each element of S is swapped with another element at least once every 256 iterations.

Once you click on the RC4 button in the main window, the RC4 window will show up, and now you can enter either the key and plaintext to encrypt and click on Encrypt, or key and ciphertext to decrypt and click on Decrypt. The steps of the encryption or decryption are also displayed.

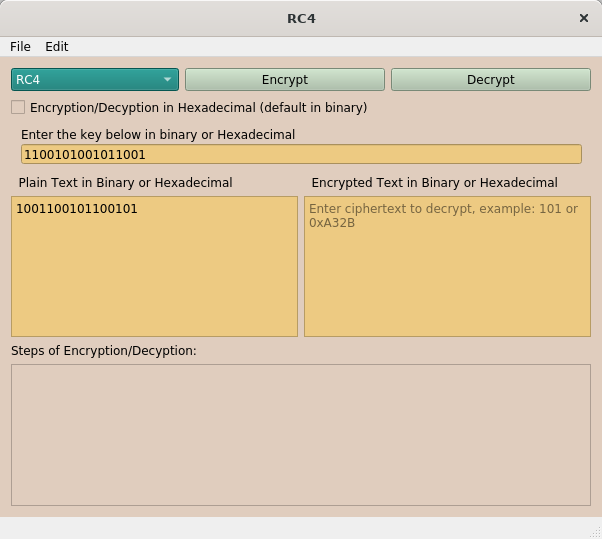
Fig. 10: RC4 Window



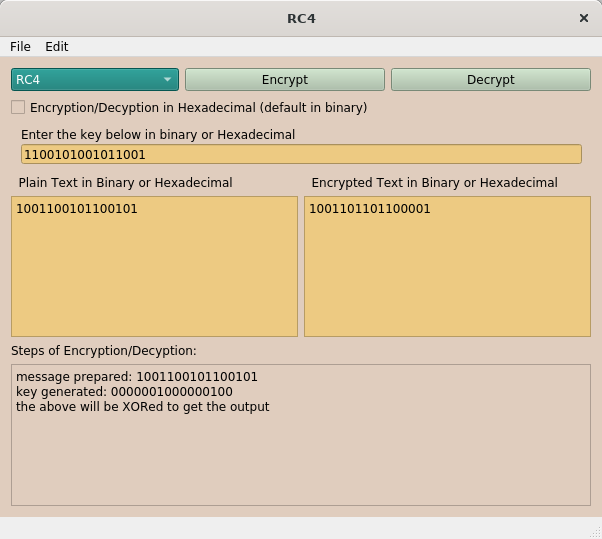
You can enter the key, plaintext or ciphertext in either binary or hexadecimal format. You can also check the checkBox “Encryption/Decryption in Hexadecimal (default in binary)” to get the result of your encryption or decryption in hexadecimal. The default result is in binary.

Below is an example when we enter the key and plaintext in binary format:

Fig. 11: key and plaintext entered in binary

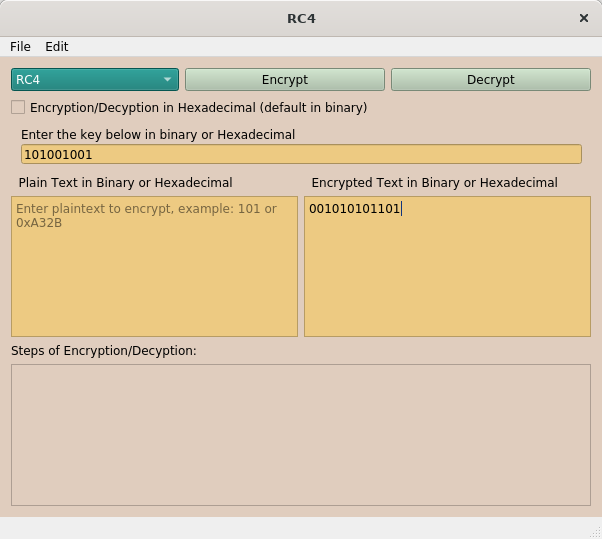


After clicking on Encrypt, the the ciphertext and the steps of encryption will be displayed in binary:

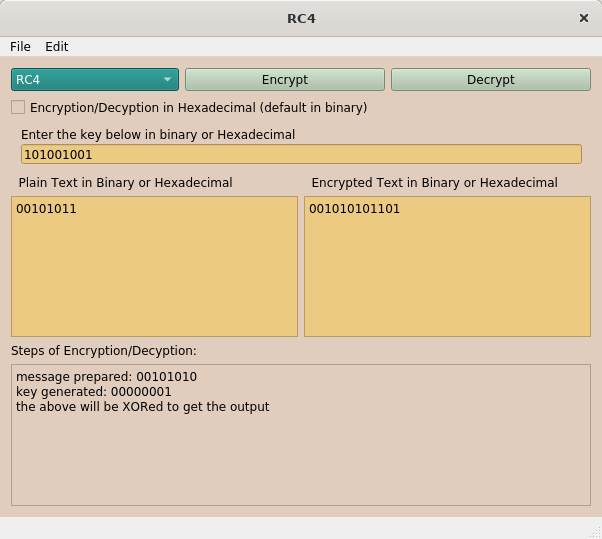
Fig. 12: result of encrypting a plaintext

Let’s take another example where we enter the key and ciphertext in binary:

Fig. 13: Entering key and ciphertext in binary

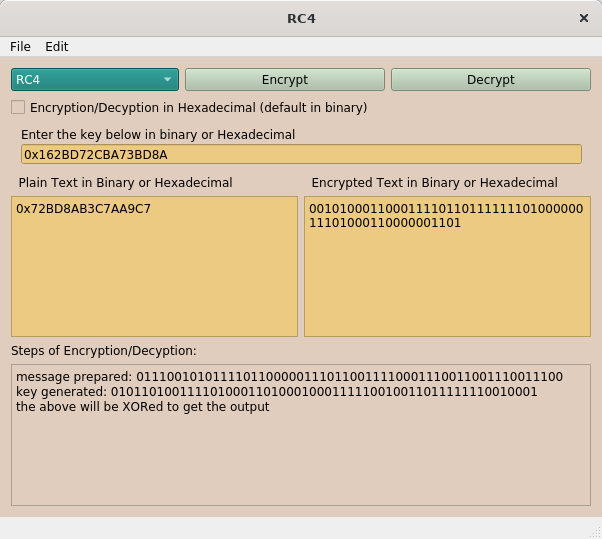


After clicking on decrypt, the results are below:

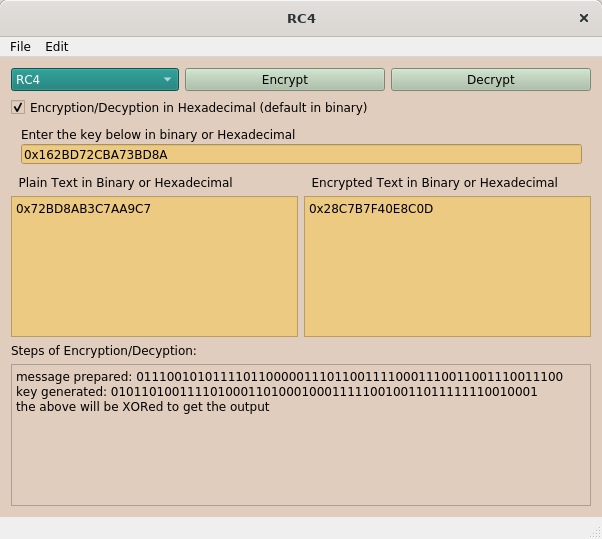
Fig. 14: result of decrypting a ciphertext

We can also enter the values in hexadecimal, but you should add the prefix “0x” before the hexadecimal number. Below is an example:

Fig. 15: entering key and plaintext in hexadecimal



Now what let’s check the box and click on encrypt again, notice how the result is now in hexadecimal:

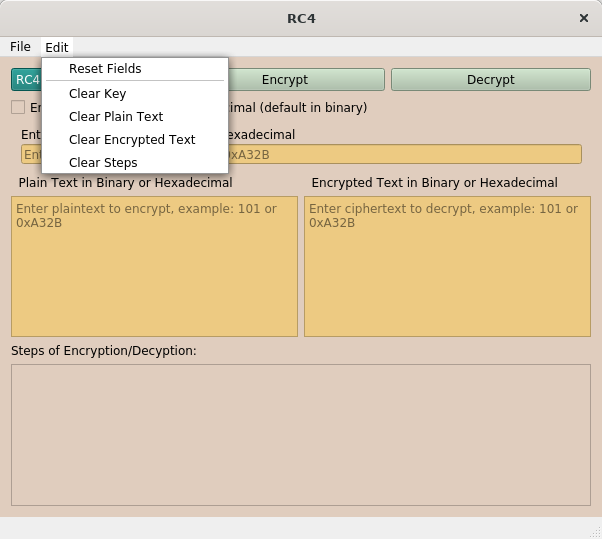
Fig. 16: changing ciphertext to hexadecimal

We can do the same for the decryption.

Now what if we want to clear the key, plaintext, ciphertext or the steps shown?

We can do that!

Fig. 17: Edit menu



To clear the steps of encryption/decryption, you can click on the “Clear Steps” button. You can do the same for the key, plaintext, and ciphertext. You can also clear all fields in one shot by clicking on the “Reset Fields” button.

Let’s say you want to go back to the main screen or exit the application, you can do that in the File menu using “Exit” and “Go Back to Main Menu”:

Fig. 18: File menu