University of Padua
Department of Information Engineering

Information Security Report
Laboratory Session 1

Implementation and linear cryptanalysis of a Feistel cipher

Author:
Luca Badin
Arturo Bellin
Nikhil Karakuchi Chidananda
Simone Favaro
Filippo Giambartolomei
Fawad Ul Haq

Teacher:
Nicola Laurenti

8th November 2020

Solution

Our solution to laboratory 1 is entirely implemented using Python. Specifically, we made use of the `NumPy` library to easily manipulate vectors and quickly compute operations between them. The solution is composed of three Python source files: `main.py` contains the encryptor implementations and attack results, `attack.py` contains functions necessary to carry out the attacks, and `hexutils.py` contains two functions for the string to binary `NumPy` vector conversion and vice versa.

Task 1

We implemented all of the Feistel encryptors using the same function `encrypt()`, which takes different input parameters and functions based on the cipher type (*linear*, *nearly linear* and *non linear*): the plaintext `u`, the initial key `k`, the number of rounds `r`, the message length `l` and the round function `f`.
In fact, even if those may differ for each implementation, the way in which the three transformations are computed is always the same: *substitution*, then *linear tf* and finally *transposition*.
With regards to the subkey generation and the round functions, `key_gen` takes as input the initial key `k` and the specific round `i` in order to output the subkey for that round, while there are three different round functions, one for each Feistel encryptor.

To implement the linear Feistel encryptor required by Task 1, we simply used the `encrypt` function with the linear round function `lin_f`, which uses the round subkey `k`, the round ordinality `i` and the input `y` to compute the so-called value `w`.

Task 2

Since in Feistel ciphers the encryptor and decryptor are identical, but for the order in which the subkeys are used inside the round functions, we made a `decrypt` function which works with the same parameters used in `encrypt`, except for swapping plaintext `u` with ciphertext `x`, where the order of the subkeys order used for the round function computations is reversed. In this way we preserved the `f_lin` function, avoiding the rewrite of a nearly identical piece of code.

Task 3

In order to find the two matrices A and B which identify the linear relationship for the linear Feistel cipher, we created the `find_mat` function, which uses the encryption function `encrypt`, the round function `f` and the message length `l`. The way it works is just an implementation of the methodology presented in *Appendix 1* of the lab instructions, using `numpy` arrays.

Task 4

The linear cryptanalysis KPA against the linear Feistel cipher is done using the `find_key_kpa` function, which takes as input the two matrices we found in Task 3 `a`, `b` and the plaintext-ciphertext pair `u`, `x`. The function uses `numpy` arrays and `numpy` operations in order to quickly compute $k = A^{-1}(x + Bu)$. Moreover, the computation of the inverse matrix A^{-1} follows the methodology presented in *Appendix 2* of the lab instructions.

To carry out the cryptanalysis on the five `(u, x)` pairs provided to us in the file `KPApairsVancouver_linear.hex`, we firstly used the `find_mat` function to find the A and B matrices (which must be the same for all five pairs, since the encryption method is the same). They are:

$$A = \left[\begin{array}{cccccccccccccccccccccccccccccccc}
0&1&0&1&0&1&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&0&0&1&0&1&0 \\
1&0&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&0&0 \\
0&0&0&0&1&0&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0 \\
1&0&0&0&0&0&0&0&1&0&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&1&0&1&1&0&1&0 \\
1&0&1&0&1&0&0&0&0&0&0&0&1&0&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&1&0&1 \\
0&1&0&1&1&0&1&0&1&0&0&0&0&0&0&0&1&0&1&0&0&1&0&1&0&1&0&0&0&0&0&1 \\
0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&0&0&1&0&1&0&0&1&0&1&0&1&0&0 \\
0&1&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&0&0&1&0&1&0&0&1&0&1 \\
0&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&0&0&1 \\
0&0&0&1&0&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0 \\
0&0&0&0&0&0&0&1&0&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1 \\
0&1&0&1&0&0&0&0&0&0&0&1&0&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&1&0&1&1 \\
1&0&1&1&0&1&0&1&0&0&0&0&0&0&0&1&0&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0 \\
0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&0&0&1&0&1&0&0&1&0&1&0&1&0&0&0 \\
1&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&0&0&1&0&1&0&0&1&0&1&0 \\
1&0&1&0&1&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&0&0&1&0&1&0&0 \\
0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&0 \\
0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0 \\
0&1&0&0&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&1&0&1&0&1&1&0&1 \\
1&1&0&1&0&1&0&0&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&1&0&1&0 \\
1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0 \\
0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0 \\
1&0&1&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&0&0&1&0&1&0&1&1&0 \\
0&1&1&0&1&0&1&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&0&0&1&0&1 \\
0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0 \\
1&0&0&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&1&0&1&0&1&1&0&1&0 \\
1&0&1&0&1&0&0&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&1&0&1&0&1 \\
0&1&0&1&1&0&1&0&1&0&0&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&1 \\
0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0 \\
0&1&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&0&0&1&0&1&0&1&1&0&1 \\
1&1&0&1&0&1&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&0&0&1&0&1&0 \\
1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&1&0&1&0&1&1&0&1&0&1&0&0&0&0&0&0&0
\end{array}\right]$$

$$
B = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
$$

Then, using `find_key_kpa` function for each pair, we checked the correctness of the guessed key, by observing that it is always the same and it maps each plaintext to the corresponding ciphertext. That is, $\hat{k} = k =$`55D9F224`.

Task 5

With regards to the nearly linear Feistel cipher, we simply used the `encrypt` function with a different round function implementation, called `near_lin_f`. A similar process was followed by `decrypt`.

Task 6

For task 6 we did not manage to find the exact matrices and the correct encryption key, but we created an algorithm that, with enough plaintext/ciphertext pairs, can find them in a much faster way than a brute force attack on the entire key space.
We first noticed that the nearly-linear function was in the form `f` $\wedge$ `x`, where f is the same function used in the linear implementation of the cipher and x is a chain of four $\vee$ operations. `x` assumes value 0 only when all terms of the chain are equal to 0, 1 otherwise. Furthermore the value of `x` is important only when `f` is equal to 1 due to the nature of the and operation.
All considered it is possible to approximate the value of `x` with 1 committing an error only in the specific case where `f` is equal to 1 and all terms of `x` are equal to 0. With this approximation the cipher becomes linear and it is possible to use the same algorithm used in task 3 to find the matrices A and B, assuming C the identity matrix.

Using the same function used in task 4 it is possible to compute a guess for the value of the key used to encrypt each plaintext/ciphertext pair and then test it on all pairs to check if it is the correct one. Unfortunately given the 5 text pairs given we weren't able to find the correct key but the results are encouraging since the hamming distance between the real ciphertext and the ciphertext computed using the guessed key is as low as 3 bits.

Task 7

Similarly to Task 5, the non linear Feistel cipher implementation required just a different round function called `non_lin_f`. Both the `encrypt` and `decrypt` function remained unchanged.

Task 8

The Meet in the middle attack against the concatenation of two non-linear Feistel ciphers was implemented using an approach based on *Appendix 3* of the lab instructions.

The `meet_in_the_middle` function takes as input parameters the cardinalities of random guesses `n1`, `n2` for $\hat{k}'$ and $\hat{k}''$, encryption and decryption functions `enc`, `dec`, the plaintext-ciphertext pair `u`, `x`, the round function `f` and the length of the message `l`.

After generating the k' and k'' random guesses using the `numpy` *random.randomint* function (which creates random binary arrays of length `l`), for each guess $\hat{k'}_i$ we stored inside a list the pairs $(\hat{k'}_i, \hat{x'}_i = E'_{\hat{k'}_i}(u))$. Same thing is done for pairs $(\hat{k''}_i, \hat{u''}_i = D''_{\hat{k''}_i}(x))$. Then we used the `numpy` function *intersect1d* to retrieve all possible matches between the $\hat{x'}_i$ and $\hat{u''}_j$, so that the corresponding keys pairs $\hat{k'}_i$ and $\hat{k''}_j$ are stored in a list and returned by the function.

To get the key pair guess regarding the KPA attack on the five plaintext-ciphertext pairs in the `KPApairsVancouver_non_linear.hex` document, we initially ran the function for only the first `(u, x)` pair, using very high `n1` and `n2` parameters, finding some matches. Since at the end of the day we just needed the single most probable guess, we trimmed all the matches, checking if a certain matching key pair also worked for some of the other four `(u, x)` pairs provided in the document: this was done by checking if $x = E'_{\hat{k}'_i}(E''_{\hat{k}''_j}(u))$.

Clearly, if a key pair worked for each of the five `(u, x)` pairs, it is most likely the correct guess (even though it is still not certain), so when we found a $(\hat{k'}_i, \hat{k''}_j)$ that satisfied that last requirement we assumed that $\hat{k'}_i = k'_i$ and $\hat{k''}_j = k''_j$. In particular, our guess is $\hat{k'}_i = k'_i =$ `30C0` and $\hat{k''}_j = k''_j =$ `564D`.

Since the generation of the key pair is random, it is possible that at the end of the execution the correct one is not found, both when `n1`, `n2` is much lower than the cardinality of the keyspace, and when they are close to it (since we do not generate unique values).
To overcome this issue, we created the `meet_in_the_middle_sequential` function that tests key pairs in sequential order. This function is particularly helpful to test the algorithm and to secure a valid key pair result just running the program once.