A Linear Cryptanalysis of the FEAL-4 Cipher
Cryptography Assignment

Hadrien Bailly
Dublin City University
Master of Science in Computing
hadrien.bailly2@mail.dcu.ie (21266176)

An assignment submitted to Dublin City University, School of Computing for module CA642 Cryptography & Number Theory.

Name: Hadrien BAILLY
Date: November 23, 2021

CONTENTS

I Introduction 1

II Code 2
II-A Methodology 2
II-B Main Code 2
II-C Supporting Code 3

III Equations 4
III-A Feistel Round Keys 4
III-B Whitening Keys 6

IV Conclusion 7
IV-A Results 7
IV-B Discussion 7
IV-C Next Steps 7

V Annexes 8

Assignment

Linear Cryptanalysis of the FEAL-4 Cipher

This assignment will involve implementing a linear cryptanalysis attack on the weak block cipher FEAL-4, as described in the course, to find the six secret sub-keys that have been used.

The linear cryptanalysis attack can be implemented in the programming language of your choice. Your program will have to loop through a lot of different possible values, so it should be reasonably efficient. The source code for the FEAL-4 cipher (from which the six secret sub-keys have been removed) in C and Java is provided below in the files FEAL.c and FEAL.java, so you may wish to make use of this code and implement your attack in one of these languages. An executable version of this code which has the secret key built into it was used to generate the 200 random plaintext/ciphertext pairs which can be found below in the file known.txt.

Your task is to discover as many of the bits as possible of the six 32-bit sub-keys K_0-K_5 used in this cipher. The more bits, the more marks you will get. However, you will get some marks for even finding a few bits of the sub-keys, as this is a very difficult task. You should submit your code along with a written report describing how you went about the cryptanalysis and the results obtained through the Loop page for this module by 10am Monday 29th November.

I. Introduction

The present document will present the linear cryptanalysis attack of the FEAL-4 cipher, using a Java-based program of our own design and exploiting the linear relationships in the round function f. The aim is to retrieve as many bits from the 6 secret keys.

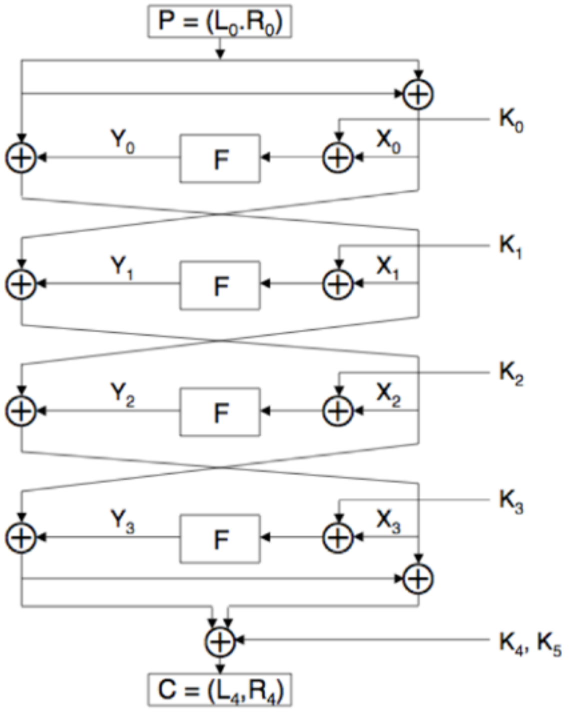


Figure 1. Feal-4 Network

In the next sections, we will first give an overview of the code of our attack, and briefly present the classes and algorithms used. Then, we will describe in further length the equations used to infer the bit values of the different round keys. Finally, we will present and discuss the results of our attack.

II. Code

First and foremost, we chose to use Java as our programming language over C. C might have been a more logical choice, since it is lower-level, closer to bit manipulation, and intrinsically faster than Java, but we favoured the latter nonetheless. The reasons are that we felt more comfortable with this language, having previous experience with it, and would benefit from a series of well-known libraries to speed up development (Maven, Lombok, Logback, Assert4J, Apache Commons to name a few).

A. Methodology

Java offers basic functionalities regarding bit manipulation (and, or, xor, bit shifting, ...) but lacks of a more comprehensive approach, so our first action consisted in establishing wrapper classes to manipulate binary and hexadecimal values, and utilities to encapsulate the conversions and bit manipulation.

Having done this, we worked to load and parse the list of known pairs plaintext-ciphertext into our wrapper classes. We chose to use the CSV format for input and output, as it is a simple and well-recognized flat format. We used the OpenCSV library to effectively read and write, as it presents built-in capabilities of conversion on both operations.

With the list of pairs loaded into memory, we were able to start advancing on the core of the assignment: breaking FEAL-4. Our first reflex was to apprehend the algorithm of encryption and decryption by making it our own, and we devised classes that reproduced the X's and Y's calculations, and a class to represent the keys used in these calculations (`Keychain`).

Moving on from this, we reasoned on the equations and linear relationships (as described in the next section), and went on to encapsulating the equations systems into successive classes: attacking K0 first, then K1, and so on. Ultimately, we were able to find key sets that allowed us to reproduce the known pairs (not without intense mind puzzling, that is).

In a final step, we devised a mechanism to analyse the keys found and export conclusions for further investigation.

The steps described here above are moulded into the tree structure of our code. Each step possesses its own package, and iterations are denoted by indexes in the name of the classes. For the purpose of this report, we will split the packages into two categories: the *main* code, whose functionality directly relates to FEAL, and the *supporting* code which provides generic utilities.

Let's now have a closer look at each of the packages, by category.

B. Main Code

As stated before, the *main code* category focuses on providing the core functionality to the cryptanalysis of FEAL-4. It contains three packages:

- `analysis`
- `equations`
- `feal`

1) `analysis`:

The *analysis* package is the orchestrating package: it controls the program flow, from reading the known pairs into memory, processing all the key rounds and reviewing the results obtained. It contains a specific class for each round.

For the first 4 rounds, these classes are structured in the following fashion:

1) A pre-round step called `Prime`, which compacts the key space by leveraging on the *squeezing* property of the f function. It uses two combined equations to exclude keys based on the value of the bits 15 and 23 and returns squeezed, prime keys.
2) An unfolding step called `Expand`, which opens up the prime keys into their full-scale equivalents, then apply a second filter to retain only the keys that actually match bits elsewhere in the pairs. It usually produces a high number of candidates.
3) A closing step called `Reduce`, which studies the full-scale candidate retained under the light of the remaining linear relationships and further filters these candidates. The number of keys returned here is very small in comparison to the previous step.

Each round is based on an exhaustive search for the examined key. Under normal condition, it would have to verify the constancy of the linear equation for the 200 pairs for a possible key space of 4 bytes ($200 \times 2^8 \times 4$ bytes $= 200 \times 2^{32}$). This repeated for each of the six round keys would amount for $\left(2^{32}\right)^6$ iterations, which is simply unbearable even for nowadays personal computer. To address this issue, the pre-round step makes full use of the *squeezing* property of the f function. This property derives from the middle bits of the output of the f function: these are directly influenced by a composite *XOR* of each half of the input. This means that a key $K = [b_1, b_2, b_3, b_4]$ will have equivalent effect to a key $K' = [0, b_1 \oplus b_2, b_3 \oplus b_4, 0]$. Since the space of K is 2^{32} and K' 2^{16}, using the latter critically reduces the space to search exhaustively. This operation could have been even further reduced if we leveraged from the fact that the first two bits of the input are always irrelevant to the bits 15 and 23. However, the gain would have only been 4×200 operations, which is mostly insignificant while it would have increased the complexity of our code.

Having exhaustively searched K' for a given round, we needed to convert this key back to its actual value, an operation we called *expansion*. The expansion consists of a loop that consider all the *XOR* possibilities for a given key K'. The calculation is simple: since we know the result of the *XOR* operation, we simply guess the values of b_1 and b_4 exhaustively, obtaining by ricochet the values of $b_2 = b_1 \oplus b'_2$ and $b_3 = b_4 \oplus b'_3$. We immediately test if one of the linear equation still holds with the reconstructed keys (max. 200 evaluations). In worst-case scenario, this loop thus have a complexity of $200 \times 2^8 \times 2^8 = 200 \times 2^{16}$, and our current round $200^2 \times 2^{17}$.

With the full-scale candidates, we finally try the remaining linear equations to confirm the keys. These operations require testing each key against at most 3×200 evaluations (at least 2). This makes this round be, at worst, $3 \times 200 \times 2^{17}$ long, and our round $3 \times 200^3 \times 2^{18}$.

Now, even though our round can possibly be amounting to $3 \times 200^3 \times 2^{18} \Leftrightarrow 3 \times 25^3 \times 2^{24}$ evaluations (a relatively large number), it is expected that most keys will be filtered in the pre-round and unfolding steps. The growth in number of operations will thus espouse a logarithmic shape instead of exponential.

For the last two rounds, the process is slightly different since the f function is not involved. Instead, we consider a simple equation to filter results to only those able to produce a constant value for the round key K. We called this operation *comparison*. The resulting classes are by consequence of much smaller size.

2) `equations`: The *equation* package is the place where the cryptanalysis is rooted. Each concrete class in this package represents one of the blueprint equations (cf. section III) used to evaluate the candidate keys, and implements the `Equation` interface.

The `Equation` interface embodies the behaviour of an equation: for a given pair plaintext-ciphertext and a set of key, it produces an evaluation (either *true* or *false*, wrapped in an Integer). An equation is *constant* if, for a collection of pairs and a given key set, it always returns the same evaluation: that is, the unknown constant A is effectively constant.

In addition to the interface, we have designed an abstract class `Relationship` that describes a linear relationship between input and output bits to a round function f. Since only the first four rounds involve the round function, only the first four concrete classes extend the relationship class.

A concrete class follows the below pattern: it always has a method called `apply` inherited from the `Equation` interface (and its parent interface `BiFunction`), which is used to compute the result of the evaluation. When the equation involves a `Relationship`, it also has methods that return terms of the equation, for the sake of division of responsibility.

3) `feal`: The *feal* package contains the source code provided for the realization of this assignment, as well as an implementation of the encryption/decryption process adapted to our libraries. The classes `Encryption` and `Decryption` simply reproduce the terms presented in the revised version of the FEAL-4 algorithm, as described in the lecture and the handbook (Stamp and Low 2007).

C. Supporting Code

The supporting code regroups all the utilities and wrapper classes that we have devised. The principles that guided us were the division of responsibility and the encapsulation of behaviour. There are three packages:

- `csv`
- `models`
- `utils`

1) `csv`: The *csv* package takes responsibility for handling all matters relating to CSV reading and writing.

- It reads and parses the known pairs of plaintext-ciphertext from hexadecimal strings into 8-byte long values, wrapped in a CsvEntry object.
- It converts and writes the collection of valid key sets (keychains) from integer values into 4-byte binary strings.

We make use of the OpenCSV library because it can automatically perform I/O and conversion operations, by the means of annotations and converter class extensions.

2) `models`: The *models* package is responsible for the representation of Binary and Hexadecimal values. Technically speaking, these classes wrap arrays of bytes and expose useful methods to manipulate these values. We use these wrapper classes to provide more human-readable values when manipulating bytes.

3) `utils`: The *utils* package provide utility methods to manipulate bits and bytes, as well as conversion function between the different representations.

- The `BytesUtils` focus on bytes and bits operations: they provide first order logic operators, bits parity calculators, bits extractors, bits masking, and bytes concatenation.
- The `ConversionUtils` focus, as their name suggests, on conversion between the different bytes interpretations (integer, long integer, binary form, hexadecimal form).

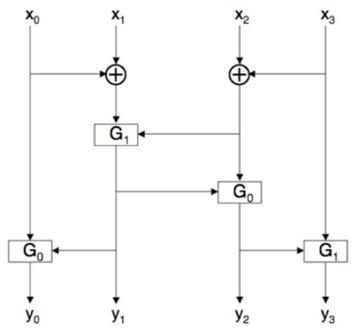
III. Equations

In this section, we will present how we have proceeded to create all the equations used in our $hack$ function. We will first recall briefly the linear relationships at the basis of this attack. Following this introduction, we will explain the methodology that we used to draw equations for the keys involved in the Feistel rounds, and detail the calculations made. Lastly, we will cover the two equations used to break the last keys.

According to the *Applied Cryptography* handbook (Stamp and Low 2007), the following four linear relationships hold in the f function of the FEAL-4 algorithm:

1) $S_{13}(Y) = S_{7,15,23,31}(X) \oplus 1$
2) $S_{5,15}(Y) = S_{7}(X)$
3) $S_{15,21}(Y) = S_{23,31}(X)$
4) $S_{23,29}(Y) = S_{31}(X) \oplus 1$

Figure 2. FEAL-4 f-Function



In order to crack the FEAL-4 encryption, we need to exploit these relationships to break the keys $K0$ to $K3$. Then, using this knowledge, we can simply attack the remaining two keys $K4$ and $K5$ with a meet-in-the-middle approach.

A. Feistel Round Keys

During our analysis of the first keys, we observed that all equations remain strictly identical independently of the linear relationship(s) used, and that the specific term ($\oplus 1$) of linear relationships (1) and (4) is always suppressed in the container unknown constant A (so it will never be relevant in our equations).
This allows us to devise our equations in abstraction of the input and output bits, denoted by β and α respectively, and the specific constants terms: e.g. $S_{23,29}(Y) = S_{31}(X) \oplus 1$ as $S_{\alpha}(Y) = S_{\beta}(X)$.
This forms a **blueprint** equation, in which we can interchange the values of α and β according to the linear relationships observed and which would still holds.

In order to find out the equations relevant to each round key of FEAL-4, we proceeded as follows:

- First, we fixed the blueprint equation, independently of the linear relationships.
- We then wrote two equations aimed at reducing the key space search using the "squeezing" property of the f function on middle bytes.
  - The first equation is from the handbook and combines the linear equations (1) and (3) to isolate bit **15**.

$$S_{5,13,21}(Y) = S_{15}(X) \oplus 1$$

    (we omit details here, as it is explained in full length in the lectures and the handbook (Stamp and Low 2007).)
  - The next equation is of the same shape, and combines the linear equations (3) and (4) to isolate bit **23**.

$$S_{15,21}(Y) \oplus S_{23,31}(Y) = S_{23,31}(X) \oplus S_{31}(X) \oplus 1 \Leftrightarrow S_{15,21,23,31}(Y) \oplus 1 = S_{23}(X)$$

- Finally, we re-created four equations reproducing each linear relationship adapted to the round.

1) K0: $\boxed{= S_\alpha(L_0 \oplus R_0 \oplus L_4) \oplus S_\beta(L_0 \oplus L_4 \oplus R_4) \oplus S_\beta(f(L_0 \oplus R_0 \oplus K_0))}$

$S_\alpha(Y_1) = S_\beta(X_1 \oplus K_1)$	Starting point
$S_\alpha(X_0 \oplus X_2) = S_\beta(X_1 \oplus K_1)$	Expansion of Y_1
$S_\alpha(L_0 \oplus R_0 \oplus Y_3 \oplus L_4 \oplus K_4) = S_\beta(L_0 \oplus Y_0 \oplus K_1)$	Expansion of X_0, X_1, X_2
$S_\alpha(L_0 \oplus R_0 \oplus L_4) \oplus S_\alpha(Y_3) \oplus S_\alpha(K_4)$ $=S_\beta(L_0) \oplus S_\beta(Y_0) \oplus S_\beta(K_1)$	Reorganization
$S_\alpha(L_0 \oplus R_0 \oplus L_4) \oplus S_\alpha(f(X_3 \oplus K_3)) \oplus S_\alpha(K_4)$ $=S_\beta(L_0) \oplus S_\beta(f(X_0 \oplus K_0)) \oplus S_\beta(K_1)$	Expansion of Y_3 and Y_0
$S_\alpha(L_0 \oplus R_0 \oplus L_4) \oplus S_\alpha(f(X_3 \oplus K_3)) \oplus S_\alpha(K_4)$ $=S_\beta(L_0) \oplus S_\beta(f(L_0 \oplus R_0 \oplus K_0)) \oplus S_\beta(K_1)$	Expansion of X_0
$S_\alpha(L_0 \oplus R_0 \oplus L_4) \oplus S_\beta(X_3 \oplus K_3) \oplus S_\alpha(K_4)$ $=S_\beta(L_0) \oplus S_\beta(f(L_0 \oplus R_0 \oplus K_0)) \oplus S_\beta(K_1)$	$S_\alpha(f(X)) \to S_\beta(X)$
$S_\alpha(L_0 \oplus R_0 \oplus L_4) \oplus S_\beta(L_4 \oplus K_4 \oplus R_4 \oplus K_5 \oplus K_3) \oplus S_\alpha(K_4)$ $=S_\beta(L_0) \oplus S_\beta(f(L_0 \oplus R_0 \oplus K_0)) \oplus S_\beta(K_1)$	Expansion of X_3
$S_\alpha(L_0 \oplus R_0 \oplus L_4) \oplus S_\beta(L_4 \oplus R_4) \oplus S_\beta(K_4 \oplus K_5 \oplus K_3) \oplus S_\alpha(K_4)$ $=S_\beta(L_0) \oplus S_\beta(f(L_0 \oplus R_0 \oplus K_0)) \oplus S_\beta(K_1)$	Reorganization
$S_\beta(K_1 \oplus K_3 \oplus K_4 \oplus K_5) \oplus S_\alpha(K_4)$ $=S_\alpha(L_0 \oplus R_0 \oplus L_4) \oplus S_\beta(L_0 \oplus L_4 \oplus R_4) \oplus S_\beta(f(L_0 \oplus R_0 \oplus K_0))$	Isolation of constant A

2) K1: $\boxed{A = S_\alpha(X_1 \oplus L_4 \oplus R_4) \oplus S_\beta(X_0) \oplus S_\beta(f(X_1 \oplus K_1))}$

$S_\alpha(Y_2) = S_\beta(X_2 \oplus K_2)$	Starting Point
$S_\alpha(X_1 \oplus X_3) = S_\beta(X_2 \oplus K_2)$	Expansion of Y_2
$S_\alpha(X_1 \oplus L_4 \oplus K_4 \oplus R_4 \oplus K_5) = S_\beta(X_0 \oplus Y_1 \oplus K_2)$	Expansion of X_2 and X_3
$S_\alpha(X_1 \oplus L_4 \oplus R_4) \oplus S_\alpha(K_4 \oplus K_5)$ $=S_\beta(X_0) \oplus S_\beta(Y_1) \oplus S_\beta(K_2)$	Reorganization
$S_\alpha(X_1 \oplus L_4 \oplus R_4) \oplus S_\alpha(K_4 \oplus K_5)$ $=S_\beta(X_0) \oplus S_\beta(f(X_1 \oplus K_1)) \oplus S_\beta(K_2)$	Expansion of Y_1
$S_\alpha(K_4 \oplus K_5) \oplus S_\beta(K_2)$ $=S_\alpha(X_1 \oplus L_4 \oplus R_4) \oplus S_\beta(X_0) \oplus S_\beta(f(X_1 \oplus K_1))$	Isolation of constant A

3) K2: $\boxed{A = S_\alpha(X_0 \oplus Y_1 \oplus L_4) \oplus S_\beta(X_1) \oplus S_\beta(f(X_2 \oplus K_2))}$

$S_\alpha(Y_3) = S_\beta(X_3 \oplus K_3)$	Starting Point
$S_\alpha(X_2 \oplus L_4 \oplus K_4) = S_\beta(X_3 \oplus K_3)$	Expansion of Y_3
$S_\alpha(X_0 \oplus Y_1 \oplus L_4 \oplus K_4) = S_\beta(X_1 \oplus Y_2 \oplus K_3)$	Expansion of X_2 and X_1
$S_\alpha(X_0 \oplus Y_1 \oplus L_4) \oplus S_\alpha(K_4)$ $=S_\beta(X_1) \oplus S_\beta(Y_2) \oplus S_\beta(K_3)$	Reorganization
$S_\alpha(X_0 \oplus Y_1 \oplus L_4) \oplus S_\alpha(K_4)$ $=S_\beta(X_1) \oplus S_\beta(f(X_2 \oplus K_2)) \oplus S_\beta(K_3)$	Expansion of Y_2
$S_\alpha(K_4) \oplus S_\beta(K_3)$ $=S_\alpha(X_0 \oplus Y_1 \oplus L_4) \oplus S_\beta(X_1) \oplus S_\beta(f(X_2 \oplus K_2))$	Isolation of constant A

4) K3: $\boxed{A = S_\alpha(X_2 \oplus L_4) \oplus S_\beta(X_2 \oplus R_4) \oplus S_\beta(f(X_3 \oplus K_3))}$

$S_\alpha(Y_3) = S_\beta(X_3 \oplus K_3)$	Starting Point
$S_\alpha(X_2 \oplus L_4 \oplus K_4) = S_\beta(X_3 \oplus K_3)$	Expansion of Y_3
$S_\alpha(X_2 \oplus L_4 \oplus K_4) = S_\beta(X_2 \oplus Y_3 \oplus R_4 \oplus K_5 \oplus K_3)$	Expansion of X_3
$S_\alpha(X_2 \oplus L_4) \oplus S_\alpha(K_4)$ $=S_\beta(X_2 \oplus R_4 \oplus K_3) \oplus S_\beta(Y_3) \oplus S_\beta(K_5)$	Reorganization
$S_\alpha(X_2 \oplus L_4) \oplus S_\alpha(K_4)$ $=S_\beta(X_2 \oplus R_4 \oplus K_3) \oplus S_\beta(f(X_3 \oplus K_3)) \oplus S_\beta(K_5)$	Expansion of Y_3
$S_\alpha(K_4) \oplus S_\beta(K_5 \oplus K_3)$ $=S_\alpha(X_2 \oplus L_4) \oplus S_\beta(X_2 \oplus R_4) \oplus S_\beta(f(X_3 \oplus K_3))$	Isolation of constant A

The equation used to infer the value of K_3 has a particularity, in the sense that the value of K_3 is utilized twice: once in the constant value A, and another time in the f function.
While this may seem odd at first, it is not problematic in itself: the key K_3 is expected to be constant, so if the equation should remain constant no matter how many bits from this key are included on its left or right member. It may be ”*always true, or always false*” depending on the bits included, but never ”*sometimes true, sometimes false*” by definition.
In addition, inferring both $K3$ and $K'3$ simultaneously is acceptable, since they are equivalent when input to the round function f.

B. Whitening Keys

The equations used to deduce the values of K_4 and K_5 are different from the previous equations, as they do not involve the round function f. However, the principle of constancy of the keys remain at the core: for all pairs plaintext-ciphertext, the remaining key must remain of same value *ceteris paribus*.

1) K4: $L_4 = X_2 \oplus Y_3 \oplus K_4 \Leftrightarrow \boxed{K_4 = X_2 \oplus Y_3 \oplus L_4}$

2) K5: $R_4 = L_4 \oplus K_4 \oplus X_3 \oplus K_5 \Leftrightarrow \boxed{K_5 = X_3 \oplus L_4 \oplus R_4 \oplus K_4}$

IV. Conclusion

A. Results

Under normal conditions and given the following hardware configuration, our program needs less than a minute to complete, and less than $\pm 2^{26}$ operations.

- **OS** : Pop!_OS 21.04
- **CPU** : 11th Gen Intel® Core™ i5-1135G7 @ 2.40GHz × 8
- **RAM** : 8 GiB

```
2021-11-16 18:59:59,477 - START
2021-11-16 19:00:38,578 - END
                  39,101s
Number of operations: [52526548]
```

For the given 200 pairs of known plaintext-ciphertext, it finds a total number of 192 possible distinct key sets. These key sets can be used interchangeably to encrypt the given plaintexts and produce the same corresponding ciphertexts. We were able to draw the following conclusions:

Figure 3. Known bits' position by key

```
K0 = [01111111 01111111 01111111 01111111]
K1 = [01111101 01111111 01111111 01111101]
K2 = [01111101 01111111 01111111 01111101]
K3 = [01111101 01111111 01111111 01111101]
K4 = [11111101 11111111 11111111 11111101]
K5 = [11111101 11111111 11111111 11111101]
```

Figure 4. Known bits' value by key

```
K0 = [X0101110 X0001111 X1110111 X1111110]
K1 = [X01100X1 X1101100 X0010000 001110X0]
K2 = [X11111X1 X0110111 X0010001 000110X1]
K3 = [X00111X1 X1111101 X1101110 000011X1]
K4 = [100111X1 01101111 01110101 001111X1]
K5 = [111110X0 01100110 00110100 111100X1]
```

B. Discussion

How can several, and distinct, key sets produce yet the same ciphertext for a given plaintext is an open question. By observing the remaining unknown bits, we presume that it might be caused either by

- a **particular condition in the known pairs of plaintext-ciphertext**, which does not allow us to observe a subset of the possible permutations and exclude further keys.
  To validate this supposition, we need to gain access to more pairs and see if it effectively excludes more keys, or study the relations between the known 200 pairs and see if we can deduce a pattern.

or

- a **side effect of the round function**, which causes the first bit of each byte to be irrelevant to the result, and which is propagated to the second to last bit of the next key by way of the cyclic shift of 2.
  To confirm this suspicion, a deeper analysis of the f function is required.

These are only hypothesizes yet to be properly investigated.

C. Next Steps

In order to further enhance the performance of the program, provided that an explanation is found for the presence of the concurrent keys, we could rewrite the cracking process to use a backtracking algorithm.

Currently, the process explores the entire key space and identifies all the possible keys, where the actual decryption process only needs a single key. The backtracking algorithm could instead attempt to solve each keychain individually, moving on to the next one if it is found impossible, and stop whenever a solution is found.

References

Stamp, Mark and Richard M. Low (Apr. 2007). *Applied Cryptanalysis*. John Wiley & Sons, Inc. DOI: 10.1002/9780470148778. URL: https://doi.org/10.1002/9780470148778.

V. Annexes

Logs

Listing 1. analysis.log

```
o.d.s.s.c.a.Analysis:50 - Running new analysis...
o.d.s.s.c.c.CsvHelper:38 - Loading plaintext-ciphertext pairs from
        [/home/hba/Documents/GitHub/DCU/Cryptography/Assignment/code/dictionary.csv]
o.d.s.s.c.c.CsvHelper:54 - Dictionary loaded.
o.d.s.s.c.a.Analysis:67 - Running key hack...
o.d.s.s.c.a.K0:26 - Attacking K0...
o.d.s.s.c.a.K0:94 - Processing K'0...
o.d.s.s.c.a.K0:29 - Found 2 K'0 key candidates.
o.d.s.s.c.a.K0:45 - Expanding possible K'0...
o.d.s.s.c.a.K0:82 - Successfully expanded K'0 [00000000 00100001 10001001 00000000]
o.d.s.s.c.a.K0:32 - Found 1020 K0 key candidates.
o.d.s.s.c.a.K0:115 - Validating retained K0...
o.d.s.s.c.a.K0:35 - Found 16 K0 confirmed key candidates.
o.d.s.s.c.a.K1:29 - Attacking K1...
o.d.s.s.c.a.K1:100 - Processing K'1...
o.d.s.s.c.a.K1:32 - Found 24 K'1 key candidates.
o.d.s.s.c.a.K1:49 - Expanding possible K'1...
o.d.s.s.c.a.K1:87 - Successfully expanded K'1 [00000000 01011101 00101000 00000000]
o.d.s.s.c.a.K1:87 - Successfully expanded K'1 [00000000 01011101 00101010 00000000]
o.d.s.s.c.a.K1:87 - Successfully expanded K'1 [00000000 01011111 00101000 00000000]
o.d.s.s.c.a.K1:87 - Successfully expanded K'1 [00000000 01011111 00101010 00000000]
o.d.s.s.c.a.K1:35 - Found 4080 K1 key candidates.
o.d.s.s.c.a.K1:128 - Validating retained K1...
o.d.s.s.c.a.K1:38 - Found 64 K1 confirmed key candidates.
o.d.s.s.c.a.K2:29 - Attacking K2...
o.d.s.s.c.a.K2:100 - Processing K'2...
o.d.s.s.c.a.K2:32 - Found 32 K'2 key candidates.
o.d.s.s.c.a.K2:49 - Expanding possible K'2...
o.d.s.s.c.a.K2:87 - Successfully expanded K'2 [00000000 01001010 10001010 00000000]
o.d.s.s.c.a.K2:87 - Successfully expanded K'2 [00000000 01001010 10001000 00000000]
o.d.s.s.c.a.K2:87 - Successfully expanded K'2 [00000000 01001000 10001010 00000000]
o.d.s.s.c.a.K2:87 - Successfully expanded K'2 [00000000 01001000 10001000 00000000]
o.d.s.s.c.a.K2:87 - Successfully expanded K'2 [00000000 01001010 10001010 00000000]
o.d.s.s.c.a.K2:87 - Successfully expanded K'2 [00000000 01001010 10001000 00000000]
o.d.s.s.c.a.K2:87 - Successfully expanded K'2 [00000000 01001000 10001010 00000000]
o.d.s.s.c.a.K2:87 - Successfully expanded K'2 [00000000 01001000 10001000 00000000]
o.d.s.s.c.a.K2:87 - Successfully expanded K'2 [00000000 01001010 10001010 00000000]
o.d.s.s.c.a.K2:87 - Successfully expanded K'2 [00000000 01001010 10001000 00000000]
o.d.s.s.c.a.K2:87 - Successfully expanded K'2 [00000000 01001000 10001010 00000000]
o.d.s.s.c.a.K2:87 - Successfully expanded K'2 [00000000 01001000 10001000 00000000]
o.d.s.s.c.a.K2:87 - Successfully expanded K'2 [00000000 01001010 10001010 00000000]
o.d.s.s.c.a.K2:87 - Successfully expanded K'2 [00000000 01001010 10001000 00000000]
o.d.s.s.c.a.K2:87 - Successfully expanded K'2 [00000000 01001000 10001010 00000000]
o.d.s.s.c.a.K2:87 - Successfully expanded K'2 [00000000 01001000 10001000 00000000]
o.d.s.s.c.a.K2:35 - Found 16320 K2 key candidates.
o.d.s.s.c.a.K2:128 - Validating retained K2...
o.d.s.s.c.a.K2:38 - Found 224 K2 confirmed key candidates.
o.d.s.s.c.a.K3:29 - Attacking K3...
o.d.s.s.c.a.K3:100 - Processing K'3...
o.d.s.s.c.a.K3:32 - Found 192 K'3 key candidates.
o.d.s.s.c.a.K3:49 - Expanding possible K'3...
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100001 00000000]
```

```
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100000 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100011 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100001 00000000]
o.d.s.s.c.a.K3:87 - Successfully expanded K'3 [00000000 11100010 01100011 00000000]
o.d.s.s.c.a.K3:35 - Found 48960 K3 key candidates.
o.d.s.s.c.a.K3:128 - Validating retained K3...
o.d.s.s.c.a.K3:38 - Found 768 K3 confirmed key candidates.
o.d.s.s.c.a.K4:21 - Attacking K4...
o.d.s.s.c.a.K4:34 - Processing plaintext-ciphertext matching...
o.d.s.s.c.a.K4:24 - Found 192 K4 key candidates.
o.d.s.s.c.a.K5:21 - Attacking K5...
o.d.s.s.c.a.K5:34 - Processing plaintext-ciphertext matching...
o.d.s.s.c.a.K5:24 - Found 192 K5 key candidates.
o.d.s.s.c.a.Analysis:74 - Key hacking completed.
o.d.s.s.c.a.Analysis:75 - Number of operations: [52526548]
o.d.s.s.c.a.Analysis:88 - Checking candidate keys...
o.d.s.s.c.a.Analysis:92 - Check over.
o.d.s.s.c.a.Analysis:102 - Reviewing...
o.d.s.s.c.a.Review:52 - [192] keychains found.
o.d.s.s.c.a.Review:78 - Known bits position:
        K0 = [01111111 01111111 01111111 01111111]
        K1 = [01111101 01111111 01111111 01111101]
        K2 = [01111101 01111111 01111111 01111101]
        K3 = [01111101 01111111 01111111 01111101]
        K4 = [11111101 11111111 11111111 11111101]
        K5 = [11111101 11111111 11111111 11111101]
o.d.s.s.c.a.Review:110 - Known bits values:
        K0 = [00101110 00001111 01110111 01111110]
        K1 = [00110001 01101100 00010000 00111000]
        K2 = [01111101 00110111 00010001 00011001]
        K3 = [00011101 01111101 01101110 00001101]
        K4 = [10011101 01101111 01110101 00111101]
        K5 = [11111000 01100110 00110100 11110001]
o.d.s.s.c.c.CsvHelper:72 - Exporting keychains to csv...
o.d.s.s.c.c.CsvHelper:87 - Keychains exported to
        [/home/hba/Documents/GitHub/DCU/Cryptography/Assignment/code/keychains.csv]
o.d.s.s.c.a.Analysis:56 - Analysis completed.
```