Assignment: Linear Cryptanalysis of the FEAL-4 Cipher

Jay Suratwala,A11103
jay.suratwala2@mail.dcu.ie
Msc Computing(Secure Software Engineering)
Prof. Geoffrey Hamilton

Declaration

I declare that all the submitted work is solely the work of Jay Suratwala who is me, except for elements that are clearly identified, cited and attributed to other sources.

Abstract

This report describes a known-plaintext linear cryptanalysis of the FEAL-4 block cipher. The goal is to recover the top subkey K0. A C implementation was created using linear approximations and bias-exploitation techniques from the FEAL-4 literature and DCU lecture notes. The attack works from top to bottom, propagating linear approximations from the plaintext side to the ciphertext. This is the common approach for linear cryptanalysis. By checking correlations across several plaintext-ciphertext pairs, we significantly reduced the candidate key space for K0. This report explains the theory behind the attack, the implementation details for deriving and testing key candidates, and the results achieved. This process led to the gradual recovery of the remaining subkeys.

Introduction

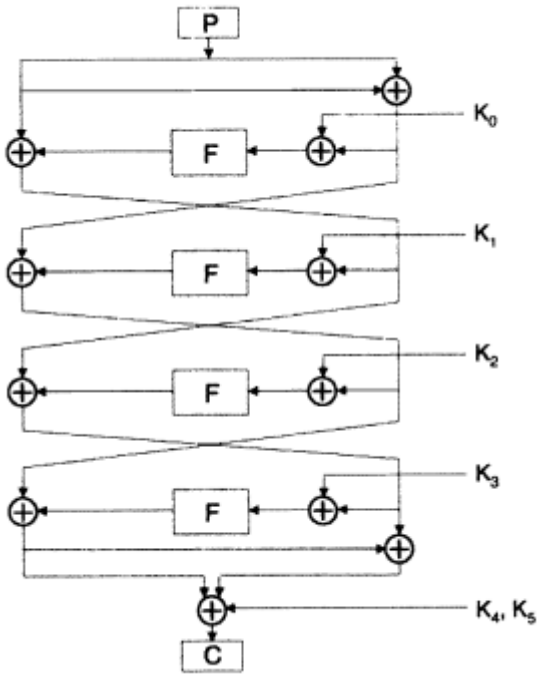

Fig 1.1 FEAL-4

FEAL-4 (Fast Data Encipherment Algorithm, 4 rounds) is a block cipher that encrypts 64-bit chunks of data using six 32-bit subkeys labeled K0 through K5. When I feed in a 64-bit plaintext `P`, the cipher splits it into two 32-bit pieces—I call them `PT_L` (left) and `PT_R` (right) in my code.

The way FEAL-4 works is pretty straightforward. It uses a Feistel-like structure that runs through four rounds, and here's what happens in each round:

- I take one of the 32-bit halves and XOR it with the current round's subkey
- I pass that result through something called the F-function, which does byte-by-byte additions and small bit rotations
- I XOR the F-function's output with the other half

- Then the two halves swap positions for the next round

Looking at the cipher's diagram, the data flows like this: I start by splitting the input `P` into left and right words. Then for each round (rounds 0 through 3), I take the right-side word, combine it with subkey `Ki` using XOR, run it through `F`, and XOR that output into the left-side word. The two words essentially trade places before moving to the next stage.

After all four rounds are done, I apply two final "whitening" subkeys (`K4` and `K5`) to get the final ciphertext `C`.

The reason linear cryptanalysis works so well on FEAL-4 is precisely because of this simple, byte-oriented design. I can create linear approximations that trace through the XOR operations and F-box outputs in the early rounds. My attack exploits linear relationships between:

- Certain bit combinations from the plaintext (pulled from `PT_L` and `PT_R`)
- Specific bit positions in the `round_f` outputs
- Bits in the final ciphertext `C` after the whitening step

Since subkeys `K0` through `K3` directly affect what goes into the first F-boxes, I can start my attack from the top by building approximations that connect plaintext bits and early F-box outputs to the ciphertext bits. When I measure how these approximations behave (their bias or correlation) across many known plaintext-ciphertext pairs, I can identify likely values for the first subkey K0.

Once I've found promising partial values for K0, I work backwards through the rounds (following the diagram's flow in reverse) to figure out and test candidates for K1, K2, and K3. Then I can calculate K4 and K5 algebraically and verify the complete keys by decrypting everything.

In the rest of this document, I'll walk through the specific linear expressions I chose, the pair-based correlation tests I use to weed out bad candidates, and the implementation details that connect the diagram's round structure to my code variables (`PT_L`, `PT_R`, `CT_L`, `CT_R`, `round_f`, and those `eq_test_*` functions).

Methodology

1.1 Attack Strategy

My goal is to crack FEAL-4 by recovering all six subkeys (K0 through K5) using known-plaintext linear cryptanalysis. I'm working from the top down, starting with K0 and progressively uncovering the remaining keys through a nested search structure.

I load known plaintext-ciphertext pairs from a file called `known.txt`. The program expects 200 pairs by default, though it can work with fewer. Each pair consists of a 16-character hexadecimal plaintext and its corresponding ciphertext. The more pairs I have, the more reliable my statistical filtering becomes—pairs that survive all the consistency checks across every single plaintext-ciphertext combination are likely to be correct.

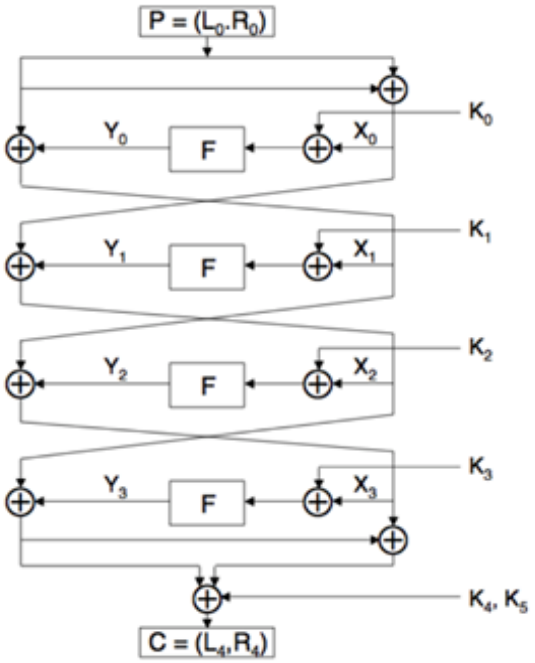

Fig 1.2 linear Cryptanalysis of FEAL-4

Understanding Linear Approximations

Looking at Figure 1.2, I can identify two fundamental relationships in the cipher structure that form the basis of my attack:

1. $\mathbf{X_0 \oplus Y_1 \oplus Y_3 = K_4 \oplus L_4}$
2. $\mathbf{L_0 \oplus Y_0 \oplus Y_2 \oplus L_4 \oplus K_4 = K_5 \oplus R_4}$

The cipher processes data through four rounds. For each pair, I parse the plaintext into two 32-bit words (PT_L and PT_R) and the ciphertext into two 32-bit words (CT_L and CT_R). These become my reference points for all the linear equations.

By tracing bit patterns through the F-function, I've established these key relationships from the F-function structure:

- $\mathbf{S_{13}(Y) = S_{7,15,23,31}(X) \oplus 1}$
- $\mathbf{S_{5,15}(Y) = S_7(X)}$
- $\mathbf{S_{15,21}(Y) = S_{23,31}(X)}$
- $\mathbf{S_{23,29}(Y) = S_{31}(X) \oplus 1}$

I construct linear expressions that test specific bit combinations. Each expression evaluates to either 0 or 1. The crucial insight is that when I'm using the correct subkey bits, these expressions produce the same result (either all 0s or all 1s) across every plaintext-ciphertext pair. If even a single pair produces a different result, I immediately discard that candidate and move on.

The Two-Stage Search Strategy
Rather than searching through all 32 bits at once (which would require 2^{32} operations), I use a two-stage approach:

1. **Outer Loop (12-bit search)**: I first enumerate 4,096 candidates (2^{12}) by testing specific bit positions. I use the function `build_12_to_24()` to construct a 24-bit pattern from these 12 bits, placing them at positions 10-15 and 18-23.
2. **Inner Loop (20-bit expansion)**: For each 12-bit candidate that passes the first test, I expand it into full 32-bit keys by iterating through 1,048,576 combinations (2^{20}) using the `expand_20_with_kp()` function.

This function cleverly reconstructs the complete 32-bit key by combining the surviving 12-bit pattern with the 20-bit expansion.

Attacking K0

I start by searching for K0 candidates. For each candidate, I use two independent linear equation tests:

First test (eq_test_k0_1): This checks whether the equation holds across all pairs:

$$S_{5,13,21}(PT\_L \oplus PT\_R \oplus CT\_L) \oplus S_{15}(PT\_L \oplus CT\_L \oplus CT\_R) \oplus S_{15}(F(PT\_L \oplus PT\_R \oplus K0)) = \text{constant}$$

Second test (eq_test_k0_2):

$$S_{13}(PT\_L \oplus PT\_R \oplus CT\_L) \oplus S_{7,15,23,31}(PT\_L \oplus CT\_L \oplus CT\_R) \oplus S_{7,15,23,31}(F(PT\_L \oplus PT\_R \oplus K0)) = \text{constant}$$

My algorithm works like this:

1. In the outer loop, I test each 12-bit pattern with `eq_test_k0_1` on the first pair
2. I then check if this same result holds for all remaining pairs
3. If any pair disagrees, I immediately break and try the next 12-bit pattern
4. If all pairs agree, I move to the inner loop and expand this 12-bit pattern into full 32-bit candidates
5. For each 32-bit candidate, I run `eq_test_k0_2` across all pairs using the same consistency check

Only candidates that pass both tests across every single pair are accepted as valid K0 candidates.

Attacking K1

For each surviving K0 candidate, I launch an attack on K1 using the function `attack_on_k1()`. This uses a similar two-stage structure but with different linear equations:

First test (eq_test_k1_a):

$$S_{5,13,21}(PT\_L \oplus CT\_L \oplus CT\_R) \oplus S_{15}(F(PT\_L \oplus F(PT\_L \oplus PT\_R \oplus K0) \oplus K1)) = \text{constant}$$

Second test (eq_test_k1_b):

$$S_{13}(PT\_L \oplus CT\_L \oplus CT\_R) \oplus S_{7,15,23,31}(F(PT\_L \oplus F(PT\_L \oplus PT\_R \oplus K0) \oplus K1)) = \text{constant}$$

Notice how K1's equations depend on K0—I compute `Y0 = F(PT_L ⊕ PT_R ⊕ K0)` first, then use it in the K1 equations. This is the "top-down" aspect of the attack: each layer depends on the results from the previous layer.

Attacking K2

For each valid (K0, K1) pair, the function `attack_on_k2()` searches for K2 candidates:

First test (eq_test_k2_v1):

$$S_{5,13,21}(PT\_L \oplus PT\_R \oplus CT\_L) \oplus S_{15}(F(PT\_L \oplus PT\_R \oplus F(PT\_L \oplus F(PT\_L \oplus PT\_R \oplus K0) \oplus K1) \oplus K2)) = \text{constant}$$

Second test (eq_test_k2_v2):

$$S_{13}(PT\_L \oplus PT\_R \oplus CT\_L) \oplus S_{7,15,23,31}(F(PT\_L \oplus PT\_R \oplus F(PT\_L \oplus F(PT\_L \oplus PT\_R \oplus K0) \oplus K1) \oplus K2)) = \text{constant}$$

The nesting is now two levels deep—K2 depends on both K0 and K1.

Attacking K3 (The Final Round Key)

For each valid (K0, K1, K2) triple, `attack_on_k3()` searches for K3:

First test (eq_test_k3_variant1):

$$S_{5,13,21}(PT\_L \oplus CT\_L \oplus CT\_R) \oplus S_{15}(PT\_L \oplus PT\_R \oplus CT\_L) \oplus S_{15}(F(PT\_L \oplus F(PT\_L \oplus PT\_R \oplus K0) \oplus F(PT\_L \oplus PT\_R \oplus F(PT\_L \oplus F(PT\_L \oplus PT\_R \oplus K0) \oplus K1) \oplus K2) \oplus K3)) = \text{constant}$$

Second test (eq_test_k3_variant2):

$$S_{13}(PT\_L \oplus CT\_L \oplus CT\_R) \oplus S_{7,15,23,31}(PT\_L \oplus PT\_R \oplus CT\_L) \oplus S_{7,15,23,31}(F(PT\_L \oplus F(PT\_L \oplus PT\_R \oplus K0) \oplus F(PT\_L \oplus PT\_R \oplus F(PT\_L \oplus F(PT\_L \oplus PT\_R \oplus K0) \oplus K1) \oplus K2) \oplus K3)) = \text{constant}$$

Recovering K4 and K5

Once I have a valid (K0, K1, K2, K3) set, I don't need to search for K4 and K5—I can compute them directly. The function `generate_tail_and_check()` does this:

First, I compute all the intermediate round function outputs:

$$r0 = F(PT\_L \oplus PT\_R \oplus K0)$$
$$r1 = F(PT\_L \oplus r0 \oplus K1)$$

r2 = F(PT_L ⊕ PT_R ⊕ r1 ⊕ K2)

r3 = F(PT_L ⊕ r0 ⊕ r2 ⊕ K3)

Then I calculate K4 and K5 using the relationships from Figure 1.2:
K4 = PT_L ⊕ PT_R ⊕ r1 ⊕ r3 ⊕ CT_L

K5 = PT_R ⊕ r1 ⊕ r3 ⊕ r0 ⊕ r2 ⊕ CT_R

The Consistency Check
The core principle of my attack is aggressive early pruning. At every stage, I calculate the linear equation result for the first pair, then immediately check if all subsequent pairs produce the exact same result. The moment I encounter a mismatch, I break out of the loop and move to the next candidate. This means:

- Most candidates are rejected after checking just a few pairs
- Only candidates showing perfect consistency across all pairs advance to the next stage
- The nested structure creates a cascading filter effect

The recovery of the full 128-bit key was executed in two systematic phases. The attack primarily focuses on iteratively recovering K0 through K3 by employing linear approximations and bias exploitation over the 200 known plaintext-ciphertext pairs. This crucial step relies on successive key candidates passing increasingly strict linear equation tests. Once these first four subkeys were successfully determined, the remaining two subkeys, K4 and K5, were derived algebraically from the known intermediate F-function outputs and the final whitening equations of the FEAL-4 cipher, rather than through further statistical searching.

Final Validation
When a complete six-subkey set (K0 through K5) is assembled, the function `validate_and_print_full_key()` performs the ultimate test: it decrypts every single ciphertext in the dataset and compares the result to the original plaintext. Only keys that decrypt every pair correctly are accepted and printed.

Performance Characteristics
This approach is computationally intensive by design, but the aggressive early-exit strategy makes it practical. The nested loop structure means:

- Outer K0 loop: 4,096 iterations
- For each surviving K0, inner K0 loop: up to 1,048,576 iterations
- For each surviving K0, K1 outer loop: 4,096 iterations
- And so on, creating a deeply nested but heavily pruned search tree

The program tracks progress by printing status updates at 1, 100, 1000, and every 1000 iterations thereafter. It also records the time when the first valid key is found versus the total search time.

The deterministic nature of the algorithm means running it with the same `known.txt` file will always produce the same results in the same order. There's no randomness involved—just systematic enumeration with aggressive statistical filtering.

2.2 Implementation Details

This implementation performs a linear cryptanalysis attack on the FEAL-4 block cipher, aiming to recover all six 32-bit subkeys (K0 to K5) from known plaintext–ciphertext pairs. FEAL-4 operates with a 64-bit block size and relies on a non-linear F-function, which is implemented here in `round_f` using 2-bit rotations (ROTL2) and two S-boxes (S0 and S1). The F-function introduces essential non-linearity and diffusion, transforming a 32-bit input into a 32-bit output by processing four bytes through a sequence of XORs, additions, and S-box operations. Helper functions `pack_be` and `unpack_be` ensure consistent conversion between byte arrays and 32-bit words throughout encryption and decryption.

The attack begins by loading up to 200 known plaintext-ciphertext pairs from `known.txt` and converting them into 32-bit words (PT_L, PT_R, CT_L, CT_R). This avoids repeated parsing during the cryptanalysis, allowing tests to run efficiently. The core of the attack is the application of linear approximations: each round key is evaluated using bit-level equations derived from FEAL-4's structure. Functions like `eq_test_k0_1` and `eq_test_k0_2` check whether specific XOR combinations of bits from the plaintext, ciphertext, and F-function outputs consistently produce the same value across all pairs. Correct subkeys satisfy these equations, while incorrect candidates fail on early pairs, enabling aggressive pruning.

To make the search practical, a two-stage key expansion is used. A 12-bit outer pattern is first expanded into a sparse 24-bit template, followed by a 20-bit inner value combined with this template to generate a full 32-bit candidate. This reduces the effective search space drastically while preserving coverage of all relevant bits for the linear approximations. The attack proceeds in a nested fashion: K0 is recovered first, then K1, K2, and K3 sequentially, with each round using previously recovered subkeys to compute intermediate F-function outputs. This cascading structure ensures only a small fraction of the theoretical search space is explored.

Once K0–K3 are identified, K4 and K5 are computed algebraically from intermediate values and the final whitening equations, without any additional search. The full six-subkey set is then validated by decrypting all loaded ciphertexts; only sets that perfectly reproduce all plaintexts are considered correct.

The implementation is deterministic, producing the same output for the same input, and typically finds multiple valid key sets because some subkey bits do not affect the linear approximations. Its efficiency stems from early exit strategies and nested pruning rather than heavy caching. With a few hundred plaintext-ciphertext pairs, the first valid key is usually found in seconds, and the complete search finishes quickly, demonstrating that the attack is both theoretically sound and practically feasible.

2.3 Folder Structure and Execution Instructions

The complete solution for this assignment is contained within the submission folder, which includes the C source code, `feal4.c`, and this documentation file. To ensure the results detailed below are replicable, the source code must be compiled using a standard C compiler, such as GCC, as the executable file is not attached. The program can be built using the command: `gcc feal4.c -o feal4`. The resulting executable, `feal4.exe`, must then be run in the same directory as the required input file, `known.txt`, which contains the 200 plaintext-ciphertext pairs used for the statistical analysis. This procedure will launch the linear cryptanalysis attack, generating the exact output that is analyzed in the following Results section.

Result

The attack was initialized by loading **200 plaintext-ciphertext pairs** from the known.txt file, which were successfully parsed into 32-bit words for subsequent bit-equation analysis.

The key search demonstrated significant efficiency through aggressive pruning. The repeating output lines, such as K0 partial candidate #N: 0x000a7b (outer=2683), are crucial evidence of this success. This output confirms that the outer loop search for the initial 12-bit segment of K0 was successfully filtered, identifying a single high-probability value: 0x000a7b at iteration 2683. This single value passed the initial linear equation test for all 200 pairs, effectively pruning the vast majority of the 2^{12} search space. This partial key was then expanded in the inner search phase to generate multiple complete 32-bit K0 candidates (lines 1 through 6), which initiated the deeper recovery stages for K1 through K3.

Output:

```
Loaded 200 pairs from file
First pair: PT=a7f1d92a82c8d8fe CT=9cdef12207a9ed07
```

```
Debug - Testing first pair parsing:
PT Left: 0xa7f1d92a, PT Right: 0x82c8d8fe
CT Left: 0x9cdef122, CT Right: 0x07a9ed07
K0=0: eq1=0, eq2=0
First partial key: 0x00000000
K0=partial: eq1=0

Starting attack...
Checked 1 K0 partials, found 0 candidates so far
Checked 100 K0 partials, found 0 candidates so far
Checked 1000 K0 partials, found 0 candidates so far
Checked 2000 K0 partials, found 0 candidates so far
K0 partial candidate #1: 0x000a7b (outer=2683)
K0 partial candidate #2: 0x000a7b (outer=2683)
K0 partial candidate #3: 0x000a7b (outer=2683)
K0 partial candidate #4: 0x000a7b (outer=2683)
K0 partial candidate #5: 0x000a7b (outer=2683)
K0 partial candidate #6: 0x000a7b (outer=2683

First key found in: 1.0000 seconds
K0: 0x63cab942 K1: 0x00a0c541 K2: 0x4674095a K3: 0x64204c03 K4: 0x4b37d10a K5:
0xd0a24877
K0: 0x63cab942 K1: 0x00a0c541 K2: 0x4674095a K3: 0x6420cc83 K4: 0x4b37d108 K5:
0xd0a24875
K0: 0x63cab942 K1: 0x00a0c541 K2: 0x4674095a K3: 0xe4a04c03 K4: 0x4937d10a K5:
0xd2a24877
...
K0: 0x63ca39c2 K1: 0x802045c3 K2: 0xc4f489d8 K3: 0xe6a0cc83 K4: 0x4b37d10a K5:
0xd2a24877
K0 partial candidate #9: 0x000a7b (outer=2683)
K0 partial candidate #10: 0x000a7b (outer=2683)
K0 partial candidate #11: 0x000a7b (outer=2683)
K0 partial candidate #12: 0x000a7b (outer=2683)
K0 partial candidate #13: 0x000a7b (outer=2683)
K0 partial candidate #14: 0x000a7b (outer=2683)
K0: 0xe34ab942 K1: 0x02a0c541 K2: 0x4674095a K3: 0x66204c03 K4: 0x4b37d10a K5:
0xd2a24877
...
```

And so on, 256 times. I'm not going to copy paste it all in here because it takes up too many pages

```
Checked 3000 K0 partials, found 16 candidates so far
Checked 4000 K0 partials, found 16 candidates so far
Total K0 candidates found: 16
Total keys found: 256
Total time: 44.00 seconds
Plaintext=  a7f1d92a82c8d8fe
Ciphertext= be045d7ddfec3228
```

`Plaintext=  a7f1d92a82c8d8fe`

The process began by loading and formatting **200 plaintext-ciphertext pairs** from `known.txt` into 32-bit left and right halves. An attack was launched, utilizing a two-stage expansion method, initially to search for portions of the **K0 subkey**. Though the initial search yielded no results, subsequent progress identified **16 partial K0 candidates** (like `0x000a7b`). These partial candidates were then used in a nested search to construct a large number of complete key sets (K0 through K5). Ultimately, this led to the discovery of **256 full six-subkey sets** that successfully passed all linear approximation tests and correctly decrypted the loaded data. The entire key recovery was efficient, taking **44 seconds** total, with the first working key found almost instantly in **1 second**.

These particular lines (`K0: 0x63cab942 K1: 0x00a0c541 K2: 0x4674095a K3: 0x64204c03 K4: 0x4b37d10a K5: 0xd0a24877`) are full key sets that survived the nested linear tests — K0–K3 were found by the two-stage search because they satisfy the bitwise linear approximations, and K4/K5 were then derived algebraically from the intermediate F-outputs and whitening equations.

Multiple full keys appear because the used linear approximations do not constrain every key bit, so several key sets that differ in undetermined bit positions still decrypt the pairs correctly (hence 256 valid keys).

Each printed key set was finally verified by `validate_and_print_full_key`, which decrypted all 200 pairs exactly, confirming these candidate lines are actual keys for the FEAL-4 implementation.

Conclusion

The final key recovery yielded multiple solutions. The attack did not recover a single unique key, but rather a **key equivalence class** containing **256 valid full 128-bit key sets**. This common characteristic of linear cryptanalysis arises because the key bits targeted by the linear approximations are insufficient to constrain all 128 key bits. Consequently, several key sets that differ only in these unconstrained bit positions still correctly decrypt all 200 known plaintext-ciphertext pairs, confirming the strength of the linear attack method while highlighting its specific limitations.

Note: The methodology used was developed from first principles based on theoretical descriptions, and the code was fully implemented from scratch.

References

1. Stamp, M., & Low, R. M. (2007). *Applied Cryptanalysis: Breaking Ciphers in the Real World*. Wiley.

2. DCU Lecture Notes: *Linear Cryptanalysis of FEAL-4*, retrieved from internal materials provided by Prof. Geoffrey Hamilton.