FEAL CIPHER

AGARWALA PRATHAM – DIFFERENTIAL CRYPTANALYSIS – FEAL4

PARASHAR KSHITIJ - BOOMERANG ATTACK – FEAL6

AGENDA

INTRODUCTION

STRUCTURE OF FEAL CIPHER

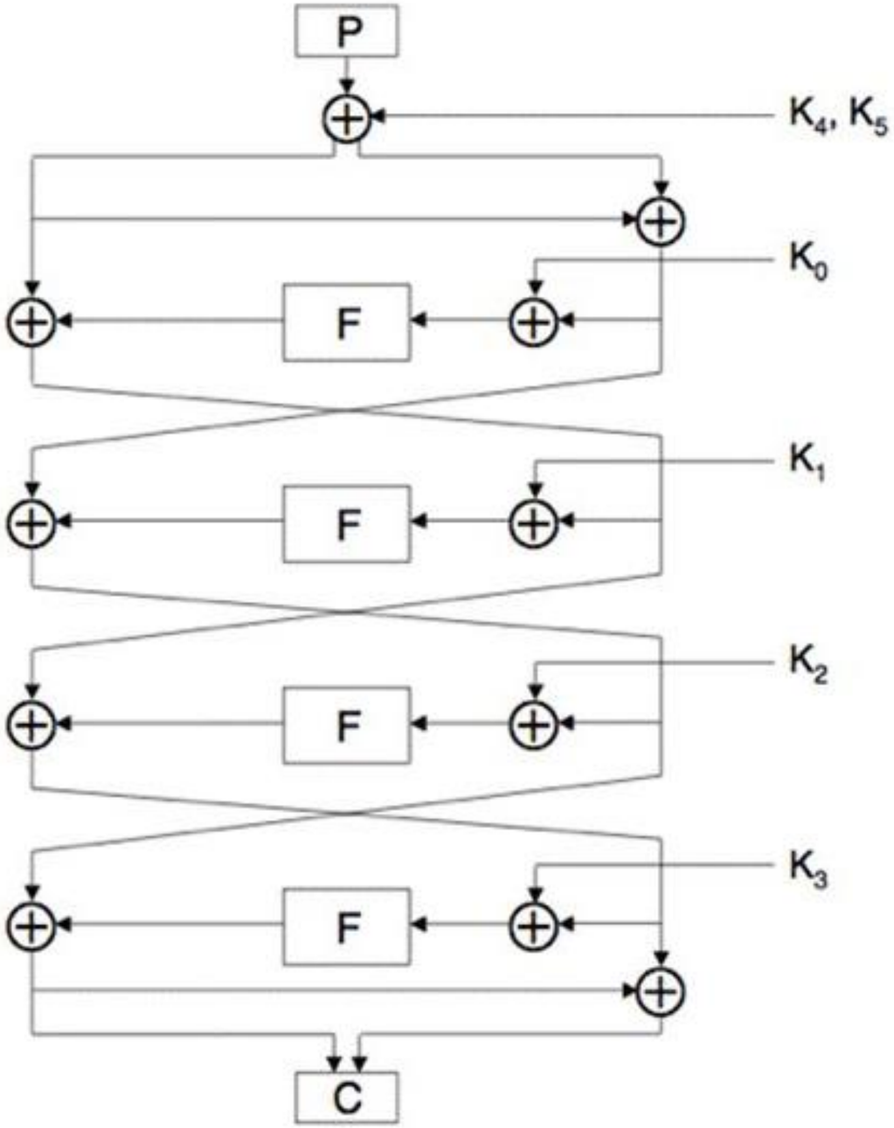
CRYPTANALYSIS

LEGACY

NTTResearch

INTRODUCTION

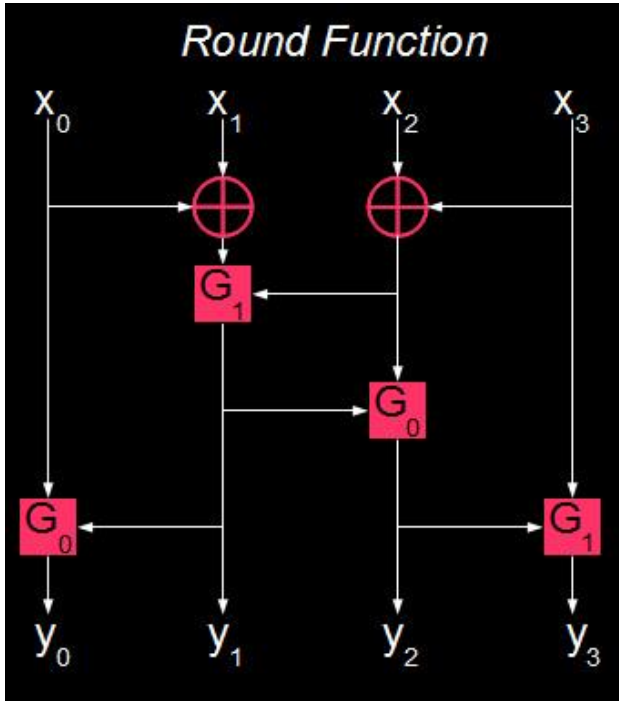
- FEAL (Fast data Encipherment ALgorithm) is a symmetric block cipher developed in 1987 by Akihiro Shimizu and Shoji Miyaguchi of NTT labs in Japan.
- It was designed to be faster and more secure than the DES (Data Encryption Standard) cipher.
- FEAL has a lot of variations
  - ➢ FEAL-4
  - ➢ FEAL-N (n<=4)
  - ➢ FEAL-NX



STRUCTURE OF FEAL-4

- FEAL is a Feistel cipher, hence the plain text splits into left and right halves for encryption.
- First, subkeys 4 and 5 are used to XOR the incoming plaintext.
- The right half is mixed with the round's subkey, then it is run through a one-way function called the round function and combined with the left half using the XOR operation.
- The resulting halves are then swapped, and the process is repeated for n-1 rounds.
- For the final round, output halves do not swap. Instead, they come straight down, and the final round's left output half becomes the final ciphertext's left half. The final ciphertext's right half is produced by XORing that left half with the final round's original right half.
- Three main parts of the cipher are:
  - Round Function (F)
  - G-Box
  - Subkey Generation/Key Schedule

ROUND FUNCTION AND G-BOX



- G-Box is the only non-linear component of the Cipher and is the source of confusion of the cipher

$$G_0(a , b) =((a +b) \text{ mod256})\lll 2$$

$$G_1(a , b) =((a +b +1) \text{ mod256})\lll 2$$

Where "<<<" is left cyclic shift (rotation)

- Round Function is denoted by

$$F (x_0, x_1, x_2, x_3)=(y_0, y_1, y_2, y_3)$$

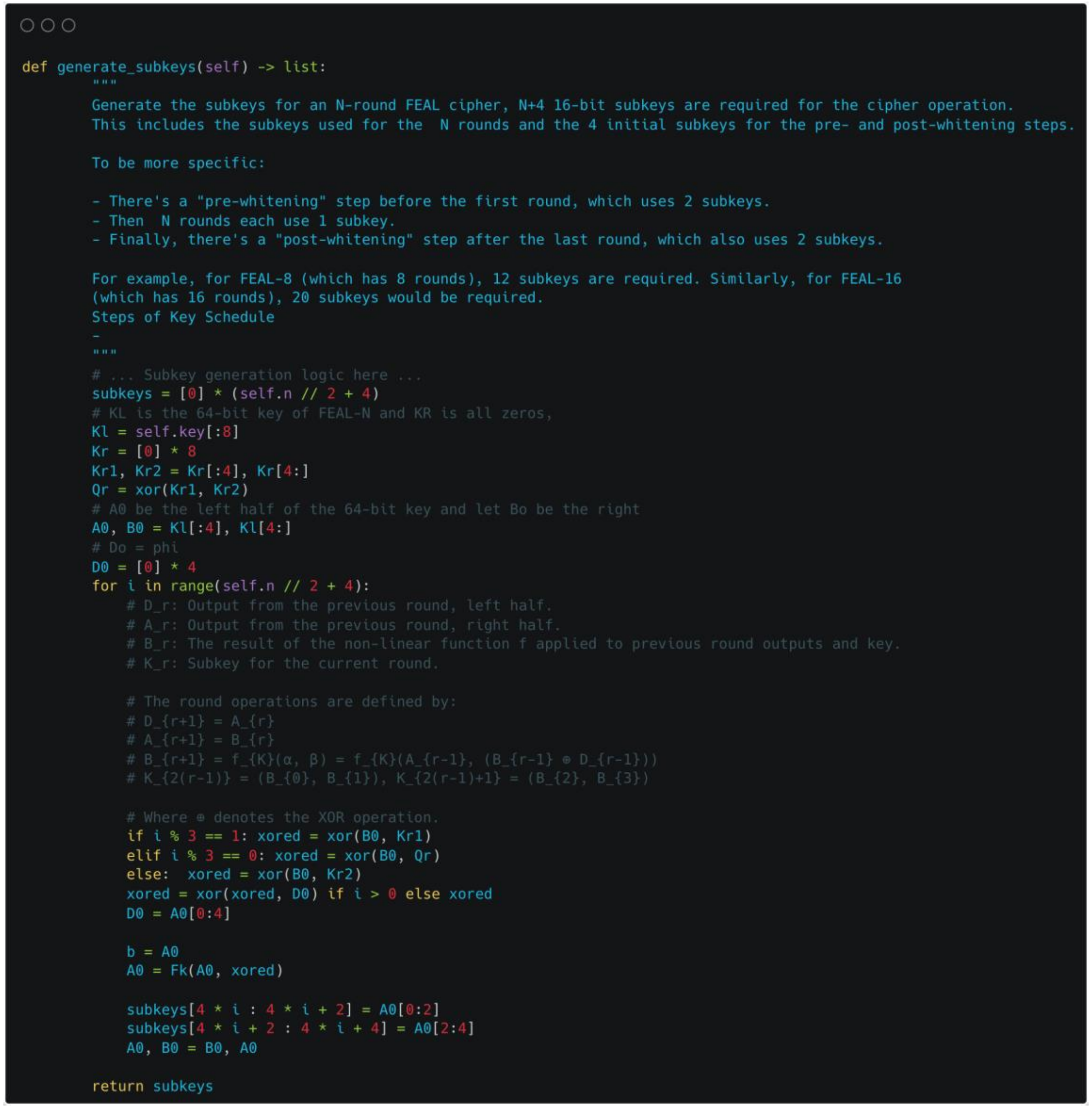
$$Y_1 =G_1 (x_0 \oplus x_1 , x_2 \oplus x_3)$$

$$Y_2 =G_1 (y_1, x_2 \oplus x_3)$$

$$Y_3 =G_0 (x_0, y_1)$$

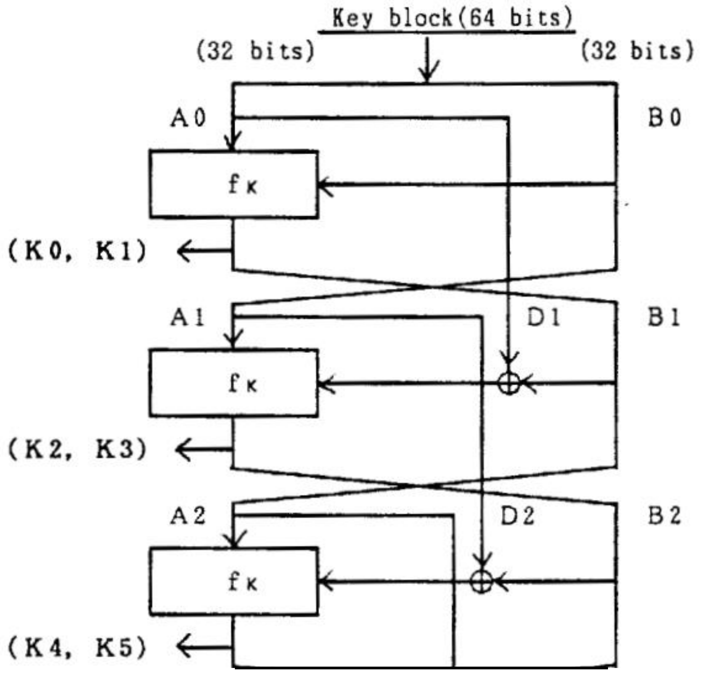
$$Y_4 =G_1 (y_2, y_3)$$

SUBKEY GENERATION



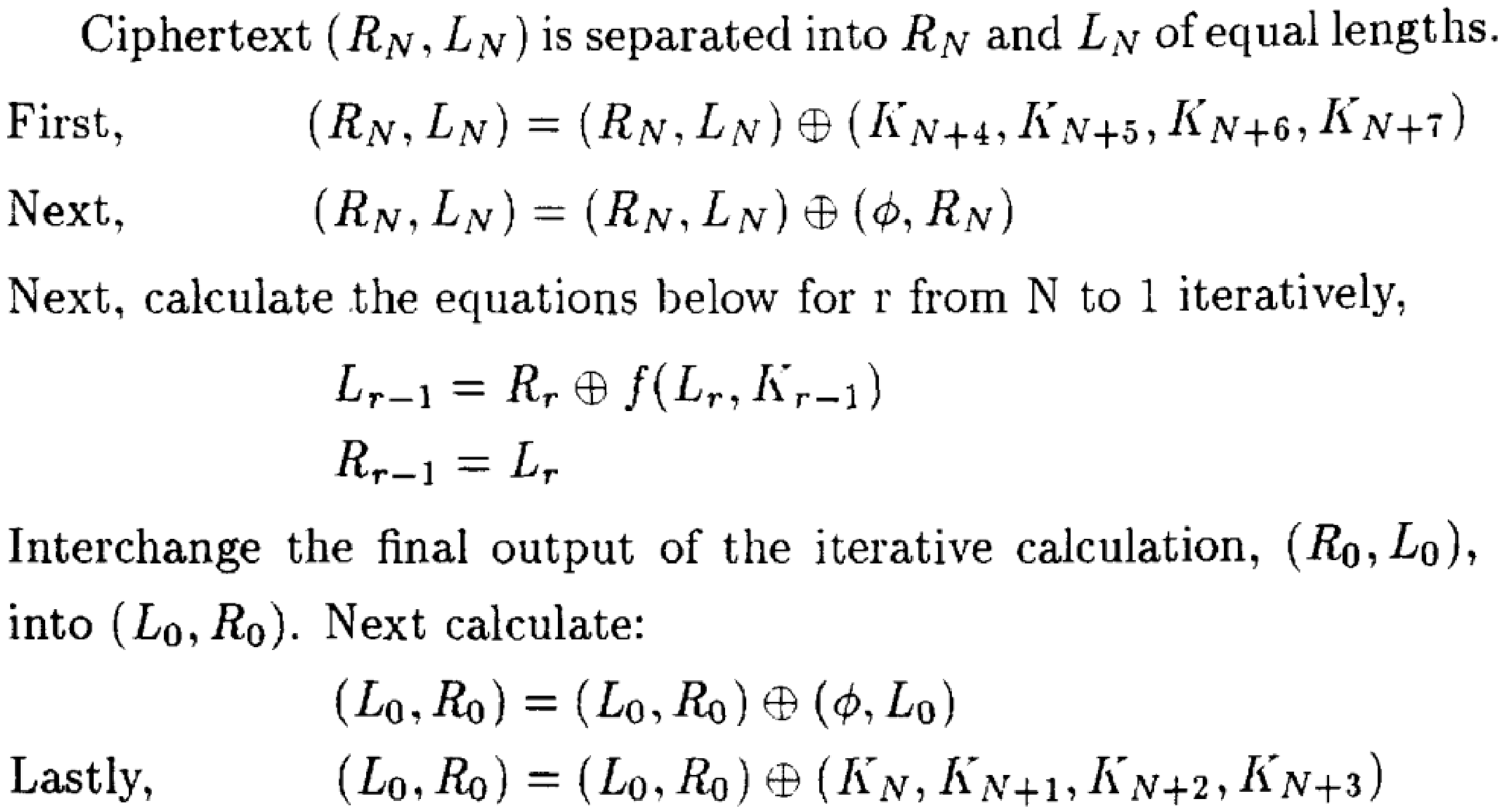
```python
def generate_subkeys(self) -> list:
        """
        Generate the subkeys for an N-round FEAL cipher, N+4 16-bit subkeys are required for the cipher operation.
        This includes the subkeys used for the  N rounds and the 4 initial subkeys for the pre- and post-whitening steps.

        To be more specific:

        - There's a "pre-whitening" step before the first round, which uses 2 subkeys.
        - Then  N rounds each use 1 subkey.
        - Finally, there's a "post-whitening" step after the last round, which also uses 2 subkeys.

        For example, for FEAL-8 (which has 8 rounds), 12 subkeys are required. Similarly, for FEAL-16
        (which has 16 rounds), 20 subkeys would be required.
        Steps of Key Schedule
        -
        """
        # ... Subkey generation logic here ...
        subkeys = [0] * (self.n // 2 + 4)
        # KL is the 64-bit key of FEAL-N and KR is all zeros,
        Kl = self.key[:8]
        Kr = [0] * 8
        Kr1, Kr2 = Kr[:4], Kr[4:]
        Qr = xor(Kr1, Kr2)
        # A0 be the left half of the 64-bit key and let Bo be the right
        A0, B0 = Kl[:4], Kl[4:]
        # Do = phi
        D0 = [0] * 4
        for i in range(self.n // 2 + 4):
            # D_r: Output from the previous round, left half.
            # A_r: Output from the previous round, right half.
            # B_r: The result of the non-linear function f applied to previous round outputs and key.
            # K_r: Subkey for the current round.

            # The round operations are defined by:
            # D_{r+1} = A_{r}
            # A_{r+1} = B_{r}
            # B_{r+1} = f_{K}(α, β) = f_{K}(A_{r-1}, (B_{r-1} ⊕ D_{r-1}))
            # K_{2(r-1)} = (B_{0}, B_{1}), K_{2(r-1)+1} = (B_{2}, B_{3})

            # Where ⊕ denotes the XOR operation.
            if i % 3 == 1: xored = xor(B0, Kr1)
            elif i % 3 == 0: xored = xor(B0, Qr)
            else:  xored = xor(B0, Kr2)
            xored = xor(xored, D0) if i > 0 else xored
            D0 = A0[0:4]

            b = A0
            A0 = Fk(A0, xored)

            subkeys[4 * i : 4 * i + 2] = A0[0:2]
            subkeys[4 * i + 2 : 4 * i + 4] = A0[2:4]
            A0, B0 = B0, A0

        return subkeys
```

- This our implementation of official subkey generation algorithm suggested in the Feal-N and Feal-NX paper.



DECRYPTION

- Plaintext P is separated into L(0) and R(0) of equal lengths (32 bits)
- Note: Φ refers to a 32-bit long zero-block



Ciphertext (R_N, L_N) is separated into R_N and L_N of equal lengths.

First, $(R_N, L_N) = (R_N, L_N) \oplus (K_{N+4}, K_{N+5}, K_{N+6}, K_{N+7})$

Next, $(R_N, L_N) = (R_N, L_N) \oplus (\phi, R_N)$

Next, calculate the equations below for r from N to 1 iteratively,

$$L_{r-1} = R_r \oplus f(L_r, K_{r-1})$$
$$R_{r-1} = L_r$$

Interchange the final output of the iterative calculation, (R_0, L_0), into (L_0, R_0). Next calculate:

$$(L_0, R_0) = (L_0, R_0) \oplus (\phi, L_0)$$

Lastly, $(L_0, R_0) = (L_0, R_0) \oplus (K_N, K_{N+1}, K_{N+2}, K_{N+3})$

CRYPTANALYSIS

- Differential Cryptanalysis of FEAL-4
- Boomerang Attack on FEAL-6

TIMELINE OF ATTACKS

TIMELINE

1987

The FEAL-4 cipher was introduced by Akihiro Shimizu and Shoji Miyaguchi of NTT.

1988

Den Boer published a paper describing an attack on FEAL-4 requiring between 100 to 10,000 chosen plaintexts. In response to these weaknesses, the designers released FEAL-8 .

1990

Sean Murphy found an improvement to the chosen plaintext attack on FEAL-4 that needed only 20 chosen plaintexts. Additionally, Gilbert and Chassé published a statistical attack on FEAL-8 similar to differential cryptanalysis, requiring 10,000 pairs of chosen plaintexts .

1991

Eli Biham and Adi Shamir described a differential attack on FEAL-8 . They also showed that both FEAL-N and FEAL-NX could be broken faster than exhaustive search for N ≤ 31 rounds using differential cryptanalysis .

1992

Matsui and Yamagishi presented a new method for a known plaintext attack of FEAL cipher. Their attack could break FEAL-4 with just 5 known plaintexts, FEAL-6 with 100, and FEAL-8 with 2^15 known plaintexts .

NUMBER CHOSEN PLAIN TEXTS REQUIRED

FEAL VARIANT	ATTACK TYPE	NUMBER OF TEXT	COMPLEXITY
FEAL-4	Differential	12	5 mins
FEAL-6	Boomerang	4	1 min

DIFFERENTIAL CRYPTANALYSIS

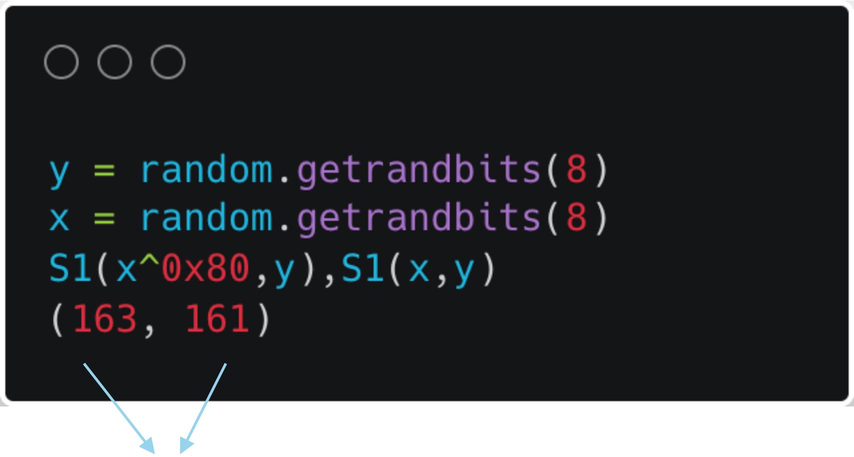
This is type of chosen plaintext attack, which analyzes the changes in the plaintext on changes in the resulting cipher text.

Basically, at each step it produces two plaintext messages differing by some value. It observes the difference in the two ciphertexts generated by the encryptor. Hence deriving a differential characteristic for the S-Box or the non-linear component of the cipher.

Objective is to perform a differential characteristic over the first n-1 rounds and generate pairs plaintexts with defined differential and extract the respective cyphertexts.

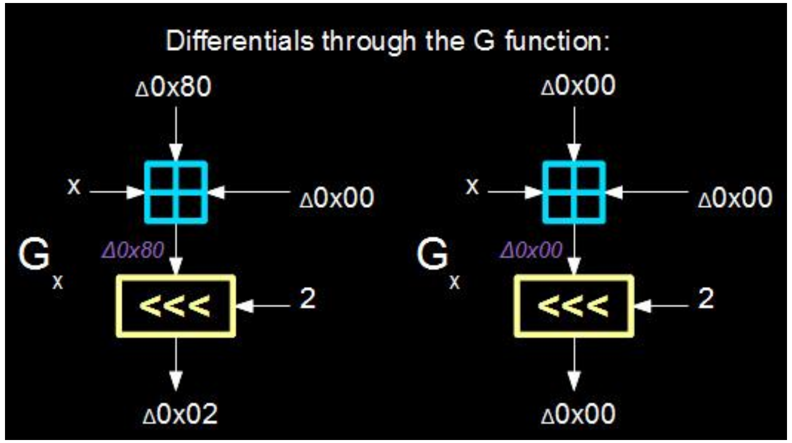
For retrieving the the last key, we backpropagate from the cipher text and perform search over the key space and choose the key which satisfies the differential characteristic the maximum number of times.

G-BOX DIFFERENTIAL ANALYSIS



```
y = random.getrandbits(8)
x = random.getrandbits(8)
S1(x^0x80,y),S1(x,y)
(163, 161)
```

Differential of 0x02



Linear operations in the scheme affect differentials in a both ways, so once again the problem boils down to analysing the effects of the non-linear G-boxes on differentials.

For every possible input difference, the distribution of output differences is analysed. Each G-box differential is a tuple (ΔX, ΔY, p) such that an input difference of ΔX gives an output difference of ΔY with probability p.

For FEAL G-Box the differentials

$$G_x(0x80\ ,\ 0x00)\ =\ 0x02\ \ ,\ \text{with } p=1$$

$$G_x(0x00\ ,\ 0x00)\ =\ 0x00\ \ ,\ \text{with } p=1$$

$$G_x(0x00\ ,\ 0x80)\ =\ 0x02\ \ ,\ \text{with } p=1$$

$$G_x(0x80\ ,\ 0x80)\ =\ 0x00\ \ ,\ \text{with } p=1$$

EXAMPLE

$$G_i(a,b) = ((a+b+i)\ \%\ 256) << 2$$

Values used in G	Binary	Hex & Decimal
A 0x80	0000 0001 1000 0000	0x01 = 1d 0x80 = 128d
A XOR 0x80	1000 0001	0x81 = 129d

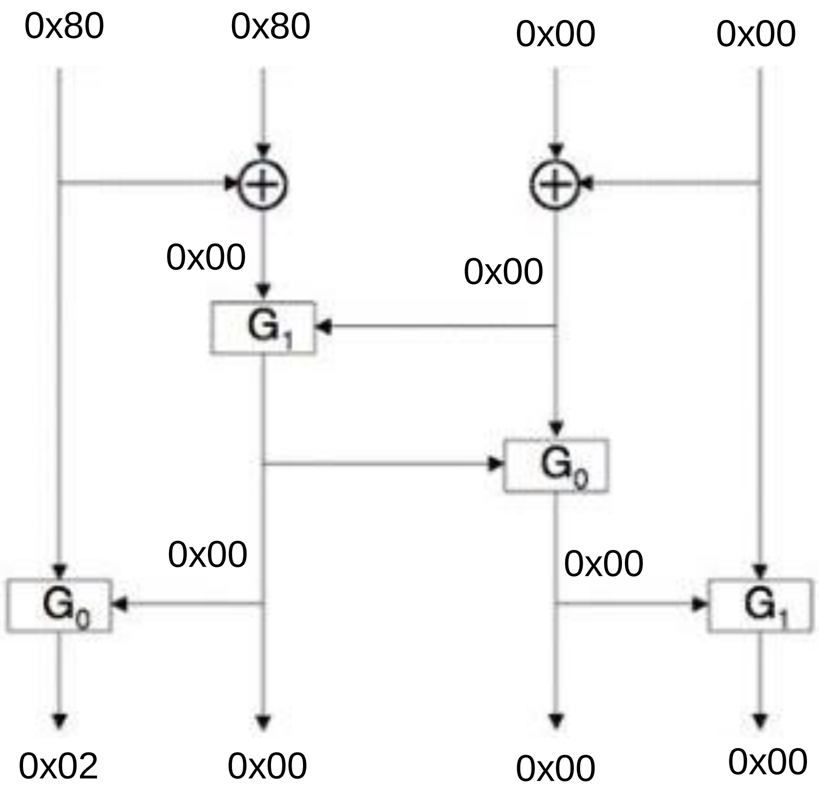
A+
B+
1
	0000 0001
	0000 0100
	0000 0001
		0x01 = 1d
		0x04 = 4d
		0x01 = 1d

	0000 0110	0x06 = 6d
Rotate 2 bits from left	0001 1000	0x18 = 24d

B+
(A XOR 0x80)+
1
	0000 0100
	1000 0001
	0000 0001
		0x04 = 4d
		0x81 = 128d
		0x01 = 1d

	1000 0110	0x86 = 134d
Rotate 2 bits from left	0001 1010	0x1A = 26d

ROUND FUNCTION DIFFERENTIAL ANALYSIS



The Input Difference to G-Box is 0x80800000

$$\Delta X \oplus \Delta X = 0x00$$

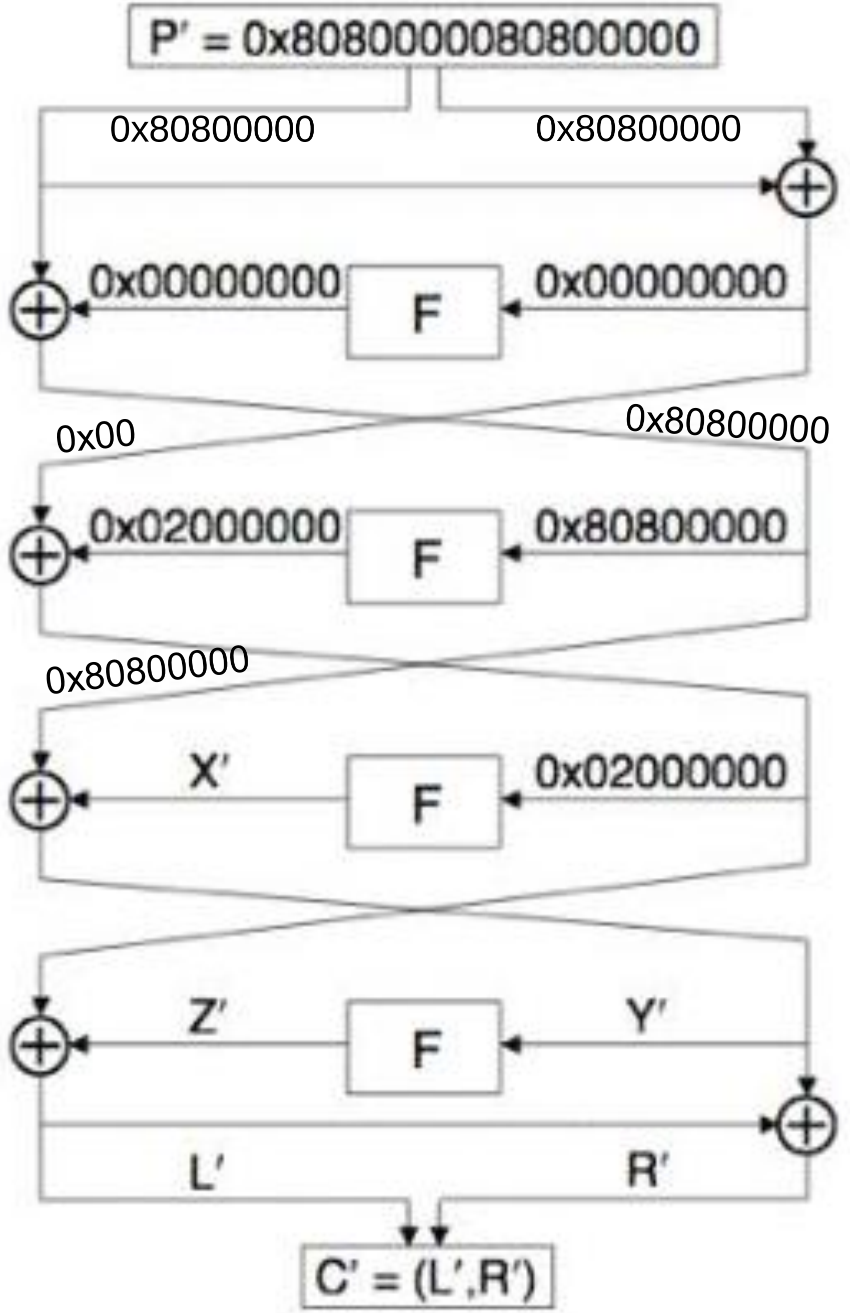
Using the above equation and relation derived from G-Box we can trace the differentials of the Round Function

The output differentials of the Round function is **0x02000000 with probability of 1.**

We can perform another tracing for input differential of 0x00000000. The output differentials of the Round function is **0x00000000 with probability of 1.**

TRACING THE WHOLE CIPHER

- Now using the differential characteristic, we derived for the Round Function(F) we can apply it to the whole FEAL-4 encryption scheme.
- If we inject the cipher with input differential of 0x8080000080800000, will split into two differentials of 32-bit each.
- Note : Before each round function(F) we perform XOR operation with round Subkey. Since same key is applied for both plaintexts so the difference is always 0x00 for this operation.
- We can trace the differentials with 100% certainty until round 3 - F function.



BACK-TRACING THE FINAL ROUND FROM CIPHER TEXT

Extracting K3

- Using the ciphertexts (known), the values of the bitstrings which were XOR-ed with K3 in the final round can be calculated (as left-ciphertext XOR right-ciphertext).

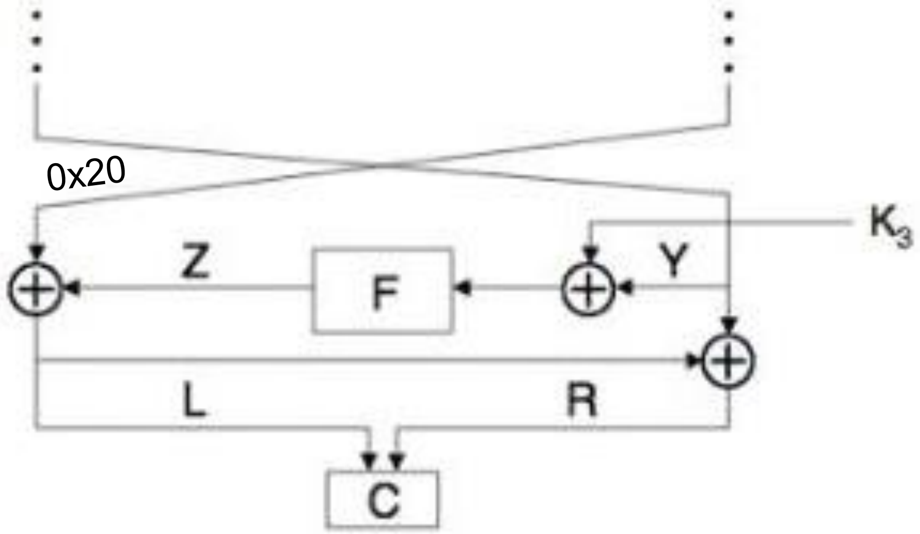
$$Y = L \oplus R$$

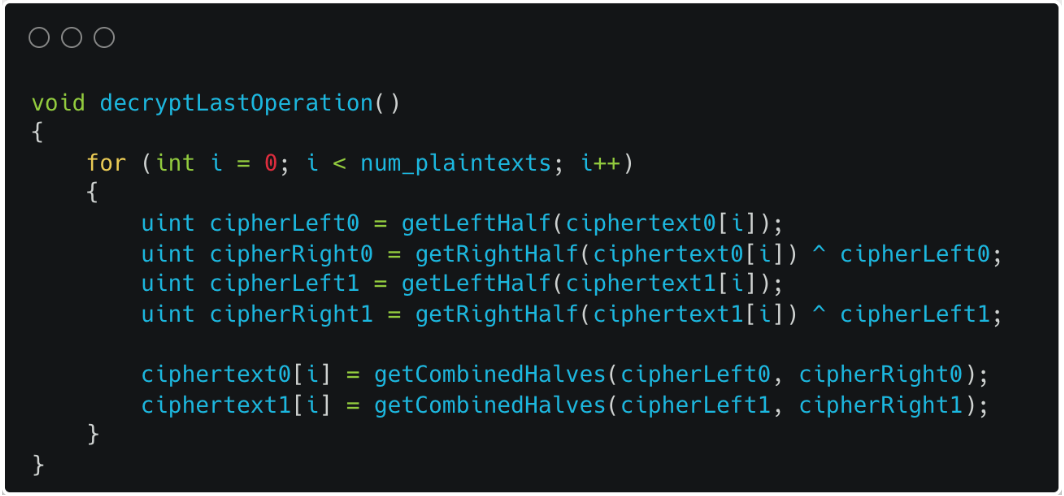
- We now do a linear search over the key space 6 (32-bit strings) of K3, and for every key value, we XOR it with the bitstrings calculated just a step ago. Then the results are passed through the f-Box, to get encryptions on the other side of the f-function.
- Run these pairs of texts individually through the round function to produce a pair of last round output texts. XOR these together to get a candidate differential.

$$C1(L) \oplus C2(L) \oplus 0x20 = \text{Predicted Difference}$$

$$\Delta Z = \text{Observed Difference for that key}$$

- If observed and actual matches, it means that the candidate key we used is a good bet for being the real one. We keep score for each candidate subkey and increment the score if these differentials match for each pair.
- If we get a key with all pairs being satisfied, we break and that keys is last round's key.
- Input Differential for K3 = **0x8080000080800000**





```
void decryptLastOperation()
{
    for (int i = 0; i < num_plaintexts; i++)
    {
        uint cipherLeft0 = getLeftHalf(ciphertext0[i]);
        uint cipherRight0 = getRightHalf(ciphertext0[i]) ^ cipherLeft0;
        uint cipherLeft1 = getLeftHalf(ciphertext1[i]);
        uint cipherRight1 = getRightHalf(ciphertext1[i]) ^ cipherLeft1;

        ciphertext0[i] = getCombinedHalves(cipherLeft0, cipherRight0);
        ciphertext1[i] = getCombinedHalves(cipherLeft1, cipherRight1);
    }
}
```

BACK-TRACING THE SECOND AND THIRD ROUND FROM CIPHER TEXT

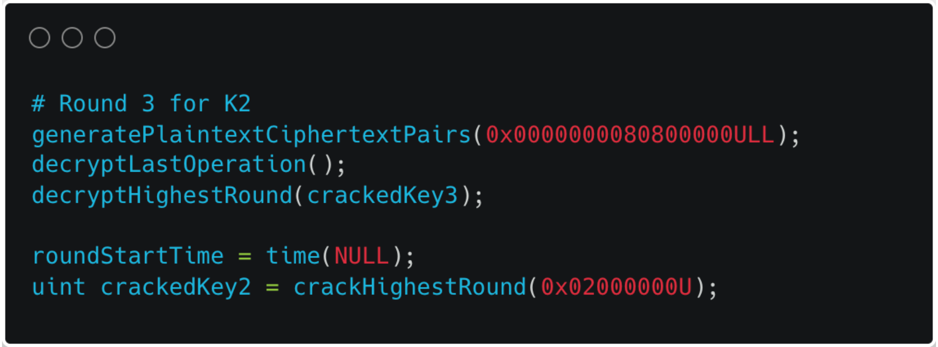
Extracting K2, K1

- Similar process as K3 is repeated for cracking K1,K2 just that the input differentials will change. We can use the previous K3 we cracked to decrypt the Round 4.

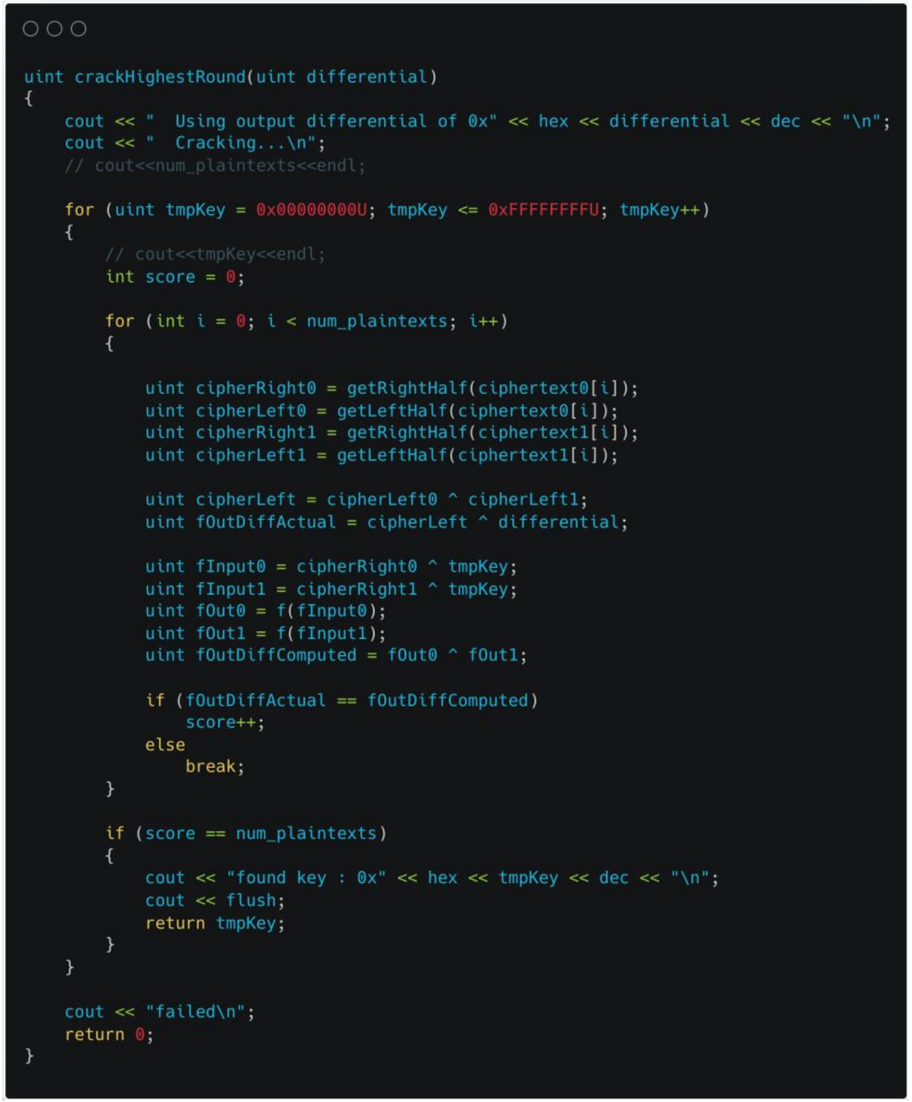
  Input Differential for K2 = 0x0000000080800000

  Input Differential for K1 = 0x0000000020000000

- We generate new chosen-plaintext pairs for each round using these input differentials. And perform linear search over key space of 32 bits. And take round key as the key with highest score or score = number of Plaintext pairs generated



```
# Round 3 for K2
generatePlaintextCiphertextPairs(0x0000000080800000ULL);
decryptLastOperation();
decryptHighestRound(crackedKey3);

roundStartTime = time(NULL);
uint crackedKey2 = crackHighestRound(0x02000000U);
```

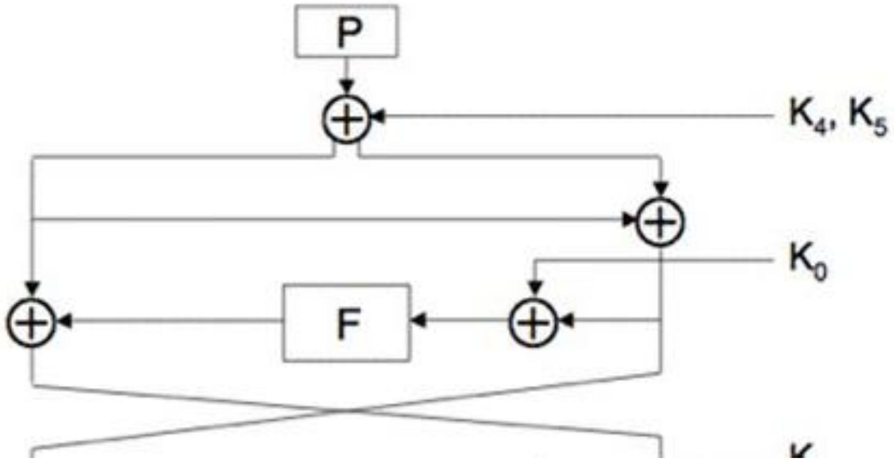


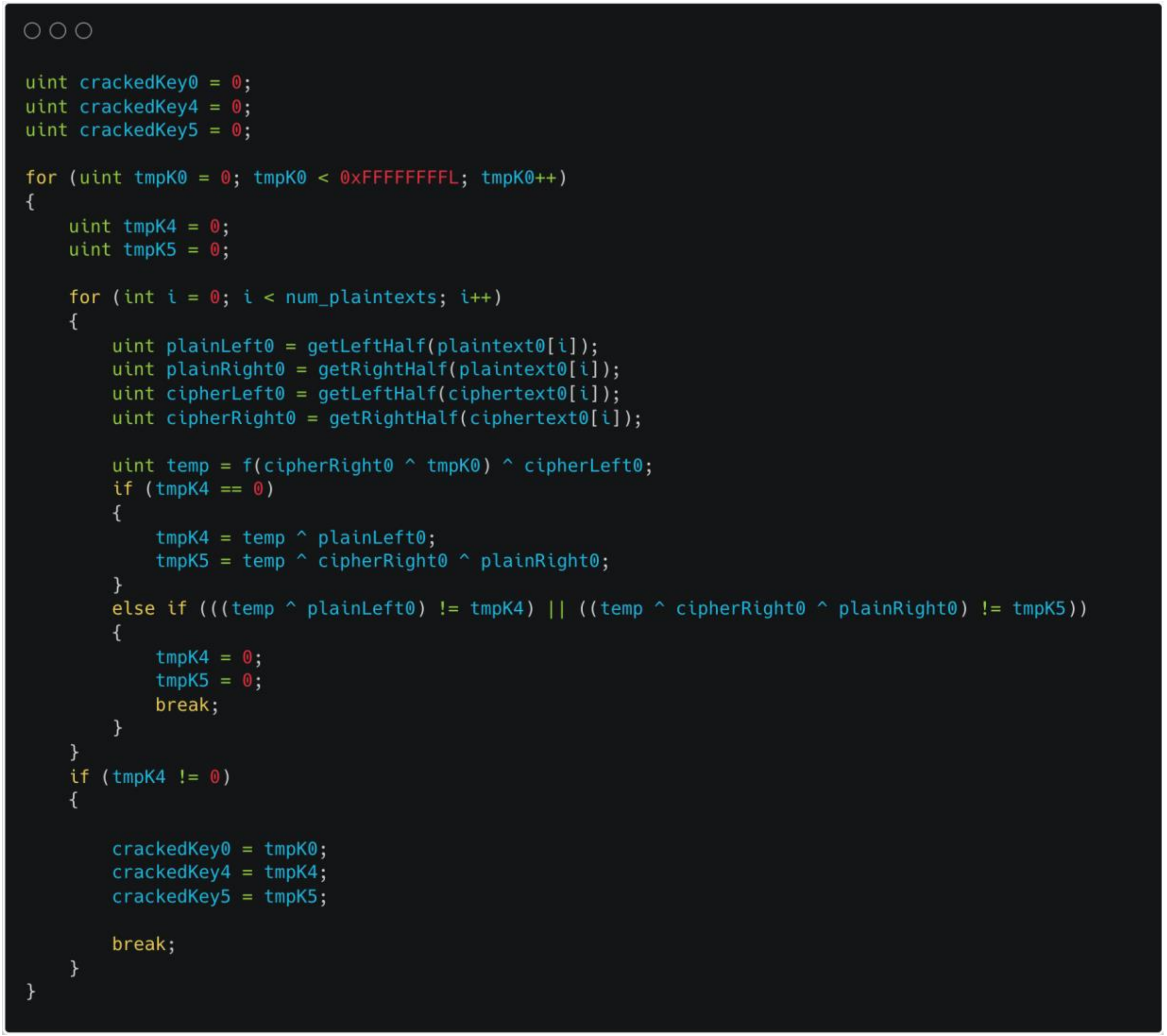
```
uint crackHighestRound(uint differential)
{
    cout << "  Using output differential of 0x" << hex << differential << dec << "\n";
    cout << "  Cracking...\n";
    // cout<<num_plaintexts<<endl;

    for (uint tmpKey = 0x00000000U; tmpKey <= 0xFFFFFFFFU; tmpKey++)
    {
        // cout<<tmpKey<<endl;
        int score = 0;

        for (int i = 0; i < num_plaintexts; i++)
        {

            uint cipherRight0 = getRightHalf(ciphertext0[i]);
            uint cipherLeft0 = getLeftHalf(ciphertext0[i]);
            uint cipherRight1 = getRightHalf(ciphertext1[i]);
            uint cipherLeft1 = getLeftHalf(ciphertext1[i]);

            uint cipherLeft = cipherLeft0 ^ cipherLeft1;
            uint fOutDiffActual = cipherLeft ^ differential;

            uint fInput0 = cipherRight0 ^ tmpKey;
            uint fInput1 = cipherRight1 ^ tmpKey;
            uint fOut0 = f(fInput0);
            uint fOut1 = f(fInput1);
            uint fOutDiffComputed = fOut0 ^ fOut1;

            if (fOutDiffActual == fOutDiffComputed)
                score++;
            else
                break;
        }

        if (score == num_plaintexts)
        {
            cout << "found key : 0x" << hex << tmpKey << dec << "\n";
            cout << flush;
            return tmpKey;
        }
    }

    cout << "failed\n";
    return 0;
}
```

BACK-TRACING THE FIRST ROUND FROM CIPHER TEXT

Extracting K0, K4, K5

- For the first round, all three of K0 , K4 , and K5 need to be extracted. For this, we do a linear search over the 32-bit space for K0 , and for every candidate, the bitstrings are decrypted through the candidate K0 , and used to compute K4 and K5 by using the input plaintext.
- The candidate key which yields the same K4 and K5 for all plaintext pairs it is tested with is selected as K0 , and the corresponding K4 and K5 are evaluated.





```
uint crackedKey0 = 0;
uint crackedKey4 = 0;
uint crackedKey5 = 0;

for (uint tmpK0 = 0; tmpK0 < 0xFFFFFFFFL; tmpK0++)
{
    uint tmpK4 = 0;
    uint tmpK5 = 0;

    for (int i = 0; i < num_plaintexts; i++)
    {
        uint plainLeft0 = getLeftHalf(plaintext0[i]);
        uint plainRight0 = getRightHalf(plaintext0[i]);
        uint cipherLeft0 = getLeftHalf(ciphertext0[i]);
        uint cipherRight0 = getRightHalf(ciphertext0[i]);

        uint temp = f(cipherRight0 ^ tmpK0) ^ cipherLeft0;
        if (tmpK4 == 0)
        {
            tmpK4 = temp ^ plainLeft0;
            tmpK5 = temp ^ cipherRight0 ^ plainRight0;
        }
        else if (((temp ^ plainLeft0) != tmpK4) || ((temp ^ cipherRight0 ^ plainRight0) != tmpK5))
        {
            tmpK4 = 0;
            tmpK5 = 0;
            break;
        }
    }
    if (tmpK4 != 0)
    {

        crackedKey0 = tmpK0;
        crackedKey4 = tmpK4;
        crackedKey5 = tmpK5;

        break;
    }
}
```

RUNTIME AND EFFICIENCY

- Exhaustive search of the key space from the perspective of subkeys means brute forcing every permutation of 192 bits, which 2^192 complexity

- However, with differential cryptanalysis, we only need to do individual key space searches for four 32-bit keys, i.e., each key space search is over 2^32 = 4294967296 keys, which sums up to a total of 4 × 4294967296 = 17179869184 ≈ 10^10 keys.

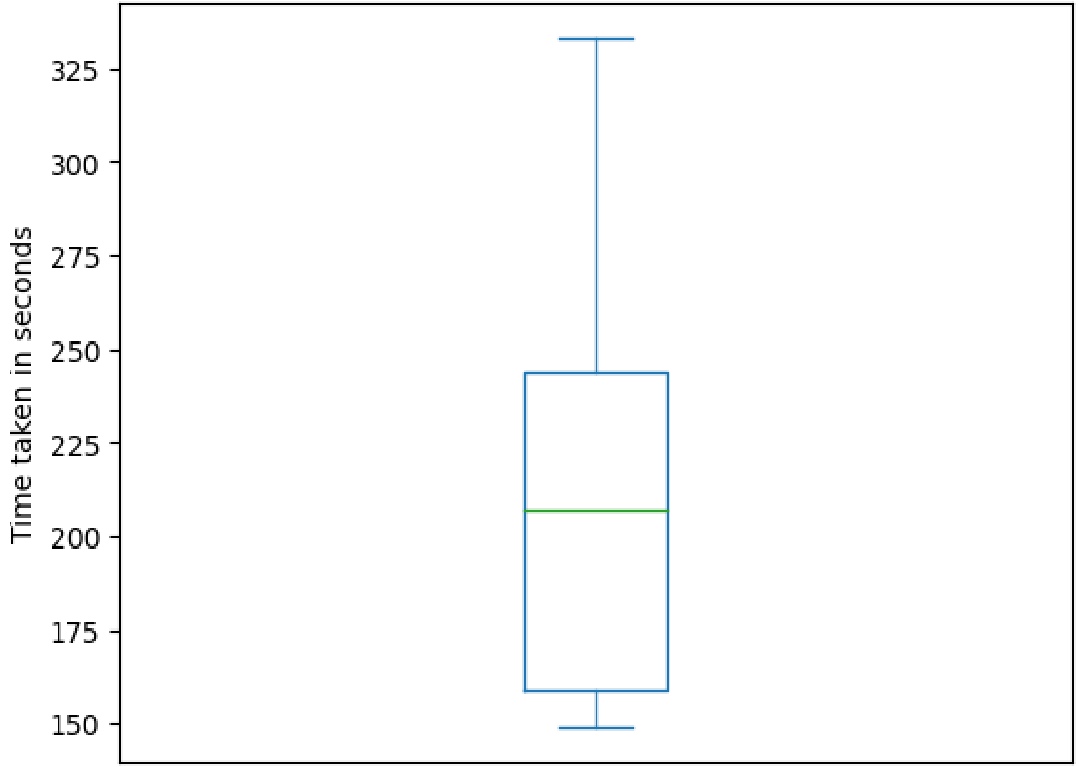
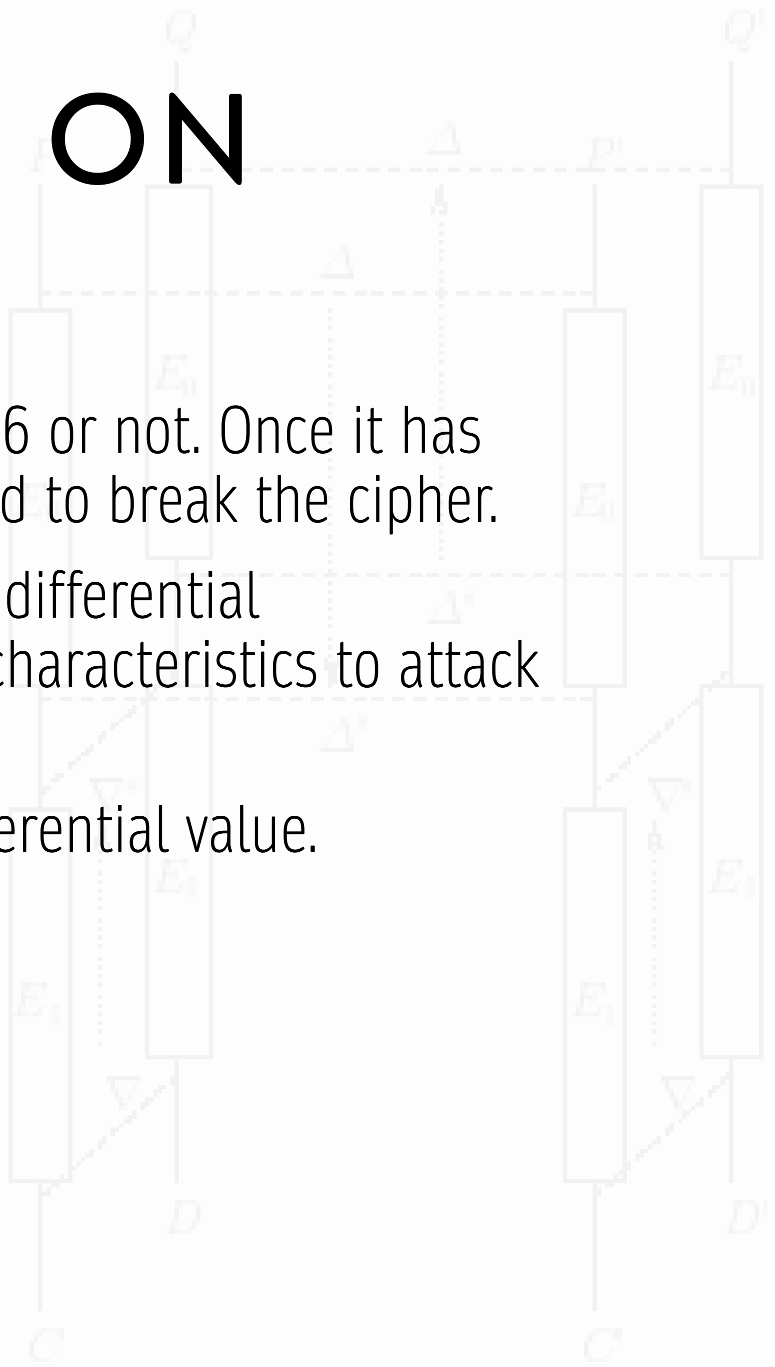


Fig : Runtime distribution to crack the keys with 12 plaintext

BOOMERANG ATTACK ON FEAL-6

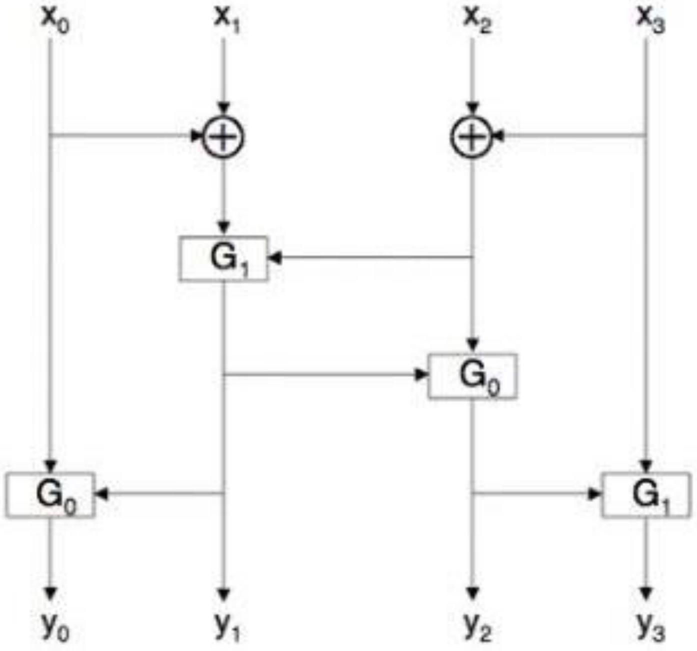
- This attack helps to ascertain whether the algorithm is FEAL-6 or not. Once it has been confirmed, further linear cryptanalysis can be performed to break the cipher.
- Boomerang attacks are a way to extend the power of normal differential cryptanalysis. It allows the use of two unrelated differential characteristics to attack the same cipher.
- 2 plaintext blocks are chosen which differ by some input differential value.

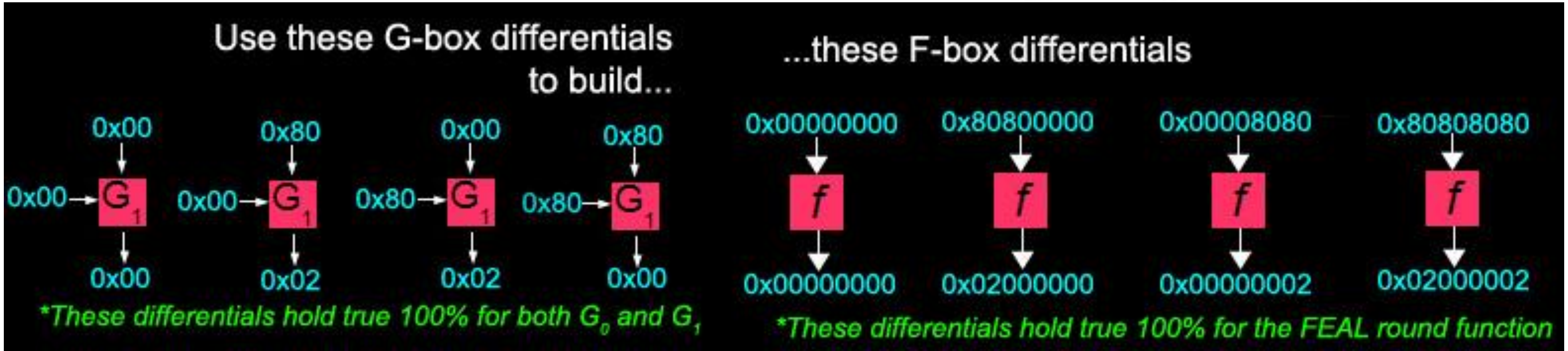


PREREQUISITES

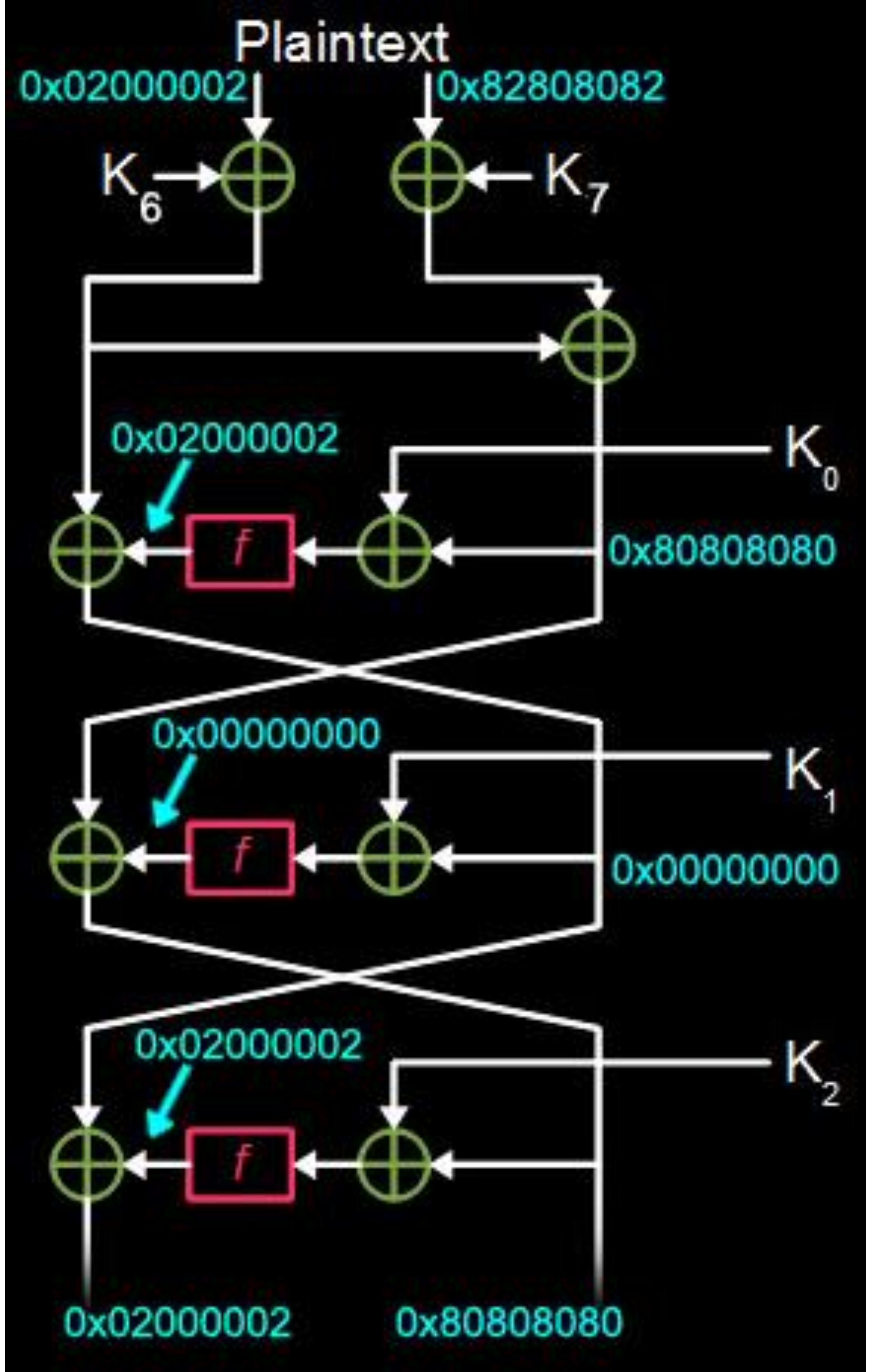
Like the standard G-box differentials, there are some F-box differentials which never change:

- 0x00000000 ⟶ 0x00000000
- 0x80800000 ⟶ 0x02000000
- 0x00008080 ⟶ 0x00000002
- 0x80808080 ⟶ 0x02000002



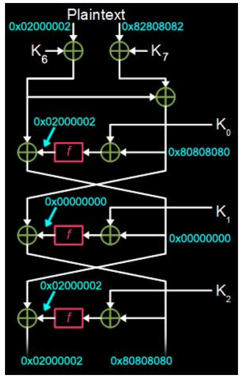

Use these G-box differentials to build...
0x00
0x00
G_1
0x00
0x80
0x00
G_1
0x02
0x00
0x80
G_1
0x02
0x80
0x80
G_1
0x00
*These differentials hold true 100% for both G_0 and G_1
...these F-box differentials
0x00000000
f
0x00000000
0x80800000
f
0x02000000
0x00008080
f
0x00000002
0x80808080
f
0x02000002
*These differentials hold true 100% for the FEAL round function

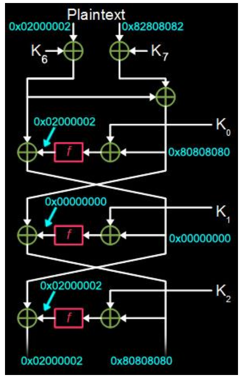
WORKING OF BOOMERANG ATTACK

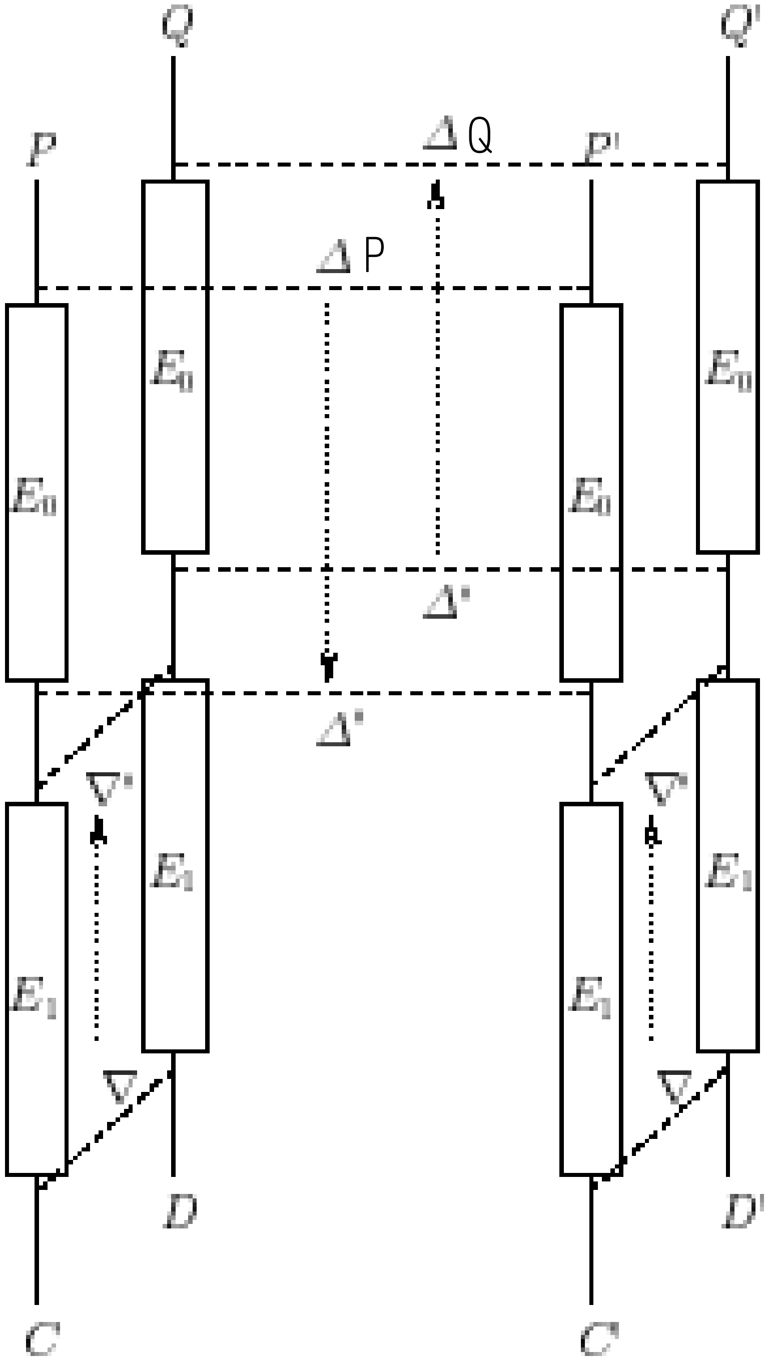


CRYPTANALYSIS

WORKING OF BOOMERANG ATTACK







EXPLANATION

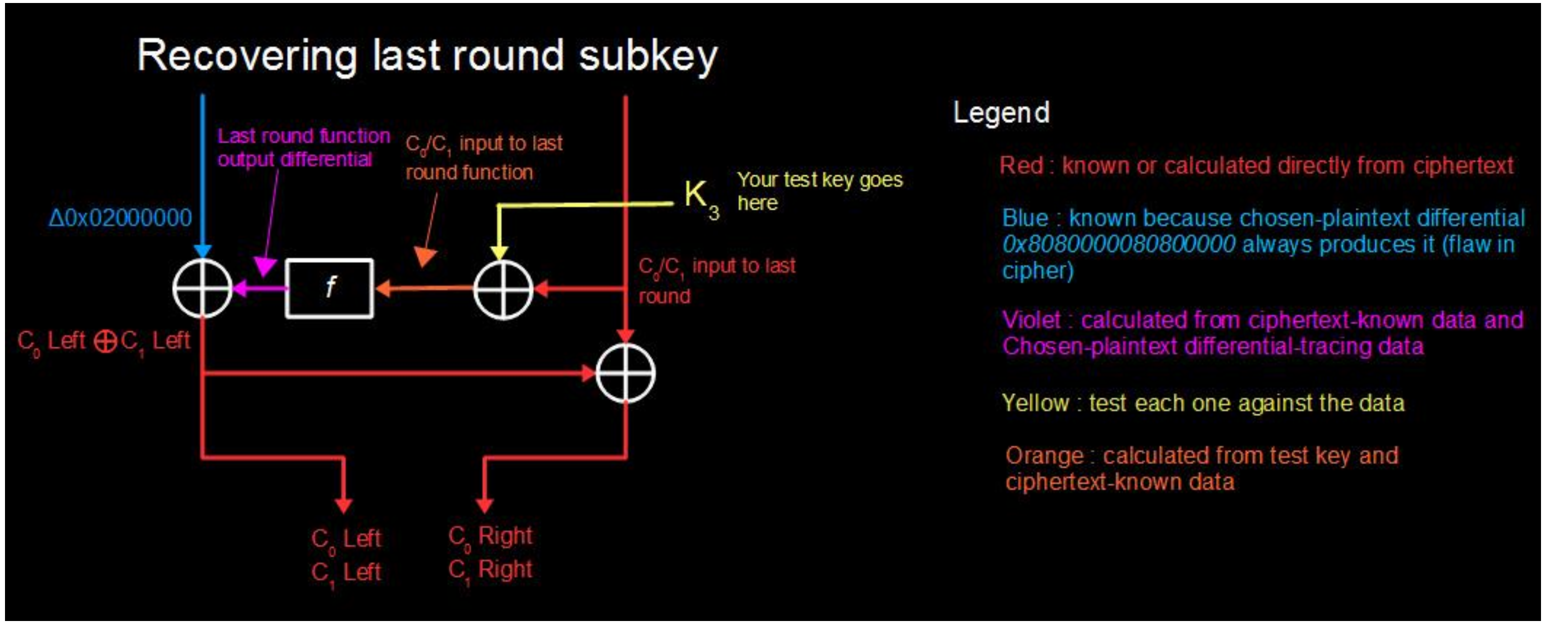
The key steps of a boomerang attack on FEAL-6 are:

- Choose two plaintext pairs (P1, P2) and (P3, P4) that have a specific XOR difference (ΔP). When encrypted, this is likely to lead to ciphertexts C1, C2, C3 and C4 that also have some XOR difference (ΔC).
- Then decrypt C3 and C4 to get the plaintexts P3' and P4'. XOR the decrypted plaintexts to get ΔP'.
- If ΔP' = ΔP, it means the differential characteristic held with high probability over the 6 rounds of encryption and decryption. This allows the attacker to determine information about the key.
- By analyzing many plaintext pairs and finding many "boomerang" quartets where ΔP = ΔP', the key bits can be determined faster than brute force through statistical analysis.

REFERENCES

1. Matsui, Mitsuru, and Atsuhiro Yamagishi. "A new method for known plaintext attack of FEAL cipher." Advances in Cryptology—EUROCRYPT'92: Workshop on the Theory and Application of Cryptographic Techniques Balatonfüred, Hungary, May 24–28, 1992 Proceedings 11. Springer Berlin Heidelberg, 1993.
2. http://www.theamazingking.com/crypto-feal.php
3. Miyaguchi, Shoji. "The FEAL cipher family." Advances in Cryptology-CRYPTO'90: Proceedings 10. Springer Berlin Heidelberg, 1991.
4. https://github.com/Vozec/Feal-ALL/tree/main/FEAL
5. Wagner, D. (1999). The Boomerang Attack. In: Knudsen, L. (eds) Fast Software Encryption. FSE 1999. Lecture Notes in Computer Science, vol 1636. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-48519-8_12

THANK YOU



Recovering last round subkey

Last round function output differential
C_0/C_1 input to last round function
K_3
Your test key goes here
Δ0x02000000
f
C_0/C_1 input to last round
C_0 Left $\oplus$ C_1 Left
C_0 Left
C_1 Left
C_0 Right
C_1 Right

Legend

Red : known or calculated directly from ciphertext

Blue : known because chosen-plaintext differential *0x8080000080800000* always produces it (flaw in cipher)

Violet : calculated from ciphertext-known data and Chosen-plaintext differential-tracing data

Yellow : test each one against the data

Orange : calculated from test key and ciphertext-known data