Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

przedmiot
Kryptografia stosowana (KRYS)

Szyfr blokowy z kluczem symetrycznym - Camellia

Kamil Chróścicki, Filip Smejda, Jakub Kitka, Andrzej Gawor

Numer albumu 300502, 300503, 300552, 300528

prowadzący
dr inż. Adam Komorowski

WARSZAWA 26 stycznia 2023

Spis treści

1. Wstęp

Szyfrowanie symetryczne jest podstawą współczesnej kryptografii. Są to algorytmy, które wykorzystują ten sam klucz zarówno do szyfrowania, jak i odszyfrowywania danych. Celem jest wykorzystanie krótkich tajnych kluczy do bezpiecznego i efektywnego przesyłania długich wiadomości. W dobie Internetu niezwykle ważna jest poufność i integralność danych. Transfer takich informacji musi być nie tylko odpowiednio szybki, ale przede wszystkim prawidłowo zabezpieczony przed niepowołanym dostępem. Szyfrowanie symetryczne jest w stanie to zapewnić i dzięki swojej charakterystyce jest powszechnie wykorzystywane w różnych rozwiązaniach. Przykłady, gdzie kryptografia symetryczna może zostać wykorzystana:

- Sektor bankowy: aplikacje płatnicze, walidacje potwierdzające nadawcę.
- Szyfrowanie wrażliwych danych na dysku pamięci (np. BitLocker).

Mnogość zastosowań szyfrowania symetrycznego sprawia, iż bezpieczeństwo użytkowników w sieci zależy w dużej mierze od wykorzystywanych algorytmów kryptograficznych. Szyfrowanie z kluczem symetrycznym można podzielić na dwa rodzaje:

- blokowy - tekst jawny jest dzielony na bloki o stałej długości i przechodzi przez funkcję szyfrującą wraz z sekretnym kluczem.
- strumieniowy - pojedynczy bajt tekstu jawnego jest szyfrowany poprzez operację XOR pseudolosowego strumienia klucza z danymi.

W naszej pracy skupimy się i szerzej omówimy szyfr blokowy z kluczem symetrycznym - Camellia.

1.1. Camellia

Camellia została opracowana wspólnie przez Nippon Telegraph[1] and Telephone Corporation i Mitsubishi Electric Corporation w 2000 roku.[2] Camellia określa 128-bitowy rozmiar bloku oraz 128-, 192- i 256-bitowy rozmiar klucza. Charakteryzuje się przydatnością zarówno do implementacji programowych, jak i sprzętowych, a także wysokim poziomem bezpieczeństwa. Z praktycznego punktu widzenia została zaprojektowana tak, aby umożliwić elastyczność w implementacjach programowych i sprzętowych na procesorach 32-bitowych szeroko stosowanych w Internecie i wielu aplikacjach, procesorach 8-bitowych stosowanych w kartach inteligentnych, sprzęcie kryptograficznym, czy w systemach wbudowanych. Jest zatwierdzona jako skuteczny i bezpieczny algorytm szyfrujący przez wiele organizacji na całym świecie m.in. Międzynarodową Organizację Normalizacyjną (ang. *International Organization for Standardization* - ISO), projekt badawczy UE NESSIE oraz japoński CRYPTREC.[3]

1.2. Camellia vs AES

W kryptografii występują różne implementacje blokowych algorytmów szyfrujących z kluczem symetrycznym. Najpopularniejszym i najczęściej stosowanym jest Advanced Encryption Standard (AES). Camellia jest uważana za mniej więcej równoważną AES pod względem bezpieczeństwa. Porównując oba rozwiązania można spostrzec pewne podobieństwa i różnice każdego z nich:

- Należą do grupy szyfrowania symetrycznego w trybie blokowym.
- Określają 128-bitowy rozmiar bloku i 128-, 192- i 256-bitowe rozmiary kluczy.
- Tylko AES jest standardem rządowym w USA. Zarówno NESSIE (UE) jak i CRYPTREC (Japonia) nadały AES i Camellia równy status [4].
- AES został sprawdzony przez kryptoanalityków w szerszym zakresie niż Camellia.
- AES działa na strukturze sieci SP, a Camellia na sieci Feistela.
- AES wypada wydajnościowo nieco lepiej porównując czas wymagany przez te algorytmy w funkcji długości tekstu jawnego.
- Camellia zapewnia doskonały czas konfiguracji klucza, a jego zwinność jest lepsza niż w przypadku AES [2].
- Camellia posiada poziomy bezpieczeństwa porównywalne z szyfrem AES/Rijndael.

2. Specyfikacja kryptosystemu

2.1. Wstęp

Camelia oparta jest na strukturze sieci Feistela. W wersji ze 128-bitowym kluczem, algorytm podzielony jest na 3 bloki po 6 rund Feistel'a. W wersjach z 192 i 256-bitowym kluczem występuje dodatkowy blok. Między blokami wywoływane są funkcje FL oraz FL^{-1}. Przed pierwszym oraz za ostatnim blokiem stosowana jest technika "Wybielania Klucza". Całość poprzedza "Faza Planowania Kluczy". Opis algorytmu może zostać podzielony na 3 części:

- Faza Planowania Kluczy,
- Szyfrowanie i deszyfrowanie,
- Funkcje algorytmu.

W tym rozdziale opisujemy poszczególne etapy działania kryptosystemu omawiając jednocześnie wykorzystywane oznaczenia i funkcje. Dodatkowo została zawarta nasza implementacja każdego z kluczowych elementów systemu.

2.2. Terminologia

Użyte oznaczenia:

X - dowolny wektor bitowy

$\mathbf{X_{L(n)}}$ - wektor powstały jako n bitów wektora X znajdujących się najbardziej po lewej stronie np. $0011_{L(2)} = 00$

$\mathbf{X_{R(n)}}$ - wektor powstały jako n bitów wektora X znajdujących się najbardziej po prawej stronie np. $0011_{R(2)} = 11$

!x - negacja wektora x

|| - operator konkatenacji

***x*«n** - cykliczna rotacja wektora x w lewą stronę o n bitów

$\vee - operatorlogicznyOR$

$\wedge - operatorlogicznyAND$

$\mathbf{K} - kluczgłówny$

Poniżej na listingu 1 przedstawiona została implementacja operatorów z których będziemy korzystać w całym rozwiązaniu. Są one podstawą do działania całego kryptosystemu.

```
def AND(x, y):
   return bytes(a & b for a, b in zip(x, y))
def OR(x, y):
    return bytes(a | b for a, b in zip(x, y))
def XOR(x, y):
    res = bytes(a ^ b for a, b in zip(x, y))
```

```
    return ''.join([format(x, 'b') for x in res]).encode('ascii')
def NOT(x):
    return bytes(a ^ 1 for a in x)
def LEFT(x, n):
    return x[:n]
def RIGHT(x, n):
    return x[(len(x)-n):]
def CONCATENATE(x, y):
    return x + y
def ROTATE(x, n):
   return x[n:]+x[:n]
```

Listing 1. Implementacja operatorów

2.3. Faza Planowania Kluczy

2.3.1. Derywacja zmiennych KL i KR

Na początku definiowane są 128-bitowe dwie zmienne KL oraz KR w następujący sposób, w zależności od długości klucza głównego:

- 128: $KL = K$, KR nie istnieje
- 192: $KL = K_{L(128)}$, $KR = K_{R(64)} \,||\, !K_{R(64)}$
- 256: $KL = K_{L(128)}$, $KR = K_{R(128)}$

Na listingu 2 została przedstawiona derywacja zmiennych KL i KR w języku Python.

```
def KL_KR_derivation(K):
    KL = bytes()
    KR = bytes()
    if (N_KEY_BITS == 128):
        KL = K
        KR = from_hex('00000000000000000000000000000000')
    if (N_KEY_BITS == 192):
        KL = LEFT(K,128)
        KR = CONCATENATE(RIGHT(K,64),NOT(RIGHT(K,64)))
    if (N_KEY_BITS == 256):
        KL = LEFT(K,128)
        KR = RIGHT(K,128)
    return KL, KR
```

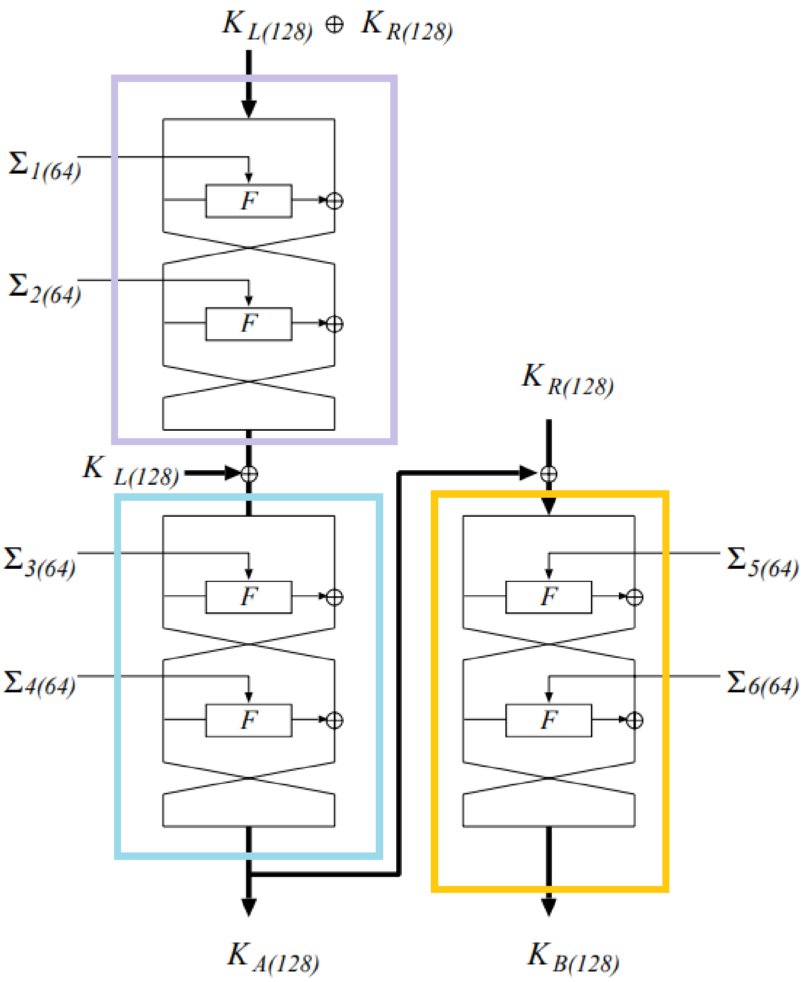
Listing 2. Derywacja zmiennych KL i KR

2.3.2. Wygenerowanie zmiennych KA i KB

Następnym krokiem jest wygenerowanie 128-bitowych zmiennych KA oraz KB (ta zmienna występuje jedynie w wersji z 192/256-bitowym kluczem głównym). Owa generacja opiera się na trzech blokach po dwie rundy szyfru Feistel'a. Jako klucze do funkcji F podawane są stałe zdefiniowane na rysunku 2.1. Schemat blokowy kroku znajduje się na rysunku 2.2. W postaci równań może zostać zapisany jak pokazano na listingu. D1 i D2 są tymczasowymi zmiennymi pomocniczymi.

```
Sigma1 = 0xA09E667F3BCC908B;
Sigma2 = 0xB67AE8584CAA73B2;
Sigma3 = 0xC6EF372FE94F82BE;
Sigma4 = 0x54FF53A5F1D36F1C;
Sigma5 = 0x10E527FADE682D1D;
Sigma6 = 0xB05688C2B3E6C1FD;
```

Rysunek 2.1. Stałe Sigma



Rysunek 2.2. Schemat blokowy generacji zmiennych KA i KB

//sekcja fioletowa

$D1 = (KL \oplus KR)_{L(64)}$

$D1 = (KL \oplus KR)_{\mathrm{R}(64)}$

$D2 = D2 \oplus F(D1, Sigma1)$

$D1 = D1 \oplus F(D2, Sigma2)$

$//sekcjaniebieska$

$D1 = D1 \oplus KL_{\mathrm{L}(64)}$

$D2 = D2 \oplus KL_{\mathrm{R}(64)}$

$D2 = D2 \oplus F(D1, Sigma3)$

$D1 = D1 \oplus F(D2, Sigma4)$

$KA = D1||D2$

$//sekcja\text{żół}ta$

$D1 = KA \oplus KR_{\mathrm{L}(64)}$

$D2 = KA \oplus KR_{\mathrm{R}(64)}$

$D2 = D2 \oplus F(D1, Sigma5)$

$D1 = D1 \oplus F(D2, Sigma6)$

$KB = D1||D2$

Na listingu 3 zostało przedstawione generowanie zmiennych KA i KB w języku Python.

```
def KA_KB_generation(KL, KR):
    D1 = LEFT(XOR(KL,KR),64)
    D2 = RIGHT(XOR(KL,KR),64)
    D2 = XOR(D2,F(D1, sigma1))
    D1 = XOR(D1,F(D2, sigma2))
    D1 = XOR(D1, LEFT(KL, 64))
    D2 = XOR(D2, RIGHT(KL, 64))
    D2 = XOR(D2, F(D1, sigma3))
    D1 = XOR(D1, F(D2, sigma4))
    KA = CONCATENATE(D1, D2)
    if (N_KEY_BITS==192 or N_KEY_BITS==256):
        D1 = LEFT(XOR(KA, KR), 64)
        D2 = RIGHT(XOR(KA, KR), 64)
        D2 = XOR(D2, F(D1, sigma5))
        D1 = XOR(D1, F(D2, sigma6))
        KB = CONCATENATE(D1, D2)
        return KA, KB
    return KA, None
```

Listing 3. Generowanie zmiennych KA i KB

2.3.3. Wygenerowanie właściwych pod-kluczy

Wszystkie utworzone wcześniej zmienne (KL, KR, KA, KB) są 128-bitowe. Wygenerowanie pod-kluczy polega na ich rotacji oraz braniu lewej lub prawej 64-bitowej połowy.

- 2 klucze do pre-whitening,

- po 6 kluczy wejściowych do funkcji F dla każdego 6-cio rudowego bloku szyfru Feistel'a,
- po 2 klucze wejściowe do funkcji FL oraz FL^{-1} między każdym blokiem,
- 2 klucze do post-whitening.

Rysunek 2.3 prezentuje cel, nazwę oraz sposób generacji pod-kluczy.

Subkeys for 128-bit secret key

	subkey	value
Prewhitening	$kw_{1(64)}$	$(K_L \lll_0)_{L(64)}$
	$kw_{2(64)}$	$(K_L \lll_0)_{R(64)}$
F (Round1)	$k_{1(64)}$	$(K_A \lll_0)_{L(64)}$
F (Round2)	$k_{2(64)}$	$(K_A \lll_0)_{R(64)}$
F (Round3)	$k_{3(64)}$	$(K_L \lll_{15})_{L(64)}$
F (Round4)	$k_{4(64)}$	$(K_L \lll_{15})_{R(64)}$
F (Round5)	$k_{5(64)}$	$(K_A \lll_{15})_{L(64)}$
F (Round6)	$k_{6(64)}$	$(K_A \lll_{15})_{R(64)}$
FL	$kl_{1(64)}$	$(K_A \lll_{30})_{L(64)}$
FL^{-1}	$kl_{2(64)}$	$(K_A \lll_{30})_{R(64)}$
F (Round7)	$k_{7(64)}$	$(K_L \lll_{45})_{L(64)}$
F (Round8)	$k_{8(64)}$	$(K_L \lll_{45})_{R(64)}$
F (Round9)	$k_{9(64)}$	$(K_A \lll_{45})_{L(64)}$
F (Round10)	$k_{10(64)}$	$(K_L \lll_{60})_{R(64)}$
F (Round11)	$k_{11(64)}$	$(K_A \lll_{60})_{L(64)}$
F (Round12)	$k_{12(64)}$	$(K_A \lll_{60})_{R(64)}$
FL	$kl_{3(64)}$	$(K_L \lll_{77})_{L(64)}$
FL^{-1}	$kl_{4(64)}$	$(K_L \lll_{77})_{R(64)}$
F (Round13)	$k_{13(64)}$	$(K_L \lll_{94})_{L(64)}$
F (Round14)	$k_{14(64)}$	$(K_L \lll_{94})_{R(64)}$
F (Round15)	$k_{15(64)}$	$(K_A \lll_{94})_{L(64)}$
F (Round16)	$k_{16(64)}$	$(K_A \lll_{94})_{R(64)}$
F (Round17)	$k_{17(64)}$	$(K_L \lll_{111})_{L(64)}$
F (Round18)	$k_{18(64)}$	$(K_L \lll_{111})_{R(64)}$
Postwhitening	$kw_{3(64)}$	$(K_A \lll_{111})_{L(64)}$
	$kw_{4(64)}$	$(K_A \lll_{111})_{R(64)}$

Subkeys for 192/256-bit secret key

	subkey	value
Prewhitening	$kw_{1(64)}$	$(K_L \lll_0)_{L(64)}$
	$kw_{2(64)}$	$(K_L \lll_0)_{R(64)}$
F (Round1)	$k_{1(64)}$	$(K_B \lll_0)_{L(64)}$
F (Round2)	$k_{2(64)}$	$(K_B \lll_0)_{R(64)}$
F (Round3)	$k_{3(64)}$	$(K_R \lll_{15})_{L(64)}$
F (Round4)	$k_{4(64)}$	$(K_R \lll_{15})_{R(64)}$
F (Round5)	$k_{5(64)}$	$(K_A \lll_{15})_{L(64)}$
F (Round6)	$k_{6(64)}$	$(K_A \lll_{15})_{R(64)}$
FL	$kl_{1(64)}$	$(K_R \lll_{30})_{L(64)}$
FL^{-1}	$kl_{2(64)}$	$(K_R \lll_{30})_{R(64)}$
F (Round7)	$k_{7(64)}$	$(K_B \lll_{30})_{L(64)}$
F (Round8)	$k_{8(64)}$	$(K_B \lll_{30})_{R(64)}$
F (Round9)	$k_{9(64)}$	$(K_L \lll_{45})_{L(64)}$
F (Round10)	$k_{10(64)}$	$(K_L \lll_{45})_{R(64)}$
F (Round11)	$k_{11(64)}$	$(K_A \lll_{45})_{L(64)}$
F (Round12)	$k_{12(64)}$	$(K_A \lll_{45})_{R(64)}$
FL	$kl_{3(64)}$	$(K_L \lll_{60})_{L(64)}$
FL^{-1}	$kl_{4(64)}$	$(K_L \lll_{60})_{R(64)}$
F (Round13)	$k_{13(64)}$	$(K_R \lll_{60})_{L(64)}$
F (Round14)	$k_{14(64)}$	$(K_R \lll_{60})_{R(64)}$
F (Round15)	$k_{15(64)}$	$(K_B \lll_{60})_{L(64)}$
F (Round16)	$k_{16(64)}$	$(K_B \lll_{60})_{R(64)}$
F (Round17)	$k_{17(64)}$	$(K_L \lll_{77})_{L(64)}$
F (Round18)	$k_{18(64)}$	$(K_L \lll_{77})_{R(64)}$
FL	$kl_{5(64)}$	$(K_A \lll_{77})_{L(64)}$
FL^{-1}	$kl_{6(64)}$	$(K_A \lll_{77})_{R(64)}$
F (Round19)	$k_{19(64)}$	$(K_R \lll_{94})_{L(64)}$
F (Round20)	$k_{20(64)}$	$(K_R \lll_{94})_{R(64)}$
F (Round21)	$k_{21(64)}$	$(K_A \lll_{94})_{L(64)}$
F (Round22)	$k_{22(64)}$	$(K_A \lll_{94})_{R(64)}$
F (Round23)	$k_{23(64)}$	$(K_L \lll_{111})_{L(64)}$
F (Round24)	$k_{24(64)}$	$(K_L \lll_{111})_{R(64)}$
Postwhitening	$kw_{3(64)}$	$(K_B \lll_{111})_{L(64)}$
	$kw_{4(64)}$	$(K_B \lll_{111})_{R(64)}$

Rysunek 2.3. Generowanie pod-kluczy

Na listingu 4 został przedstawiony zaimplementowany przez nas proces generowania właściwych podkluczy dla klucza 128 bit. Na listingu 5 został przedstawiony zaimplementowany przez nas proces generowania właściwych podkluczy dla klucza 192 i 256 bit.

```
def subkeys_generation_128(KL, KR, KA, KB):
```

```
    KW1 = o.LEFT(o.ROTATE(KL,0),64)
    KW2 = o.RIGHT(o.ROTATE(KL,0),64)

    K1 = o.LEFT(o.ROTATE(KA,0),64)
    K2 = o.RIGHT(o.ROTATE(KA,0),64)
    K3 = o.LEFT(o.ROTATE(KL,15),64)
    K4 = o.RIGHT(o.ROTATE(KL,15),64)
    K5 = o.LEFT(o.ROTATE(KA,15),64)
    K6 = o.RIGHT(o.ROTATE(KA,15),64)

    KL1 = o.LEFT(o.ROTATE(KA,30),64)
    KL2 = o.RIGHT(o.ROTATE(KA,30),64)

    K7 = o.LEFT(o.ROTATE(KL,45),64)
    K8 = o.RIGHT(o.ROTATE(KL,45),64)
    K9 = o.LEFT(o.ROTATE(KA,45),64)
    K10 = o.RIGHT(o.ROTATE(KL,60),64)
    K11 = o.LEFT(o.ROTATE(KA,60),64)
    K12 = o.RIGHT(o.ROTATE(KA,60),64)

    KL3 = o.LEFT(o.ROTATE(KL,77),64)
    KL4 = o.RIGHT(o.ROTATE(KL,77),64)

    K13 = o.LEFT(o.ROTATE(KL,94),64)
    K14 = o.RIGHT(o.ROTATE(KL,94),64)
    K15 = o.LEFT(o.ROTATE(KA,94),64)
    K16 = o.RIGHT(o.ROTATE(KA,94),64)
    K17 = o.LEFT(o.ROTATE(KL,111),64)
    K18 = o.RIGHT(o.ROTATE(KL,111),64)

    KW3 = o.LEFT(o.ROTATE(KL,111),64)
    KW4 = o.RIGHT(o.ROTATE(KL,111),64)

    return (KW1, KW2, K1, K2, K3, K4, K5, K6, KL1, KL2,
            K7, K8, K9, K10, K11, K12, KL3, KL4, K13, K14,
            K15, K16, K17, K18, KW3, KW4)
```

Listing 4. Wygenerowanie podkluczy dla 128-bit sekretnego klucza.

```
def subkeys_generation_192_256(KL, KR, KA, KB):
    KW1 = o.LEFT(o.ROTATE(KL,0),64)
    KW2 = o.RIGHT(o.ROTATE(KL,0),64)

    K1 = o.LEFT(o.ROTATE(KB,0),64)
    K2 = o.RIGHT(o.ROTATE(KB,0),64)
```

```
    K3 = o.LEFT(o.ROTATE(KR,15),64)
    K4 = o.RIGHT(o.ROTATE(KR,15),64)
    K5 = o.LEFT(o.ROTATE(KA,15),64)
    K6 = o.RIGHT(o.ROTATE(KA,15),64)

    KL1 = o.LEFT(o.ROTATE(KR,30),64)
    KL2 = o.RIGHT(o.ROTATE(KR,30),64)

    K7 = o.LEFT(o.ROTATE(KB,30),64)
    K8 = o.RIGHT(o.ROTATE(KB,30),64)
    K9 = o.LEFT(o.ROTATE(KL,45),64)
    K10 = o.RIGHT(o.ROTATE(KL,45),64)
    K11 = o.LEFT(o.ROTATE(KA,45),64)
    K12 = o.RIGHT(o.ROTATE(KA,45),64)

    KL3 = o.LEFT(o.ROTATE(KL,60),64)
    KL4 = o.RIGHT(o.ROTATE(KL,60),64)

    K13 = o.LEFT(o.ROTATE(KR,60),64)
    K14 = o.RIGHT(o.ROTATE(KR,60),64)
    K15 = o.LEFT(o.ROTATE(KB,60),64)
    K16 = o.RIGHT(o.ROTATE(KB,60),64)
    K17 = o.LEFT(o.ROTATE(KL,77),64)
    K18 = o.RIGHT(o.ROTATE(KL,77),64)

    KL5 = o.LEFT(o.ROTATE(KA,77),64)
    KL6 = o.RIGHT(o.ROTATE(KA,77),64)

    K19 = o.LEFT(o.ROTATE(KR,94),64)
    K20 = o.RIGHT(o.ROTATE(KR,94),64)
    K21 = o.LEFT(o.ROTATE(KA,94),64)
    K22 = o.RIGHT(o.ROTATE(KA,94),64)
    K23 = o.LEFT(o.ROTATE(KL,111),64)
    K24 = o.RIGHT(o.ROTATE(KL,111),64)

    KW3 = o.LEFT(o.ROTATE(KB,111),64)
    KW4 = o.RIGHT(o.ROTATE(KB,111),64)

    return (KW1, KW2, K1, K2, K3, K4, K5, K6, KL1, KL2,
            K7, K8, K9, K10, K11, K12, KL3, KL4, K13, K14, K15,
            K16, K17, K18, KL5, KL6, K19, K20, K21, K22, K23,
            K24, KW3, KW4)
```

Listing 5. Wygenerowanie podkluczy dla 192-bit i 256-bit sekretnego klucza.

2.4. Szyfrowanie i deszyfrowanie

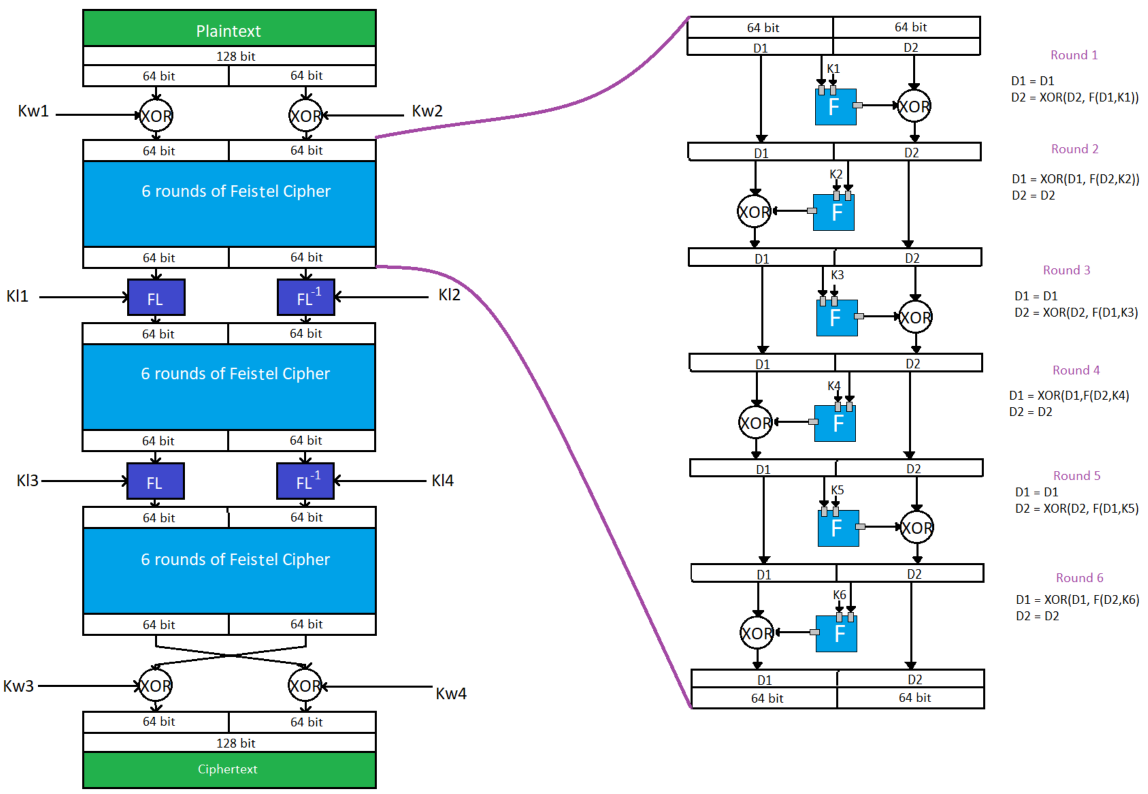
2.4.1. Szyfrowanie

Jako wejście do algorytmu pobierany jest 128-bitowy *plaintext*, który jest rozdzielany na dwie 64-bitowe części. Wyjściem algorytmu jest 128-bitowy *ciphertext*. Specyfikacja przedstawiona jest na rysunku 2.4. Funkcje F oraz FL i FL^{-1} znajdujące się na rysunku opisane są w następnej sekcji. Rysunek przedstawia wariant z 128-bitowym kluczem głównym. Wariant z kluczem głównym o długość 192 lub 256-bitów zawiera dodatkowy 6-cio rundowy blok szyfru Feistel'a.

Metoda obsługująca szyfrowanie została przedstawiona na listingu 6

```
def encrypt(plaintext, key):
    plaintext_blocks = get_128bit_blocks(plaintext)
    ciphertext = b''
    for i in plaintext_blocks:
        ciphertext += encrypt_block(i, key)
    return ciphertext
```

Listing 6. Operacja deszyfrowania szyfrogramu.



Rysunek 2.4. Szyfrowanie

2.4.2. Deszyfrowanie

Procedura deszyfrowania jest taka sama jak szyfrowania, jednakże należy podmienić kolejność używanych kluczy zgodnie z rysunkiem 2.5.

Metoda obsługująca deszyfrowanie została przedstawiona na listingu 7

```
def decrypt(ciphertext, key):
    ciphertext_blocks = get_128bit_blocks(ciphertext)
    plaintext = b''
    for i in ciphertext_blocks:
        plaintext += decrypt_block(i, key)
    return plaintext
```

Listing 7. Operacja deszyfrowania szyfrogramu.

128-bit key:
- Kw1 <-> Kw3
- Kw2 <-> Kw4
- K1 <-> K18
- K2 <-> K17
- K3 <-> K16
- K4 <-> K15
- K5 <-> K14
- K6 <-> K13
- K7 <-> K12
- K8 <-> K11
- K9 <-> K10
- Kl1 <-> Kl4
- Kl2 <-> Kl3

192- or 256-bit key:
- Kw1 <-> Kw3
- Kw2 <-> Kw4
- K1 <-> K24
- K2 <-> K23
- K3 <-> K22
- K4 <-> K21
- K5 <-> K20
- K6 <-> K19
- K7 <-> K18
- K8 <-> K17
- K9 <-> K16
- K10 <-> K15
- K11 <-> K14
- K12 <-> K13
- Kl1 <-> Kl6
- Kl2 <-> Kl5
- Kl3 <-> Kl4

Rysunek 2.5. Odwrócenie pod-kluczy

2.5. Funkcje algorytmu

2.5.1. Funkcja-F

Funkcja pobiera jako argumenty dwa wektory 64-bitowe, a zwraca jeden wektor 64-bitowy. Najpierw XORuje ona ze sobą wektory wejściowe, a wynikiem tej operacji wywołuje Funkcję-S. Następnie, to co zwróci funkcja S, przekazywane jest jako argument do Funkcji-P. Opis funkcji S oraz P znajduje się w następnych sekcjach. (X,k) → $Y = P(S(X \oplus k))$

Implementacja funkcji F jest zaprezentowana na listingu 8.

```
def F(x, k):
```

```
    result = P_function(S_function(XOR(x, k)))
    return result
```

Listing 8. Funkcja F.

2.5.2. Funkcja-S

Funkcja pobiera jako argument 64-bitowy wektor, i zwraca również 64-bitowy wektor. Swój argument dzieli na 8 części, które procesuje niezależnie zamieniając je odpowiednio bazując na S-box'ach (tabelach zamian).

x1||x2||x3||x4||x5||x6||x7||x8 → $y1||y2||y3||y4||y5||y6||y7||y8$

S-box'y mapują wejściowe 8 bitów na inny zestaw 8-śmiu bitów. Camellia definiuje 4 S-boxy, zaprezentowane w na rysunku 2.6 .

S1

This table below reads $s_1(0) = 112, s_1(1) = 130, \ldots, s_1(255) = 158$.

112	130	44	236	179	39	192	229	228	133	87	53	234	12	174	65
35	239	107	147	69	25	165	33	237	14	79	78	29	101	146	189
134	184	175	143	124	235	31	206	62	48	220	95	94	197	11	26
166	225	57	202	213	71	93	61	217	1	90	214	81	86	108	77
139	13	154	102	251	204	176	45	116	18	43	32	240	177	132	153
223	76	203	194	52	126	118	5	109	183	169	49	209	23	4	215
20	88	58	97	222	27	17	28	50	15	156	22	83	24	242	34
254	68	207	178	195	181	122	145	36	8	232	168	96	252	105	80
170	208	160	125	161	137	98	151	84	91	30	149	224	255	100	210
16	196	0	72	163	247	117	219	138	3	230	218	9	63	221	148
135	92	131	2	205	74	144	51	115	103	246	243	157	127	191	226
82	155	216	38	200	55	198	59	129	150	111	75	19	190	99	46
233	121	167	140	159	110	188	142	41	245	249	182	47	253	180	89
120	152	6	106	231	70	113	186	212	37	171	66	136	162	141	250
114	7	185	85	248	238	172	10	54	73	42	104	60	56	241	164
64	40	211	123	187	201	67	193	21	227	173	244	119	199	128	158

S2

224	5	88	217	103	78	129	203	201	11	174	106	213	24	93	130
70	223	214	39	138	50	75	66	219	28	158	156	58	202	37	123
13	113	95	31	248	215	62	157	124	96	185	190	188	139	22	52
77	195	114	149	171	142	186	122	179	2	180	173	162	172	216	154
23	26	53	204	247	153	97	90	232	36	86	64	225	99	9	51
191	152	151	133	104	252	236	10	218	111	83	98	163	46	8	175
40	176	116	194	189	54	34	56	100	30	57	44	166	48	229	68
253	136	159	101	135	107	244	35	72	16	209	81	192	249	210	160
85	161	65	250	67	19	196	47	168	182	60	43	193	255	200	165
32	137	0	144	71	239	234	183	21	6	205	181	18	126	187	41
15	184	7	4	155	148	33	102	230	206	237	231	59	254	127	197
164	55	177	76	145	110	141	118	3	45	222	150	38	125	198	92
211	242	79	25	63	220	121	29	82	235	243	109	94	251	105	178
240	49	12	212	207	140	226	117	169	74	87	132	17	69	27	245
228	14	115	170	241	221	89	20	108	146	84	208	120	112	227	73
128	80	167	246	119	147	134	131	42	199	91	233	238	143	1	61

S3

56	65	22	118	217	147	96	242	114	194	171	154	117	6	87	160
145	247	181	201	162	140	210	144	246	7	167	39	142	178	73	222
67	92	215	199	62	245	143	103	31	24	110	175	47	226	133	13
83	240	156	101	234	163	174	158	236	128	45	107	168	43	54	166
197	134	77	51	253	102	88	150	58	9	149	16	120	216	66	204
239	38	229	97	26	63	59	130	182	219	212	152	232	139	2	235
10	44	29	176	111	141	136	14	25	135	78	11	169	12	121	17
127	34	231	89	225	218	61	200	18	4	116	84	48	126	180	40
85	104	80	190	208	196	49	203	42	173	15	202	112	255	50	105
8	98	0	36	209	251	186	237	69	129	115	109	132	159	238	74
195	46	193	1	230	37	72	153	185	179	123	249	206	191	223	113
41	205	108	19	100	155	99	157	192	75	183	165	137	95	177	23
244	188	211	70	207	55	94	71	148	250	252	91	151	254	90	172
60	76	3	53	243	35	184	93	106	146	213	33	68	81	198	125
57	131	220	170	124	119	86	5	27	164	21	52	30	28	248	82
32	20	233	189	221	228	161	224	138	241	214	122	187	227	64	79

S4

112	44	179	192	228	87	234	174	35	107	69	165	237	79	29	146
134	175	124	31	62	220	94	11	166	57	213	93	217	90	81	108
139	154	251	176	116	43	240	132	223	203	52	118	109	169	209	4
20	58	222	17	50	156	83	242	254	207	195	122	36	232	96	105
170	160	161	98	84	30	224	100	16	0	163	117	138	230	9	221
135	131	205	144	115	246	157	191	82	216	200	198	129	111	19	99
233	167	159	188	41	249	47	180	120	6	231	113	212	171	136	141
114	185	248	172	54	42	60	241	64	211	187	67	21	173	119	128
130	236	39	229	133	53	12	65	239	147	25	33	14	78	101	189
184	143	235	206	48	95	197	26	225	202	71	61	1	214	86	77
13	102	204	45	18	32	177	153	76	194	126	5	183	49	23	215
88	97	27	28	15	22	24	34	68	178	181	145	8	168	252	80
208	125	137	151	91	149	255	210	196	72	247	219	3	218	63	148
92	2	74	51	103	243	127	226	155	38	55	59	150	75	190	46
121	140	110	142	245	182	253	89	152	106	70	186	37	66	162	250
7	85	238	10	73	104	56	164	40	123	201	193	227	244	199	158

Rysunek 2.6. S-box'y

Wartość yi wektora wyjściowego tworzone są w następujący sposób:

y1 = s1(x1)

y2 = s2(x2)

y3 = s3(x3)

y4 = s4(x4)

y5 = s2(x5)

y6 = s3(x6)

y7 = s4(x7)

y8 = s1(x8)

Implementacja funkcji S jest zaprezentowana na listingu 9.

```

def S_function(x):
    y1 = LEFT(x, 8)
    y2 = RIGHT(LEFT(x, 16), 8)
    y3 = RIGHT(LEFT(x, 24), 8)
    y4 = RIGHT(LEFT(x, 32), 8)
    y5 = RIGHT(LEFT(x, 40), 8)
    y6 = RIGHT(LEFT(x, 48), 8)
    y7 = RIGHT(LEFT(x, 56), 8)
    y8 = RIGHT(LEFT(x, 64), 8)

    y1 = '{0:08b}'.format(sbox1[int(y1, 2)]).encode('ascii')
    y2 = '{0:08b}'.format(sbox2[int(y2, 2)]).encode('ascii')
    y3 = '{0:08b}'.format(sbox3[int(y3, 2)]).encode('ascii')
    y4 = '{0:08b}'.format(sbox4[int(y4, 2)]).encode('ascii')
    y5 = '{0:08b}'.format(sbox2[int(y5, 2)]).encode('ascii')
    y6 = '{0:08b}'.format(sbox3[int(y6, 2)]).encode('ascii')
    y7 = '{0:08b}'.format(sbox4[int(y7, 2)]).encode('ascii')
    y8 = '{0:08b}'.format(sbox1[int(y8, 2)]).encode('ascii')

    return y1 + y2 + y3 + y4 + y5 + y6 + y7 + y8
```

Listing 9. Funkcja S.

2.5.3. Funkcja-P

Funkcja pobiera jako argument 64-bitowy wektor, i zwraca również 64-bitowy wektor. Swój argument dzieli na 8 części, które procesuje niezależnie.

$\mathrm{x1}||\mathrm{x2}||\mathrm{x3}||\mathrm{x4}||\mathrm{x5}||\mathrm{x6}||\mathrm{x7}||\mathrm{x8} \rightarrow y1||y2||y3||y4||y5||y6||y7||y8$

Wektor wyjściowy powstaje w następujący sposób:

$\mathrm{y1} = \mathrm{x1} \oplus x3 \oplus x4 \oplus x6 \oplus x7 \oplus x8$

$y2 = x1 \oplus x2 \oplus x4 \oplus x5 \oplus x7 \oplus x8$

$y3 = x1 \oplus x2 \oplus x3 \oplus x5 \oplus x6 \oplus x8$

$y4 = x2 \oplus x3 \oplus x4 \oplus x5 \oplus x6 \oplus x7$

$y5 = x1 \oplus x2 \oplus x6 \oplus x7 \oplus x8$

$y6 = x2 \oplus x3 \oplus x5 \oplus x7 \oplus x8$

$y7 = x3 \oplus x4 \oplus x5 \oplus x6 \oplus x8$

$y8 = x1 \oplus x4 \oplus x5 \oplus x6 \oplus x7$

Implementacja funkcji P jest zaprezentowana na listingu 10.

```
 def P_function(x):
    t1 = LEFT(x, 8)
```

```
    t2 = RIGHT(LEFT(x, 16), 8)
    t3 = RIGHT(LEFT(x, 24), 8)
    t4 = RIGHT(LEFT(x, 32), 8)
    t5 = RIGHT(LEFT(x, 40), 8)
    t6 = RIGHT(LEFT(x, 48), 8)
    t7 = RIGHT(LEFT(x, 56), 8)
    t8 = RIGHT(LEFT(x, 64), 8)

    y1 = XOR(XOR(XOR(XOR(XOR(t1, t3), t4), t6), t7), t8)
    y2 = XOR(XOR(XOR(XOR(XOR(t1, t2), t4), t5), t7), t8)
    y3 = XOR(XOR(XOR(XOR(XOR(t1, t2), t3), t5), t6), t8)
    y4 = XOR(XOR(XOR(XOR(XOR(t2, t3), t4), t5), t6), t7)
    y5 = XOR(XOR(XOR(XOR(t1, t2), t6), t7), t8)
    y6 = XOR(XOR(XOR(XOR(t2, t3), t5), t7), t8)
    y7 = XOR(XOR(XOR(XOR(t3, t4), t5), t6), t8)
    y8 = XOR(XOR(XOR(XOR(t1, t4), t5), t6), t7)

    return y1 + y2 + y3 + y4 + y5 + y6 + y7 + y8
```

Listing 10. Funkcja P.

2.5.4. Funkcja-FL

Funkcja pobiera jako argument dwa 64-bitowe wektory i zwraca jeden 64-bitowy wektor.

$(X_{L(32)} || X_{R(32)}, K_{L(32)} || K_{R(32)}) \rightarrow Y_{L(32)} || Y_{R(32)}$

Wektor wyjściowy powstaje w następujący sposób:

$Y_{R(32)} = ((X_{L(32)} \wedge K_{L(32)}) << 1) \oplus X_{R(32)},$

$Y_{L(32)} = (Y_{R(32)} \vee K_{R(32)}) \oplus X_{L(32)}$

Implementacja funkcji FL jest zaprezentowana na listingu 11.

```
def FL_function(x, k):
    XL = o.LEFT(x, 32)
    XR = o.RIGHT(x, 32)
    KL = o.LEFT(k, 32)
    KR = o.RIGHT(k, 32)
    YR = o.XOR(o.ROTATE(o.AND(XL, KL), 1), XR)
    YL = o.XOR(o.OR(YR, KR), XL)
    output = o.CONCATENATE(YL, YR)
    return output
```

Listing 11. Funkcja FL.

2.5.5. Funkcja-FL^{-1}

Funkcja pobiera jako argument dwa 64-bitowe wektory, i zwraca jeden 64-bitowy wektor.

$(Y_{L(32)} \| Y_{R(32)}, K_{L(32)} \| K_{R(32)}) \rightarrow X_{L(32)} || X_{R(32)}$

Wektor wyjściowy powstaje w następujący sposób:

$X_{L(32)} = (Y_{R(32)} \vee K_{R(32)}) \oplus Y_{L(32)}$,

$X_{R(32)} = ((X_{L(32)} \wedge K_{L(32)}) << 1) \oplus Y_{R(32)}$

Implementacja funkcji FL^{-1} jest zaprezentowana na listingu 12.

```

def FL1_function(y, k):
    YL = o.LEFT(y, 32)
    YR = o.RIGHT(y, 32)
    KL = o.LEFT(k, 32)
    KR = o.RIGHT(k, 32)
    XL = o.XOR(o.OR(YR, KR), YL)
    XR = o.XOR(o.ROTATE(o.AND(XL, KL), 1), YR)
    output = o.CONCATENATE(XL, XR)
    return output
```

Listing 12. Funkcja-FL^{-1}.

3. Bezpieczeństwo algorytmu

Camellia, oprócz wysokiego poziomu kompatybilności oraz elastyczności w przypadku implementacji programowych oraz sprzętowych, charakteryzuje się również z wysokim poziomem bezpieczeństwa. Została zatwierdzona jako skuteczny i bezpieczny algorytm szyfrujący przez takie organizacje jak ISO (ang. ang. *International Organization for Standardization*), projekt badawczy Unii Europejskiej NESSIE oraz japoński projekt CRYPTREC. Poziom bezpieczeństwa Camelli porównywalny jest do innego, popularnego szyfru z kluczem symetrycznym - AES (ang. ang. *Advanced Encryption Standard*).

3.1. Technika mieszania oraz rozproszenia

W kryptografii, dwoma właściwościami działania bezpiecznego szyfru są: technika mieszania (ang. *confusion*) oraz rozproszenia (ang. *diffusion*). W przypadku szyfrów blokowych, takich jak Camellia, zaimplementowane są obie te techniki, zapewniając:

- Mieszanie - zmniejsza związek między szyfrogramem a kluczem, poprzez to, że każdy bit szyfrogramu, powinien zależeć od kilku części klucza, czyli podkluczy. Właściwość ta utrudnia odnalezienie klucza na podstawie szyfrogramu, poprzez stworzenie wysokiej nieliniowości między nimi. W Camelli mieszanie zapewnia funkcja S, wykorzystywana przez funkcję F, czyli proces zamiany 64-bitowych danych wejściowych na inne 8 bajtów (bazując na tablicach S-box) zwracane do dalszego przetwarzania.
- Rozproszenie - ukrywa statystyczną zależność pomiędzy tekstem jawnym a szyfrogramem, poprzez to, że każdy bit tekstu jawnego, powinien mieć wpływ na szyfrogram. W Camelli rozproszenie zapewnia funkcja P, wykorzystywana przez funkcję F, czyli wykonanie kilku operacji XOR na każdym z 8 bajtów wejściowych z innymi bajtami wejściowymi, w celu otrzymania danych wyjściowych do dalszego przetwarzania.

Mieszanie pozwala stworzyć nieliniowość, jednak bez dyfuzji, ten sam bajt w tej samej pozycji otrzymywałby te same transformacje w każdej iteracji funkcji F. Pozwoliłoby to na atakowanie każdej pozycji bajtu w macierzy osobno. Tak więc, powinno się naprzemiennie stosować mieszanie (funkcja S) z rozproszeniem (funkcja P), tak aby konwersje zastosowane na jednym bajcie wpływały na wszystkie inne bajty w danym stanie. Wtedy, każde wejście do kolejnego S-box'a staje się funkcją wielu bajtów, co oznacza, że z każdą rundą algebraiczna złożoność systemu wzrasta.

3.2. Właściwości algebraiczne

Jako, że Camellia jest uznawana za bezpieczny szyfr, to nawet używając opcji najmniejszego możliwego klucza (128 bitów), uważa się, że złamanie szyfru poprzez atak siłowy (brute-force) jest niemożliwe przy aktualnej technologii. Szyfr ten może być zdefiniowany przez minimalny system wielomianów wielowymiarowych[5]:

- S-box'y Camelli (podobnie jak AES) mogą być opisane przez układ 23 równań kwadratowych przy użyciu 80 wyrażeń.
- Algorytm generowania podkluczy (key schedule) może być opisany przez 1120 równań zawierających 768 zmiennych przy użyciu 3 328 wyrażeń liniowych i kwadratowych.
- Cały szyfr blokowy może być opisany przez 5104 równania zawierających 2816 zmiennych przy użyciu 14592 wyrażeń liniowych i kwadratowych.
- Liczba wolnych wyrażeń (wyrażenia, które mogą zostać zastąpione wartością z S-box podczas wykonywania funkcji S) wynosi 11696, co daje podobną ilość jak dla AES.

W sumie, algorytm generowania podkluczy (ang. *key schedule*) oraz szyfr blokowy, składają się z 6224 równań zawierających 3584 zmiennych, wykorzystując 17920 wyrażeń liniowych i kwadratowych. Takie właściwości, w przyszłości, mogą umożliwić złamanie Camelli (oraz AES) za pomocą ataku algebraicznego, pod warunkiem, że stanie się on wykonalny. Dodatkowo, wymaga to zwiększenia mocy obliczeniowej komputerów, niezbędnej do rozwiązania niezwykle rozbudowanych problemów matematycznych.

4. Kryptoanaliza i ataki

Fakt, mówiący o tym, że Camelia wykorzystywana jest w dziedzinach bazujących na wysokim bezpieczeństwie oraz korzystających z szeroko pojętego pojęcia kryptografii wskazuje na to, iż w tym przypadku przeprowadzono szereg kryptoanaliz oraz potencjalnych ataków. Źródła powstałe na początku XXI wieku [6] dowodzą, że Camelia nie zawiera żadnych istotnych wad, czy też słabości. Dzięki jego relatywnie prostej oraz konserwatywnej budowie wszelkie przeprowadzone kryptoanalizy nie były dość problematyczne. W wyniku tego zauważono odporność tego szyfru na kryptoanalizę różnicową oraz liniową (ang. *differential and linear cryptanalisis*). Dotychczasowo, tak jak już wspomniano, uzyskano wiele wyników pochodzących z przeróżnych kryptoanaliz dla zredukowanej liczby rund Camelli rozróżniając wielostronne podejścia:

- differential and linear cryptanalysis,
- truncated differential cryptanalysis,
- integral attack,
- meet-in-the-middle attack,
- collision attack,
- impossible differential cryptanalysis,
- zero-corelation linear cryptanalysis.

Większość przeprowadzonych ataków przed 2011 r. wykluczała warstwy FL/FL-1 oraz "whitening' w celu ułatwienia kryptoanalizy (*"As a matter of fact, most attacks presented before 2011 excluded the FL/FL1and whitening layers to ease thecryptanalysis, whereas recent attacks aimed at reduced-roundCamellia with FL/FL1and/or whitening layers"* [7]). Jednakże z czasem zaczęto poznawać interesujące właściwości tego szyfru w dużym stopniu związane z pomijanymi dotychczasowo warstwami. I w ten sposób wprowadzono w przypadku jednej z kryptoanaliz 7-rundowy "impossible differential of Camelia"zawierający warstwy FL/FL-1, dzięki czemu przedstawili ulepszone ataki na 10-rundową Camelie-128, 10-rundową Camelie-192 oraz 11-rundową Camelie-256 [8]. Kolejnym przykładem ataku wykorzystującego podane warstwy było skonstruowanie 7 i 8-rundowych "impossible differentials of Camelia"z warstawmi FL/FL-1, a następnie zaatakowanie 11-rundowej Camelia-128, 12-rundową Camelia-192 oraz 13-rundową Camelia-256 [9]. Przełomowym podejściem było wykorzystywanie zerokorelacyjnych liniowych "distingusiherów"z FL/FL-1 oraz techniki opartej na szybkiej transformacie Fouriera (ang. *Fast Fourier Transform*) - FFT. Atak liniowy z zerową korelacją jest jedną z ostatnich metod kryptoanalizy wprowadzonych przez Bogdanowa oraz Rijmena[10]. Atak ten jest oparty na liniowych przybliżeniach z zerową korelacją, co w znaczny sposób różni go od klasycznej liniowej kryptoanalizy, w przypadku której wykorzystywane są charakterystyki o wysokich korelacjach. Samą idee ataku liniowego o zerowej korelacji można uznać za projekcję niemożliwej kryptoanalizy różnicowej na kryptoanalizę liniową. Do skonstruowania liniowego "distinguishera"charakteryzującego się zerową korelacją przyjmuje się technikę miss-in-the-middle co jest wykorzystywane w przypadku "impossible differential cryptanalysis", Poprzez wykorzystanie zaprezentowanej powyżej techniki zauważono, iż istnieją pewne interesujące właściwości funkcji FL/FL-1 w przypadku szyfru Camellia. Mianowicie, wówczas wprowadzone zostają tzw. słabe klucze*weakkeys*, dzięki którym zaprezentowano pierwszy 8-rundowy zero-korelacyjny liniowy "distinguisher"dla Camelli z warstwami FL/FL-1. Otrzymane wyniki pokazują, że rozważane warstwy FL/FL-1 zawarte w analizowanym szyfrze nie są w stanie skutecznie oprzeć się liniowej kryptoanalizie z zerową korelacją w przypadku niektórych słabych kluczy, gdyż obecnie najlepszy liniowy "distinguisher"z zerową korelacją również charakteryzuje się 8-rundami[7].

Table 1 Summary of the attacks on Camellia with FL/FL^{-1} and whitening layers

Key size	Cryptanalysis	Rounds	Data	Time (EN)	Memory, bytes
192	impossible differential	10	2^{121} CP	$2^{175.3}$	$2^{155.2}$
	impossible differential	10	$2^{118.7}$ CP	$2^{130.4}$	2^{135}
	impossible differential	11[a]	$2^{112.64}$ CP	$2^{146.54}$	$2^{141.64}$
	impossible differential	11	$2^{114.64}$ CP	2^{184}	$2^{141.64}$
	impossible differential	12	2^{123} CP	$2^{187.2}$	2^{160}
	multidimensional zero-correlation	12	$2^{125.7}$ KP	$2^{188.8}$	$2^{112.0}$
	zero-correlation	13[b]	2^{128} KP	$2^{169.83}$	$2^{156.86}$
256	higher-order differential	11	2^{93} CP	$2^{255.6}$	2^{98}
	impossible differential	11	2^{121} CP	$2^{206.8}$	2^{166}
	impossible differential	11	$2^{119.6}$ CP	$2^{194.5}$	2^{135}
	impossible differential	12[a]	$2^{121.12}$ CP	$2^{202.55}$	$2^{142.12}$
	impossible differential	12	$2^{116.17}$ CP	2^{240}	$2^{150.17}$
	impossible differential	13	2^{123} CP	$2^{251.1}$	2^{208}
	zero-correlation	14[b]	2^{128} KP	$2^{233.72}$	$2^{156.86}$

CP: chosen plaintext; KP: known plaintext; and EN: encryptions
aWeak keys under 2 bit conditions
[b]Weak keys under 15 bit conditions

Rysunek 4.1. Summary of the attacks on Camellia with FL/FL-1 and whitening layers

Pomimo potencjalnych "luk"skala prawdopodobieństwa skutecznego ataku jest mała, a wręcz niewspółmierna względem oferowanego bezpieczeństwa przez szyfr Camellia, w wyniku czego uważa się, iż faktyczne ataki na Camellię nie są praktycznie możliwe. Wymagałoby to przełomu w dziedzinie kryptoanaliz systemów szyfrujących. Jednakże nie jest to finalny, końcowy oraz niepodważalny wniosek. Uważa się, że sprecyzowana oraz odpowiednio długa kryptoanaliza może ujawnić właściwości, które dotychczasowo nie zostały wykryte.

5. Opis implementacji

5.1. Decyzje projektowe

Zaimplementowanie kryptosystemu Camellia wymagało od nas dokładnego przeanalizowania dostępnych do wykorzystania narzędzi i ustalenia ich przydatności do wykonania tego zadania. Do implementacji wybraliśmy popularny język programistyczny Python, z którym jesteśmy zapoznani, a także z którym pracowaliśmy w trakcie semestru na laboratoriach. Znajomość składni i funkcji wbudowanych umożliwiła sprawne posługiwanie się tym językiem. Utworzona została aplikacja konsolowa, która przyjmuje na wejściu od użytkownika dwa parametry na podstawie których zostaną wykonane działania przez kryptosystem.

Prace zostały przeprowadzone w IDE jakiem jest Visual Studio Code. W trakcie tworzenia rozwiązania nie posługiwaliśmy się zewnętrznymi bibliotekami. Wszystkie operacje bazują na wbudowanych funkcjach języka.

Praca była tworzona w zespole czteroosobowym zatem konieczne było zastosowanie systemu kontroli wersji i wybór padł na Git. Wykorzystaliśmy platformę GitHub, aby przechowywać nasz kod na zdalnym repozytorium w celu wspólnego tworzenia i omawiania decyzji projektowych.

5.2. Struktura projektu

Na nasze rozwiązanie składają się następujące klasy:

- main.py - klasa główna startowa odpowiadająca za wykonywanie głównej pętli operacji, wywoływania reszty funkcji oraz pobierania argumentów od użytkownika.
- crypt.py oraz crypt_block.py- zawiera funkcje odpowiedzialne za szyfrowanie i odszyfrowywanie.
- lib.py - klasa pomocnicza (biblioteka), która zawiera w sobie wszystkie metody pomocnicze potrzebne do przeprowadzenia operacji szyfrowania i deszyfrowania.
- operators.py - klasa posiadająca implementację wszystkich operatorów, które umożliwiają wykonywanie operacji na bitach w rozwiązaniu.
- convert.py - klasa odpowiadająca za przetwarzanie obustronne ciągu znaków (między reprezentacją binarną, a heksadecymalną oraz heksadecymalną, a ASCII).

5.3. Implementacja rozwiązania

W tej sekcji przedstawione zostały główne funkcje odpowiedzialne za szyfrowanie i deszyfrowanie. W kodzie źródłowym znajdują się komentarze, które wyjaśniają sposób działania reszty zaimplementowanych części. Ze względu na znaczną liczbę linii kodu pomijamy opis niektórych z nich w tym dokumencie.

```
from convert import *
from crypt import *
```

```
import sys

# Number of bits in the key
N_KEY_BITS = 128

if __name__ == "__main__":
    KEY = from_hex(sys.argv[1])
    PLAINTEXT = from_hex(sys.argv[2])
    CIPHERTEXT = encrypt(PLAINTEXT, KEY)
```

Listing 13. Funkcja główna programu - szyfrowanie.

```
from convert import *
from crypt import *
import sys

# Number of bits in the key
N_KEY_BITS = 128

if __name__ == "__main__":
    KEY = from_hex(sys.argv[1])
    CIPHERTEXT = from_hex(sys.argv[2])
    PLAINTEXT = decrypt(CIPHERTEXT, KEY)
```

Listing 14. Funkcja główna programu - deszyfrowanie.

Dla poprawności rozwiązania zostaje dodany również padding w przypadku, gdy wprowadzone przez użytkownika dane są krótsze niż 128 bit. Wywołane funkcje wyglądają następująco:

```
def get_128bit_blocks(TEXT):
    text_blocks = []
    import math
    loops_count = math.ceil(len(TEXT) / 128)

    for i in range(loops_count-1):
        text_blocks.append(LEFT(TEXT, 128))
        TEXT = TEXT[128:]

    for i in range(128-len(TEXT)):
        TEXT += b'0'

    text_blocks.append(TEXT)

    return text_blocks

```

```

def encrypt(plaintext, key):
    plaintext_blocks = get_128bit_blocks(plaintext)
    ciphertext = b''
    for i in plaintext_blocks:
        ciphertext += encrypt_block(i, key)
    return ciphertext


def decrypt(ciphertext, key):
    ciphertext_blocks = get_128bit_blocks(ciphertext)
    plaintext = b''
    for i in ciphertext_blocks:
        plaintext += decrypt_block(i, key)
    return plaintext
```

Listing 15. Padding, szyfrowanie i deszyfrowanie.

5.4. Instrukcja użytkowania

W celu poprawnego uruchomienia programu konieczne jest jego pobranie z repozytorium i zapisanie na dysku. Następnie należy przejść do folderu src, w którym znajduje się plik główny main.py. Wywołanie polecenia 'python main.py PLAINTEXT KEY' (za PLAINTEXT (128 bit) i KEY (192, 256 bit) należy podstawić dowolny ciąg w postaci hexadecymalnej) powoduje uruchomienie rozwiązania.

6. Podsumowanie

Zwiększenie liczby połączeń w sieci powoduje rosnącą konieczność zabezpieczenia danych przed niepowołanym dostępem. Zapewnienie wysokiego poziomu bezpieczeństwa przy optymalnym czasie operacji osiągane są dzięki korzystaniu z szyfrowania symetrycznego w trybie blokowym, które jest jednym z fundamentalnych segmentów kryptografii.

Omówiony przez nas krypto-system Camellia, mimo iż został opracowany ponad dwadzieścia lat temu, to jest uważany za nowoczesny i bezpieczny szyfr w pełni przystosowany do współczesnych wymagań. Jako szyfr blokowy o 128-bitowym rozmiarze bloku i trzech możliwych rozmiarach klucza (128, 192, 256 bit) sprawdza się odpowiednio dla różnych zastosowań. Nawet przy użyciu opcji najmniejszego rozmiaru klucza, uważa się, że złamanie go poprzez atak brute-force na klucze przy użyciu obecnej technologii jest niewykonalne.

W tej pracy udało nam się omówić początki i powody powstania systemu Camellia. Przeanalizowany został sposób implementacji i specyfikacja algorytmu. Porównaliśmy wydajność i tryb pracy Camellii do najpopularniejszego systemu jakim jest AES. Na podstawie dostępnej dokumentacji i artykułów naukowych zbadaliśmy bezpieczeństwo algorytmu. Przeprowadzona została także kryptoanaliza wraz z odnotowaniem ataków jakie były przeprowadzone na ten krypto-system. Uważamy, że opisany przez nas algorytm Camellia jest równie dobrym wyborem jak rozpowszechniony i popularny AES. W szczególnych przypadkach może być niezastąpiony, a brak znacznej rozpoznawalności i zrozumienia systemu, może być dodatkowym atutem pod względem bezpieczeństwa.

Z powodzeniem udało nam się zaimplementować kryptosystem Camellia. Nasze rozwiązanie zostało oparte o informacje z oficjalnej dokumentacji, czyli RFC 3713. Wykorzystany w tym celu został popularny język programistyczny Python. Wszystkie kolejne kroki działania systemu wraz z przykładami zostały zamieszczone w tym dokumencie. Kod źródłowy został udostępniony prowadzącemu w celu oceny.

We współczesnej technologii szyfrowanie symetryczne wciąż pełni niezwykle ważną rolę. Wraz z szyfrowaniem asymetrycznym zapewniania bezpieczeństwo i poufność podczas komunikacji użytkownika w sieci. Szczególnie ważne jest zwrócenie uwagi na tryb blokowy szyfrowania symetrycznego, który jest aktualnie powszechnie stosowany. Dzięki swojej wydajności i optymalizacji zapewnia użytkownikowi możliwość swobodnej i wydajnej wymiany danych. Camellia jest skutecznym i sprawdzonym rozwiązaniem, które w szczególnych przypadkach może stanowić ciekawą alternatywę dla bardziej rozpowszechnionych systemów.

Bibliografia

[1] *NTT Social Informatics Laboratories Information Security Technology Research Project*, Dostęp zdalny (18.12.2022): https://info.isl.ntt.co.jp/crypt/eng/camellia/technology/reference.html.

[2] M. Matsui, S. Moriai i J. Nakajima, *A Description of the Camellia Encryption Algorithm*, RFC 3713, kw. 2004. DOI: 10.17487/RFC3713. adr.: https://www.rfc-editor.org/info/rfc3713.

[3] *Camellia - SZYFR BLOKOWY Z KLUCZEM SYMETRYCZNYM*, Dostęp zdalny (18.12.2022): http://www.crypto-it.net/pl/symetryczne/camellia.html.

[4] ., S. Moriai i M. Kanda, *Addition of Camellia Cipher Suites to Transport Layer Security (TLS)*, RFC 4132, lip. 2005. DOI: 10.17487/RFC4132. adr.: https://www.rfc-editor.org/info/rfc4132.

[5] Wikipedia, *Camellia (cipher) — Wikipedia, The Free Encyclopedia*, http://en.wikipedia.org/w/index.php?title=Camellia%20(cipher)&oldid=1117477882, [Online; accessed 18-December-2022], 2022.

[6] *Analysis Of Camelia*, Dostęp zdalny (18.12.2022): Zalacznik:AnalaysisOfCamelia.pdf.

[7] *Improved zero-correlation linear cryptanalysis of reduced-round Camellia under weak keys. IET Information Security*, Dostęp zdalny (18.12.2022): https://www.researchgate.net/publication/282895646_Improved_zero-correlation_linear_cryptanalysis_of_reduced-round_Camellia_under_weak_keys.

[8] *New impossible differential cryptanalysis ofreduced-round camellia*, Dostęp zdalny (18.12.2022): https://eprint.iacr.org/2011/017.pdf.

[9] *New observations on impossible differentialcryptanalysis of reduced-round camellia*, Dostęp zdalny (18.12.2022): https://www.iacr.org/archive/fse2012/75490090/75490090.pdf.

[10] *Linear Hulls with Correlation Zero and LinearCryptanalysis of Block Ciphersa*, Dostęp zdalny (18.12.2022): https://eprint.iacr.org/2011/123.pdf.