Własny schemat szyfrowania z uwierzytelnieniem oparty o sieć Feistela w trybie pracy GCM

Katarzyna Leniec, Karol Suchecki, Oskar Werner

25 marca 2024

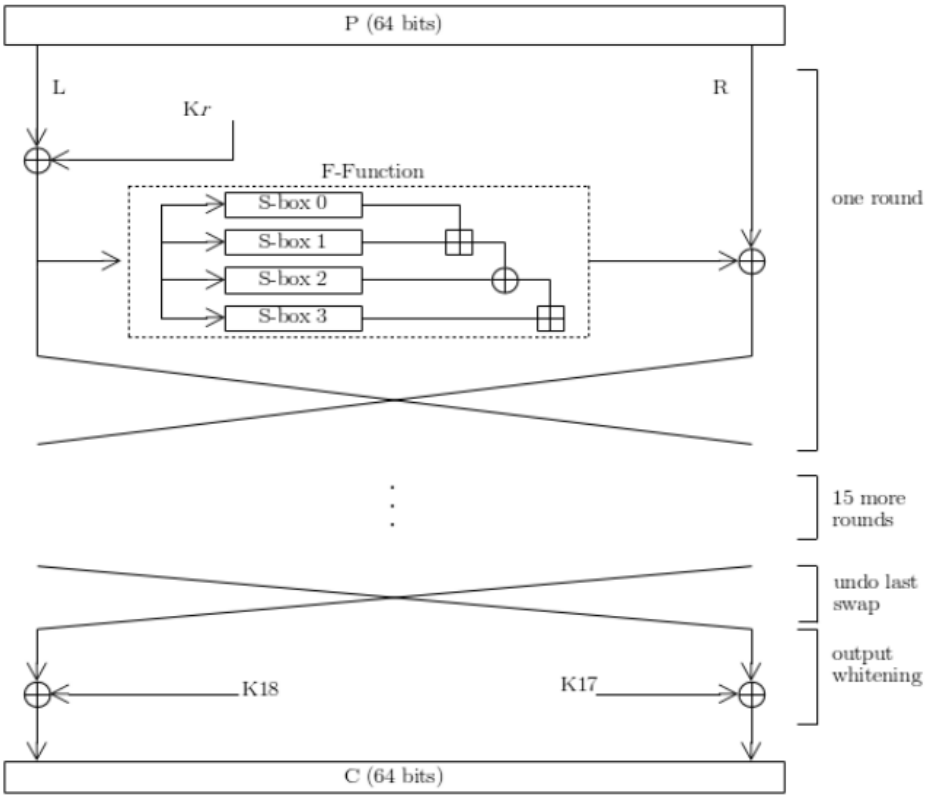
Streszczenie

W poniżej pracy prezentujemy modyfikację algorytmu Blowfish użytego w trybie pracy GCM w celu zapewnienia możliwości uwierzytelnienia wiadomości. Przedstawiamy opis skonstruowanego algorytmu MiNIfish wraz opisem wszystkich potrzebnych funkcji, przypominamy działanie trybu pracy GCM oraz uzasadniamy podjęte wybory konstrukcyjne.

Spis treści

1 Ogólny opis algorytmu szyfrowania MiNIfish

Algorytm MiNIfish przedstawiony w pracy jest szyfrem blokowym przyjmującym 64-bitowe bloki danych wraz z kluczem K o długości od 32 do 576 bitów, na podstawie którego generowane są klucze rundowe oraz *S-boxy*. Co ważne, przy każdej zmianie klucza K należy na nowo wygenerować klucze rundowe wraz ze skrzynkami podstawieniowymi. Dokładny opis generowania kluczy K_i, *S-boxów* oraz innych wykorzystywanych funkcji znajduje się w kolejnym rozdziale. Algorytm ma strukturę sieci Feistela z 16 rundami. Na początku dzielimy tekst jawny na dwie, równe, 32-bitowe części (oznaczmy jako L oraz R). W i-tej rundzie wykonujemy operację XOR na L oraz K_i, gdzie K_i jest kluczem odpowiadającym i-tej rundzie. Następnie wynik operacji XOR jest przekazywany jako R do następnej rundy, jednocześnie ta sama wartość przekazywana jest do funkcji F, której wynik jest poddawany operacji XOR z R, a następnie przekazywany jako L do następnej rundy, co kończy i-tą rundę. Po wykonaniu wszystkich rund zamieniamy L z R oraz dodajemy ostatnie klucze K_{17} oraz K_{18} odpowiednio do R oraz L. Konkatenacja $L||R$ zwraca szyfrogram. Szyfrogram pozyskany dzięki użyciu naszego algorytmu może być odszyfrowany przy pomocy tego samego algorytmu z odwróconą kolejnością kluczy.



Rysunek 1: Schemat opisywanego szyfrowania

Przedstawiamy również pseudokod opisujący strukturę sieci Feistela.

Algorithm 1 Sieć Feistela

Require: P: 64-bitowy tekst jawny; $K_1, ..., K_{18}$: 32-bitowe klucze rundowe
Ensure: C: 64-bitowy szyfrogram

1: $(L, R) \leftarrow P$ {Podziel P na dwie równe częsci}
2: **for** $i \in \{1, ...16\}$ **do**
3: $\quad L \leftarrow L \oplus K_i$
4: $\quad R \leftarrow F(L) \oplus R$
5: $\quad (L, R) \leftarrow (R, L)$
6: **end for**
7: $(L, R) \leftarrow (R, L)$
8: $R \oplus K_{17}$
9: $L \oplus K_{18}$
10: **return** (L, R)

2 Opis funkcji wykorzystanych w algorytmie MiNIfish

Funkcja F działa na 32-bitowym bloku danych. W pierwszej kolejności dzieli wejściowy blok na 4 równe, 8-bitowe bloki danych, na których stosuje się odpowiednie przekształcenia $\mathbb{Z}_2^8 \to \mathbb{Z}_2^{32}$ zwane *S-boxami*. Następnie wyniki są poddawane operacją XOR (oznaczone jako $\oplus$) oraz mnożeniu w ciele $GF(2^{32})$ z wielomianem minimalnym $p(x) = x^{32} + x^{15} + x^9 + x^7 + x^4 + x^3 + 1$ (oznaczone jako $\boxplus$) zgodnie z Rysunkiem 3.

Algorithm 2 Funkcja F

Require: X: 32-bitowy ciąg
Ensure: Y: 32-bitowy ciąg
  $(a, b, c, d) \leftarrow X$
  return $((S1[a] \boxplus S2[b]) \oplus S3[c]) \boxplus S[d])$

Proces generowania *S-boxów* oraz kluczy K_i opisany jest przez poniższy pseudokod. Warto zwrócić uwagę, że w pseudokodzie traktujemy klucze K_i oraz listy $S1[], ..., S4[]$ jako zmienne globalne, zatem do wywołania algorytmu szyfrowania przekazujemy wyłącznie $64-$bitowy tekst jawny. Ponadto korzystamy z szeroko znanej funkcji $SHAKE256$ jako generatora losowości.

Algorithm 3 Generowanie kluczy rundowych i *S-boxów*

Require: K: Klucz o długości od 32 do 576 bitów
Ensure: $K_1, ..., K_{18}$: 18 32-bitowych kluczy rundowych; $S1[], S2[], S3[], S4[]$: 4 *S-boxy*, 256-elementowe tablice
1: $Rand \leftarrow SHAKE256(K, 16672)$ {$Rand$ przyjmuje 16672-bitowy wynik operacji $SHAKE256(K, 16672)$}
2: $(K_1, ..., K_{18}, S1[], S2[], S3[], S4[]) \leftarrow Rand$
3: $K' \leftarrow (K, K, K, ...)$ {Powiel K do długości $18 \cdot 32$ bitów, czyli skonkatenuj K z samym sobą $\lfloor \frac{576}{|K|} \rfloor$ razy, a następnie skonkatenuj wynik z pierwszymi $32 - \lfloor \frac{576}{|K|} \rfloor$ bitami K }
4: $(K_1, ..., K_{18}) \leftarrow (K_1, ..., K_{18}) \oplus K'$
5: $T \leftarrow (0...0)$ {Inicjalizacja tekstu jawnego 64 bitami zer}
6: **for** $i \in \{1, ..., 9\}$ **do**
7: $T \leftarrow \text{MiNIfish}(T)$
8: $(K_{2i-1}, K_{2i}) \leftarrow T$
9: **end for**
10: **for** $j \in \{1, ..., 4\}$ **do**
11: **for** $i \in \{0, ..., 127\}$ **do**
12: $T \leftarrow \text{MiNIfish}(T)$
13: $(Sj[2i], Sj[2i + 1]) \leftarrow T$
14: **end for**
15: **end for**
16: **return** $(K_1, ..., K_{18}, S1[], S2[], S3[], S4[])$

W MiNIfish oraz w GCM korzysta się z mnożenia w ciele GCM. Poniżej przedstawiamy pomocne funkcje do implementacji

```
def byte2bin(b):
    return ''.join(format(byte, '08b') for byte in b)

def deg(a):
    deg = len(bin(a)[2:])-1
    return deg

def gfmul(a, b, m):
    p = 0
    while a > 0:
        if a & 1:
            p = p ^ b

        a = a >> 1
        b = b << 1

        if deg(b) == deg(m):
            b = b ^ m

    return p
# dla mnozenia w MiNIfish wstaw
# m = 340282366920938463463374607431768211591 # to jest p(2) dla p(x) =↩
    x^128+x^7+x^2+x+1
# dla mnozenia w gcm wstaw w gfmul
# m = 4295000729   # to jest p(2) dla p(x) = x^32 + x^15 + x^9 + x^7 + ↩
   x^4 + x^3 + 1
```

3 GCM

Dzięki trybowi Galois/Counter (GCM) MiNIfish może być szyfrem AEAD. Nazwa GCM bierze się stąd, że operujemy w ciele Galois $GF(2^{128})$ z wielomianem minimalnym równym $p(x) = x^{128} + x^7 + x^2 + x + 1$. Poza zapewnieniem uwierzytelnienia GCM pozwala na zaszyfrowanie wiadomości większej niż mógłby sam MiNIfish. Poniżej przedstawiamy opis algorytmu, schemat GCM wraz z pseudokodem, i jak sprawdzić, czy wiadomość nie została zmodyfikowana po przesłaniu.

Dane do GCM:

- K - tajny klucz szyfru blokowego
- IV - wektor inicjalizacyjny
- P - tekst jawny
- AAD - dodatkowe dane uwierzytelniające

Długości powyższych danych muszą spełniać następujące wymagania:

- K - od 32 do 576 bitów
- IV - 4 bajty. W algorytmie dodaje się jeszcze cztery bajty zer na końcu IV
- P - od 0 do $2^{39} - 256$ bitów
- AAD - od 0 do $2^{64} - 1$ bitów

3.1 Wysokopoziomowy opis algorytmu GCM

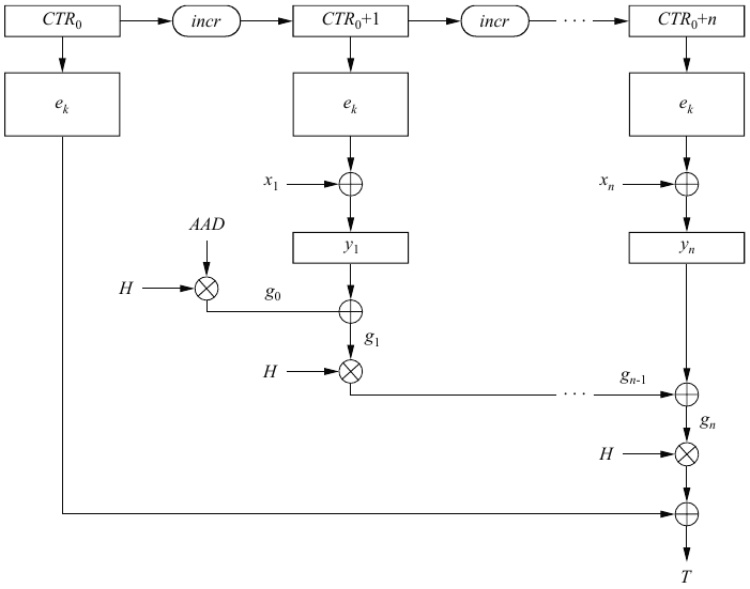
3.1.1 Część, która szyfruje wiadomość

Niech x to będzie tekst jawny, a $x_1, \ldots, x_n$ to kolejne 64-bitowe części tej wiadomości oraz AAD to informacje dodatkowe oraz e_k to jest szyfrowanie MiNIfish z kluczem k. Wtedy

- Niech $CTR_0 = IV$ i dla każdych 64-bitów wiadomości policz $CTR_i = CTR_{i-1} + 1$.
- Policz szyfrogramy $y_i = e_k(CTR_i) \oplus x_i,\ i \geqslant 1$

3.1.2 Część, która uwierzytelnia wiadomość

- Policz stałą $H = e_k(0)$
- Policz $g_0 = AAD \times H$ w ciele $GF(2^{128})$
- Policz $g_i = (g_{i-1} \oplus y_i) \times H, 1 \leqslant i \leqslant n$
- Policz ostateczny tag uwierzytelnienia $T = (g_n \times H) \oplus e_k(CTR_0)$



Rysunek 2: Schemat GCM

3.2 Co zwraca program

Program zwraca listę z następującymi wartościami. Pierwszy element to jest zaszyfrowana wiadomość. Druga wartość to jest wartość T, jest ona potrzebna do uwierzytelnienia. Ostatnia wartość to AAD które zostało wpisane wcześniej. Wytłumaczymy w kolejnej sekcji, co można zrobić z tymi wartościami.

3.3 Jak sprawdzić, czy przy danej wiadomości nikt nie manipulował

W tym celu po odebraniu wiadomości trzeba zrobić następujące rzeczy:

1. Zapisz zmienną T którą dostałeś razem z wiadomością na boku.
2. Odszyfruj wiadomość, szyfrując zaszyfrowaną wiadomość. GCM działa tak, ponieważ szyfrujemy countery, a nie same wiadomości, a w samym algorytmie używamy funkcji xor i właśnie dzięki własności funkcji xor odzyskujemy tekst jawny.
3. Zaszyfruj właśnie odszyfrowaną wiadomość i popatrz na zmienną T. Jeżeli jest taka sama jak w pierwszym kroku, to wiadomość nie została zmodyfikowana.

3.4 Pseudokod algorytmu GCM

Algorithm 4 Szyfrowanie wiadomości za pomocą GCM

Require: *keys*: lista kluczy wygenerowana przez funkcję *Generowanie kluczy rundowych i S-boxów*, IV: wektor inicjalizacji długości 4 bajtów, P: tekst jawny od 0 do $2^{39}-256$ bitów, AAD od 0 do $2^{64}-1$ bitów, *encrypt*: MiNIfish, który szyfruje 64 bity danych.

$H \leftarrow encrypt(keys, b'00000000)$ {zaszyfrowanie 64 bitów zer}
$polynomial \leftarrow 340282366920938463463374607431768211591$ {Stała w algorytmie}
$IV \leftarrow IV||b'0000$ {Konkatenacja IV z czterema bajtami zer}
$(p_1 \dots p_n) \leftarrow P$ {Podzielenie wiadomości na 64-bitowe części, z dokładnoscią do ostatniego bloku}
if $|p_n| = 64$ **then**
    $p_{n+1} \leftarrow 10...0$ {Przypisanie 64-bitowego bloku składającego się 1 i 63 zer} $(p_1 \dots p_n).append(p_{n+1})$
end if
if $|p_n| = 63$ **then**
    $p_n \leftarrow p_n||1$
end if
if $|p_n| < 63$ **then**
    $p_n \leftarrow p_n||10...0$
end if
$amount \leftarrow length(p_1 \dots)$ {Liczba 64-bitowych wiadomości}
$counter_0 \leftarrow IV$
for $i \in \{1, ..., amount\}$ **do**
    $counter_i \leftarrow counter_{i-1} + 1$ {Utworzenie counterów}
end for
for $i \in \{0, ..., amount\}$ **do**
    $cipher_i \leftarrow encrypt(keys, counter_i) \oplus p_i$ {Zaszyfrowanie counterów}
end for
$g_0 \leftarrow gfmul(H, AAD, polynomial)$
for $i \in \{1, ..., amount\}$ **do**
    $g_i \leftarrow gfmul(g_{i-1} \oplus cipher_i, H, polynomial)$ {Utworzenie tagów}
end for
$T \leftarrow gfmul(g_{amount}, H, polynomial) \oplus cipher_0$ {Utworzenie tagów}
return $Cipher, T, AAD$

3.5 Przykłady zaszyfrowanych wiadomości

```
key = b'klucz' (klucz ktorym generuje sie sboxy oraz klucze rundowe do MiNIfisha)
iv = b'0123'
AAD = b'Karol Kasia Oskar'

text1 = b'matematyka jest super'
gcm(key, text1, iv, AAD) ==
[b'w\xe9\xb7\x85\x85\x9a‘a\xb5\xb9\xc5\xe0\xc4\x12\x8fg\x90\x86\x0e\xff\xe6\xfe3\xb1',
 172611014228878535600893946448906414878,
 b'Karol Kasia Oskar']

text2 = b'cyberbezpieczenstwo to dobra zabawa :)'
gcm(key, text2, iv, AAD) ==
[b'y\xf1\xa1\x85\x9a\x99qb\xae\xb1\x80\xe9\xdb\x04\x954\x97\x84\x11\xba\xe0\x11\x13\xd5\xc0\xaaG!\xe9\x08\xf3\xc9o\x86\xef\xdf\xd4U\xddl',
 215514916959619062362211946423245933156,
 b'Karol Kasia Oskar']
```

4 Uzasadnienie wyboru konstrukcji

Główną zaletą algorytmów opartych o schemat sieci Feistela jest ich prostota i bezpieczeństwo. Szyfrogram pozyskany dzięki użyciu naszego algorytmu może być odszyfrowany przy pomocy tego samego algorytmu z odwróconą kolejnością kluczy. Ponadto konstrukcję naszego algorytmu oparliśmy o algorytm Blowfish, który został przebadany chociażby pod kątem niepowiązania tekstu jawnego z szyfrogramem [6], czy spełniania efektu lawinowego [7]. W naszym schemacie szyfrowania z uwierzytelnieniem pseudolosowość została zapewniona przez użycie algorytmu SHAKE256 należącego do rodziny funkcji SHA3, która została opisana i przebadana w roku 2015, a wyniki powyższych prac zostały opublikowane przez NIST w [1]. Uwierzytelnianie wiadomości realizujemy poprzez zastosowanie algorytmu w trybie pracy GCM, który, choć debata na ten temat trwa od niemalże 20 lat (patrz [2], [3], [4], [5]), jest uznawany jest nie tylko za bezpieczny, lecz także efektywny i szybki tryb szyfrowania dzięki możliwości równoległego szyfrowania bloków tekstu jawnego. Wspomniany tryb szyfrowania jest szeroko stosowany we współczesnej komunikacji, wykorzystuje się go chociażby w protokołach TLS w wersji 1.2 i 1.3 (protokół bezpiecznego połączenia między hostem a serwerem), SSH (zdalne połączenia z hostem/serwerem) czy WPA3-Enterprise (sieci bezprzewodowe).

5 Analiza bezpieczeństwa

5.1 Sprawdzenie własności *S-boxów*

Dla losowej liczby całkowitej r z przedziału $5 - 500$ wygenerujemy r kluczy, gdzie każdy z kluczy generuje 4 *S-boxy*. Następnie sprawdzimy i przeanalizujemy niektóre własności (zupełność, branch number oraz efekt lawinowy) tych *S-boxów*. Z użyciem napisanych wcześniej funkcji w języku Python otrzymaliśmy następujący wynik:
Wygenerowano 307 kluczy. Z 1228 wygenerowanych *S-boxów* wszystkie są zupełne, ale żaden z nich nie spełnia efektu lawinowego. Zatem przekształcenia te zapewniają konfuzję, ale nie zapewniają dyfuzji. Przy czym dla 751 *S-boxow* wartość branch number wynosi 2, a dla 477 *S-boxów* wartość ta wynosi 1. Branch number jest miarą "wielkości" zmiany wyjścia w *S-boxie* spowodowanej zmianą jednego bitu na wejściu. Wyższa wartość branch number oznacza większą różnorodność zmian na wyjściu S-boxa w odpowiedzi na zmiany wejścia, co jest pożądane w kryptografii. Ponieważ nasze przekształcenia nie zapewniają dyfuzji to wartość branch number jest mniej istotna, ponieważ zmiany na wejściu nie są efektywnie rozprzestrzeniane na wyjście.

5.2 Testy losowości

Dla 1000 różnych, wygenerowanych losowo wejść, tj. key, plaintext, iv, AAD, przeprowadziliśmy następujące testy losowości: test częstości, blokowy test częstości, test długich serii jedynek oraz runs test. Dokładniej, połączyliśmy szyfr oraz tag uwierzytelniający (które zwróciła funkcja *gcm()*), przekonwertowaliśmy otrzymany tekst na ciąg liczb binarnych oraz za pomocą napisanych przez nas funkcji w języku Python sprawdziliśmy

losowość otrzymanego ciągu. W wyniku naszego badania otrzymaliśmy jaki procent z 1000 otrzymanych wyjść wykazuje losowość. Wyniki były następujące:

- test częstości: 99.1%
- blokowy test częstości: 99.3%
- test długich serii jedynek: 98.5%
- runs test: 0%

Podsumowując, wyniki testów sugerują, że przeprowadzane szyfrowanie generuje zaszyfrowane dane, które wykazują wysoki stopień losowości z punktu widzenia testów częstościowych i testu długich serii, ale mogą być odstające pod względem testu runs. Test runs jest używany do sprawdzenia, czy sekwencje danych zawierają zbyt dużo lub zbyt mało serii (ciągów) tego samego bitu. Wynik 0% sugeruje, że zaszyfrowane dane mogą zawierać niepożądane wzorce lub struktury, które nie są zgodne z oczekiwaniami losowości.

5.3 Próba przeprowadzenia ataku liniowego

Główna trudność z przeprowadzeniem ataku liniowego na nasz algorytm jest związana z losowym generowaniem *S-boxów*. Zdecydowaliśmy się na podejście polegające na losowym wygenerowaniu 500 kluczy o losowej długości z przedziału $1 - 100$ bajtów. Następnie, po wygenerowaniu odpowiednich *S-boxów*, wybraliśmy 1 klucz, dla którego, w co najmniej 3 *S-boxach* wystąpiło co najmniej 7 wartości ± 4 (przy *S-boxie*, którego dziedziną jest $\mathbb{Z}_2^8$, wartość ± 4 musi wystąpić jako jedyna wartość w danej kolumnie LAT-profilu, co oznacza, że pewna aproksymacja liniowa zawsze zachodzi) . Każdy taki *S-box* zapewniał nam co najmniej 7 równań liniowych postaci $\alpha X \oplus \beta Y = 1$ lub $\alpha X \oplus \beta Y = 0$, dzięki czemu mogliśmy szukać zależności między bitami wejściowymi a bitami wyjściowymi funkcji F opisanej algorytmem 2.

Mając zebrane klucze i odpowiadające im *S-boxy*, które chcemy badać, za pomocą prostego skryptu wygenerowaliśmy układy aproksymacji liniowych, które potem analizowaliśmy ręcznie- każdy osobno, w zależności od uzyskanych równań. Celem ataku było znalezienie zależności miedzy bitami wejścia i wyjścia dla funkcji F, nie zaś odnalezienie części/całości klucza- to uważamy za zbyt trudne, ze względu na zastosowanie generatora liczb pseudolosowych. Niesety nie udało się przeprowadzić udanego ataku. Wyniki naszych prac przedstawiamy w poniższej podpodsekcji (:)). Wykorzystane skrypty wraz z badanymi przekształceniami dołączamy również w oddzielnym załączniku.

5.3.1 Próba ataku

Poniżej przedstawiamy zestaw danych (klucz, 4 LAT-profile) wraz z odpowiadającymi im równaniami:

```
klucz = b'\xd4\x9e\xa6\xfbL\xc2}\x82\x1a+7\x8d\x91\xdf\xb2\x1f9\xfd'
[4, -1, -1, 0, 1, 0, 0, 1, -1, 0, 0, -1, 0, -3, 1, 0, -4, 1, 1, 0, -1, 0, 0, -1, 1, 0, 0, 1, 0, 3, -1, 0]
[0, -1, -1, 0, 1, 0, -4, 1, -1, 0, 0, 3, 0, 1, 1, 0, 0, 1, 1, 0, -1, 0, 4, -1, 1, 0, 0, -3, 0, -1, -1, 0]
[0, -1, 1, -2, -1, -2, 0, 1, 3, 0, -2, 1, 2, -1, 1, 0, 0, 1, -1, 2, 1, 2, 0, -1, -3, 0, 2, -1, -2, 1, -1, 0]
[0, -1, 1, 2, -1, 2, 0, 1, -1, 0, -2, 1, 2, -1, -3, 0, 0, 1, -1, -2, 1, -2, 0, -1, 1, 0, 2, -1, -2, 1, 3, 0]
[0, 1, -3, 0, -1, 0, 0, -1, 1, 0, 0, 1, 0, -1, -1, 4, 0, -1, 3, 0, 1, 0, 0, 1, -1, 0, 0, -1, 0, 1, 1, -4]
[0, 1, 1, 0, 3, 0, 0, -1, 1, -4, 0, 1, 0, -1, -1, 0, 0, -1, -1, 0, -3, 0, 0, 1, -1, 4, 0, -1, 0, 1, 1, 0]
[0, 1, -1, 2, 1, -2, 0, 3, 1, 0, -2, -1, -2, 1, -1, 0, 0, -1, 1, -2, -1, 2, 0, -3, -1, 0, 2, 1, 2, -1, 1, 0]
[0, -3, -1, 2, 1, -2, 0, -1, 1, 0, 2, -1, 2, 1, -1, 0, 0, 3, 1, -2, -1, 2, 0, 1, -1, 0, -2, 1, -2, -1, 1, 0]

[4, 0, -1, 1, 1, 1, 0, 2, 0, 2, 1, 1, -1, 1, 4, 0, -4, 0, 1, -1, -1, -1, 0, -2, 0, -2, -1, -1, 1, -1, -4, 0]
[0, 2, -1, -1, 1, -1, 0, 0, 0, 0, 1, -1, -1, -1, 0, 2, 0, -2, 1, 1, -1, 1, 0, 0, 0, 0, -1, 1, 1, 1, 0, -2]
[0, -2, 1, 1, -1, 1, 0, 0, 0, 0, -1, 1, 1, 1, 0, -2, 0, 2, -1, -1, 1, -1, 0, 0, 0, 0, 1, -1, -1, -1, 0, 2]
[0, 0, 1, 3, 3, -1, 0, 2, 0, 2, 3, -1, 1, 3, 0, 0, 0, 0, -1, -3, -3, 1, 0, -2, 0, -2, -3, 1, -1, -3, 0, 0]
[0, -2, -1, -1, 1, -1, -4, 0, -4, 0, 1, -1, -1, -1, 0, -2, 0, 2, 1, 1, -1, 1, 4, 0, 4, 0, -1, 1, 1, 1, 0, 2]
[0, 0, -1, 1, 1, 1, 0, -2, 0, -2, 1, 1, -1, 1, 0, 0, 0, 0, 1, -1, -1, -1, 0, 2, 0, 2, -1, -1, 1, -1, 0, 0]
[0, 0, 1, -1, -1, -1, 0, 2, 0, 2, -1, -1, 1, -1, 0, 0, 0, 0, -1, 1, 1, 1, 0, -2, 0, -2, 1, 1, -1, 1, 0, 0]
[0, -2, -3, 1, -1, -3, 0, 0, 0, 0, -1, -3, -3, 1, 0, -2, 0, 2, 3, -1, 1, 3, 0, 0, 0, 0, 1, 3, 3, -1, 0, 2]

[4, -1, 2, -1, 1, 0, 1, 2, -1, 0, -1, 2, 0, 1, -2, 1, -4, 1, -2, 1, -1, 0, -1, -2, 1, 0, 1, -2, 0, -1, 2, -1]
[0, 1, 2, 1, 1, 2, 1, 0, -1, 2, -1, 0, 0, -1, 2, -1, 0, -1, -2, -1, -1, -2, -1, 0, 1, -2, 1, 0, 0, 1, -2, 1]
[0, 3, -2, 3, 1, 0, 1, 2, 1, -2, 1, 0, 2, -1, 0, -1, 0, -3, 2, -3, -1, 0, -1, -2, -1, 2, -1, 0, -2, 1, 0, 1]
[0, 1, 2, 1, 1, 2, 1, 0, 1, 0, 1, -2, -2, 1, 0, 1, 0, -1, -2, -1, -1, -2, -1, 0, -1, 0, -1, 2, 2, -1, 0, -1]
[0, 1, 0, -1, -1, 0, 1, 0, -1, -2, 1, -2, -2, 1, -2, -1, 0, -1, 0, 1, 1, 0, -1, 0, 1, 2, -1, 2, 2, -1, 2, 1]
[0, -1, 0, 1, 3, -2, 1, -2, -1, 0, -3, 0, -2, -1, -2, 1, 0, 1, 0, -1, -3, 2, -1, 2, 1, 0, 3, 0, 2, 1, 2, -1]
[0, 1, 0, -1, -1, 0, 1, 0, -3, 0, -1, 0, 0, -1, 0, -3, 0, -1, 0, 1, 1, 0, -1, 0, 3, 0, 1, 0, 0, 1, 0, 3]
[0, -1, 0, 1, -1, 2, -3, 2, 1, -2, -1, -2, 0, -3, 0, -1, 0, 1, 0, -1, 1, -2, 3, -2, -1, 2, 1, 2, 0, 3, 0, 1]

[4, -1, -3, 0, 1, 0, 0, -1, 0, 1, -1, 0, -1, 0, 0, 1, -4, 1, 3, 0, -1, 0, 0, 1, 0, -1, 1, 0, 1, 0, 0, -1]
[0, -1, 1, 0, 1, 0, 0, -1, 0, 1, -1, 0, 3, 0, -4, 1, 0, 1, -1, 0, -1, 0, 0, 1, 0, -1, 1, 0, -3, 0, 4, -1]
[0, 1, -1, 0, -1, 0, 0, 1, 4, -1, -3, 0, 1, 0, 0, -1, 0, -1, 1, 0, 1, 0, 0, -1, -4, 1, 3, 0, -1, 0, 0, 1]
[0, 1, -1, 0, 3, 0, -4, 1, 0, -1, 1, 0, 1, 0, 0, -1, 0, -1, 1, 0, -3, 0, 4, -1, 0, 1, -1, 0, -1, 0, 0, 1]
[0, 1, -1, 0, -1, 0, 0, 1, 0, -1, 1, 0, 1, -4, 0, 3, 0, -1, 1, 0, 1, 0, 0, -1, 0, 1, -1, 0, -1, 4, 0, -3]
[0, -3, -1, 4, -1, 0, 0, 1, 0, -1, 1, 0, 1, 0, 0, -1, 0, 3, 1, -4, 1, 0, 0, -1, 0, 1, -1, 0, -1, 0, 0, 1]
[0, -1, 1, 0, 1, -4, 0, 3, 0, 1, -1, 0, -1, 0, 0, 1, 0, 1, -1, 0, -1, 4, 0, -3, 0, -1, 1, 0, 1, 0, 0, -1]
[0, -1, 1, 0, 1, 0, 0, -1, 0, -3, -1, 4, -1, 0, 0, 1, 0, 1, -1, 0, -1, 0, 0, 1, 0, 3, 1, -4, 1, 0, 0, -1]
```

Rysunek 3: Pierwsze analizowane dane- klucz i LAT-profile

Poniżej przedstawiamy równania pozyskane z powyższych LAT-profili. Zapis Z_j^i oznacza $j-$tą współrzędną wektora Z w $i-$tym *S-boxie*. X to wektor wejściowy, Y wyjściowy

po zastosowaniu *S-boxa*.

$$
\begin{aligned}
X_0^0 \oplus Y_1^0 \oplus Y_2^0 \oplus Y_4^0 &= 0 \\
X_2^0 \oplus Y_0^0 \oplus Y_1^0 \oplus Y_2^0 \oplus Y_3^0 &= 0 \\
X_0^0 \oplus X_2^0 \oplus Y_0^0 \oplus Y_3^0 \oplus Y_4^0 &= 0 \\
Y_4^0 &= 1 \\
X_0^0 \oplus Y_1^0 \oplus Y_2^0 &= 1 \\
X_2^0 \oplus Y_0^0 \oplus Y_1^0 \oplus Y_2^0 \oplus Y_3^0 \oplus Y_4^0 &= 1 \\
X_0^0 \oplus X_2^0 \oplus Y_0^0 \oplus Y_3^0 &= 1 \\
Y_1^1 \oplus Y_2^1 \oplus Y_3^1 &= 0 \\
X_2^1 \oplus Y_1^1 \oplus Y_2^1 \oplus Y_4^1 &= 0 \\
X_2^1 \oplus Y_3^1 \oplus Y_4^1 &= 0 \\
Y_4^1 &= 1 \\
Y_1^1 \oplus Y_2^1 \oplus Y_3^1 \oplus Y_4^1 &= 1 \\
X_2^1 \oplus Y_1^1 \oplus Y_2^1 &= 1 \\
X_2^1 \oplus Y_3^1 &= 1 \\
Y_4^2 &= 1 \\
X_0^3 \oplus Y_1^3 \oplus Y_2^3 \oplus Y_3^3 \oplus Y_4^3 &= 0 \\
X_1^3 \oplus Y_3^3 &= 0 \\
X_0^3 \oplus X_1^3 \oplus Y_1^3 \oplus Y_2^3 \oplus Y_4^3 &= 0 \\
X_2^3 \oplus Y_0^3 \oplus Y_2^3 \oplus Y_3^3 \oplus Y_4^3 &= 0 \\
X_0^3 \oplus X_2^3 \oplus Y_0^3 \oplus Y_1^3 &= 0 \\
X_1^3 \oplus X_2^3 \oplus Y_0^3 \oplus Y_2^3 \oplus Y_4^3 &= 0 \\
X_0^3 \oplus X_1^3 \oplus X_2^3 \oplus Y_0^3 \oplus Y_1^3 \oplus Y_3^3 &= 0 \\
Y_4^3 &= 1 \\
X_0^3 \oplus Y_1^3 \oplus Y_2^3 \oplus Y_3^3 &= 1 \\
X_1^3 \oplus Y_3^3 \oplus Y_4^3 &= 1 \\
X_0^3 \oplus X_1^3 \oplus Y_1^3 \oplus Y_2^3 &= 1 \\
X_2^3 \oplus Y_0^3 \oplus Y_2^3 \oplus Y_3^3 &= 1 \\
X_0^3 \oplus X_2^3 \oplus Y_0^3 \oplus Y_1^3 \oplus Y_4^3 &= 1 \\
X_1^3 \oplus X_2^3 \oplus Y_0^3 \oplus Y_2^3 &= 1 \\
X_0^3 \oplus X_1^3 \oplus X_2^3 \oplus Y_0^3 \oplus Y_1^3 \oplus Y_3^3 \oplus Y_4^3 &= 1
\end{aligned}
$$

W powyższych równaniach nie zauważyliśmy żadnych zależości pozwalających na sku-

teczny atak na całą funkcję F. Zauważmy, że w konstrukcji funkcji F wykorzystywane jest dwukrotnie mnożenie w ciele Galois, w którym wartość dowolnego bitu wyjścia zależy od wszystkich bitów wejściowych obu wektorów (wielomianów), co sprawia, że nawet dzięki znalezieniu pewnych zależności między współrzędnymi, nie jesteśmy w stanie wyrazić wzorem wartości bitów po wykonaniu mnożenia. Warto zauważyć, że zastosowanie tego mnożenia (zamiast na przykład ciągłego używania operacji XOR) wyraźnie wpływa na bezpieczeństwo na ataki liniowe,

Co ciekawe, gdy zliczaliśmy liczbę wystąpień czwórek w *S-boxach*, jedynymi liczbami wystąpień były: $1, 3, 7, 15$, przykładowo na 500 wygenerowanych kluczy (czyli 2000 *S-boxów*) 608 miało 1 wartość ± 4, 1149 miało 3 wystąpienia ± 4, 235 pojawień się 7 i tylko 8 wystąpień 15. Przy kolejnych generacjach kluczy rozkład (intuicyjnie) pozostawał podobny. Jest to obserwacja dalece niepokojąca i potencjalnie wskazująca na pewną podatność, jednak nie została przez nas wykorzystana. Może ona też wynikać z zastosowania wbudowanych funkcji pseudolosowych.

6 Atak algebraiczny

Ze względu na stosowanie funkcji pseudolosowych do generowania *S-boxów*, ich wartości, w oczywisty sposób, nie dają się wyrazić wzorami, uniemożliwiając wykonanie ataku algebraicznego.

7 Bibliografia

[1] Morris J. Dworkin *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*

[2] David A. McGrew, John Viega *The Security and Performance of the Galois/Counter Mode (GCM) of Operation*

[3] Tetsu Iwata, Keisuke Ohashi, Kazuhiko Minematsu *Breaking and Repairing GCM Security Proofs*

[4] Yuichi Niwa, Keisuke Ohashi, Kazuhiko Minematsu, Tetsu Iwata *GCM Security Bounds Reconsidered*

[5] Viet Tung Hoang, Stefano Tessaro, Aishwarya Thiruvengadam, Tetsu Iwata *The Multi-user Security of GCM, Revisited: Tight Bounds for Nonce Randomization*

[6] Al-Abiachi, Ashwak M. and Mahmood, Ramlan and Ahmad, Faudziah and S. Mechee, Mohammed *Randomness analysis on blowfish block cipher using ECB and CBC modes*

[7] Ashwak Alabaichi; Faudziah Ahmad; Ramlan Mahmod *Security analysis of blowfish algorithm*