Implementacja dwóch wybranych szyfrów strumieniowych i porównanie wydajności pomiędzy szyframi i istniejącymi implementacjami

Zaawansowane metody kryptografii i ochrony informacji

Autorzy: Tomasz Bocheński, Carlos Zaldivar Batista

Prowadzący: Albert Sitek

Semesetr 17Z

1 Wprowadzenie

Celem projektu jest implementacja dwóch szyfrów strumieniowych w wybranym przez siebie środowisku oraz porównanie własnych implementacji z implementacjami z już istniejących dla danego środowiska bibliotek.

Sprawdzenie poprawności wytworzonego oprogramowania odbędzie się przez porównanie wyników szyfrowania tych samych danych, z wykorzystaniem tych samych parametrów, przez obie implementacje - wyniki powinny być identyczne. Sprawdzenie wydajności implementacji odbędzie się przez mierzenie i porównanie czasu szyfrowania plików o rozmiarach 1000, 10 000, 100 000, 1 000 000, 4 000 000, 8 000 000 i 10 000 000 bajtów.

Do pokazania działania zaimplementowanych szyfrów zostanie stworzony prosty program z graficznym interfejsem umożliwiający:

1. wybranie algorytmu szyfrowania;
2. wprowadzenie niezbędnych danych;
3. wyświetlenie wyników w GUI lub zapisanie ich do pliku.

Po zaszyfrowaniu/odszyfrowaniu program tworzyć będzie logi tekstowe objaśniające przebieg algorytmów.

2 Opracowanie teoretyczne

W tym rozdziale opisano czym są szyfry strumieniowe oraz wymieniono wykorzystane w projekcie narzędzia.

2.1 Szyfry strumieniowe

Szyfry strumieniowe są typem szyfrów symetrycznych, w których tekst jawny jest zamieniany w szyfrogram przez zmodyfikowanie każdego znaku tekstu jawnego, przez odpowiadający mu znak ze strumienia klucza szyfrującego. Zazwyczaj znakami są pojedyncze bity a ich modyfikacja odbywa się przez przeprowadzenie operacji XOR.

Operacje przeprowadzane na tekście jawnym są więc bardzo proste, złożoną częścią szyfrów strumieniowych jest za to generowanie strumienia klucza. Strumień klucza jest zwykle tworzony przez kolejne manipulacje podanego przez użytkownika ziarna.

Zastosowanie dwukrotnie tego samego ziarna doprowadzi do wygenerowania takiego samego strumienia klucza, dlatego istotne jest by unikać wykorzystania identycznego ziarna do szyfrowania dwóch i więcej wiadomości. W tym celu ziarno dzieli się czasem na dwie części - tajny klucz znany tylko komunikującym się stronom oraz wektor inicjacyjny (IV), który przekazywany może być otwarcie. Wykorzystując ten sam klucz oraz różne IV, uzyskuje się różne ziarna.

Ponieważ szyfrogram powstaje zwykle przez przeprowadzenie operacji XOR na strumieniu klucza i strumieniu danych, a XOR wykonany dwukrotnie przy

pomocy tego samego klucza zwraca pierwotne dane, szyfrowanie i odszyfrowywanie wiadomości wygląda zazwyczaj w szyfrach strumieniowych identycznie.

2.2 Język C#

Zdecydowano się zaimplementować szyfry w środowisku .NET z wykorzystaniem języka C#. Wybór ten podyktowany został statycznym typowaniem obecnym w jeżyku C#, które ułatwia unikanie błędów programistycznych oraz łatwością tworzenia w tym środowisku GUI (biblioteka WPF).

2.3 Bouncy Castle

Bouncy Castle to otwartoźródłowa biblioteka kryptograficzna dostępna dla języków Java oraz C#. Zawiera ona implementacje kilkudziesięciu szyfrów, w tym około dziesięciu szyfrów strumieniowych [1]. Została ona wybrana jako źródło referencyjnych implementacji ze względu na jej popularność, a co za tym idzie dostateczną pewność co do jej poprawności oraz duży wybór algorytmów.

3 Opis algorytmów

W ramach projektu zdecydowano się zaimplementować szyfry RC4 oraz ChaCha20, których opis znajduje się poniżej.

3.1 RC4

RC4 jest prostym i szybkim szyfrem strumieniowym używanym między innymi w protokole WEP. Ze względu jednak na wykryte w nim podatności, jego użycie jest obecnie odradzane [2].

Algorytm RC4 zaczyna się od przyjęcia klucza o zmiennej długości (zwykle między 40 a 128 bitów, maksimum wynosi 2048) i utworzeniu na jego podstawie tablicy `T` o długości 256 bajtów. Jest to tzw. *Key-scheduling algorithm*, który został pokazany na poniższym pseudokodzie.

```
byte[] CreateArrayT(string key)
  byte[] T = new byte[256]

  for (int i = 0; i < 256; ++i)
    T[i] = i

  int j = 0
  for (int i = 0; i < 256; ++i)
    j = (j + T[i] + key[i % key.Length]) % 256
    byte temp = T[i]
    T[i] = T[j]
    T[j] = temp
  return T
```

Następnie na tablicy `T` wykonywany jest algorytm generujący pseudolosowe bajty aż uzyska się z nich pseudolosowy klucz o długości równej długości szyfrowanych danych. Pokazano to w kolejnym blok pseudokodu.

```
byte[] PrepareKeystream(int length)
  byte[] keystream = new byte[length];

  int p1 = 0;
  int p2 = 0;

  for (int i = 0; i < length; ++i)
    p1 = (p1 + 1) % 256
    p2 = (p2 + S[p1]) % 256

    byte temp = T[p1]
    T[p1] = T[p2]
    T[p2] = temp

    int p3 = (T[p1] + T[p2]) % 256

    keystream[i] = T[p3]
  return keystream
```

Uzyskany w ten sposób strumień klucza wykorzystać można do przeprowadzenia operacji XOR z danymi do zaszyfrowania lub odszyfrowania.

3.2 ChaCha20

ChaCha20 to odmiana szyfru Salsa20, jednego z 7 szyfrów strumieniowych, które przeszły wszystkie 3 fazy projektu ESTREAM mającego wyłonić szyfry strumieniowe zdatne do szerokiego zastosowania. Jego implementacja opierać będzie się na jego opisie w RFC 7539[3], z jedną zmianą - w RFC opisana została jego zmodyfikowana wersja z IV długości 32 bitów, a w projekcie zaimplementowana zostanie wersja oryginalna z 64 bitowym IV, taka sama jak w Bouncy Castle.

Algorytm zaczyna się od przyjęcia od użytkownika 256 bitowego klucza oraz 64 bitowego IV, które zostaną wykorzystane do stworzenia początkowego stanu algorytmu. Stan algorytmu składa się z 16 słów 32 bitowych (które można traktować jak nieujemne liczby całkowite). W pseudokodzie przedstawiono funkcję GetState, która przy liczbie `counter` równej 0 zwraca stan początkowy.

```
GetState(byte[] key, byte[] nonce, ulong counter)
  uint[] state = new uint[16]
  // Slowa 0-3 sa inicjowane przez okreslone w RFC stale.
  for (int i = 0; i < 4; ++i)
    state[i] = Constants[i]
```

```
  // Slowa 4-11 sa inicjowane przez klucz.
  // Kazde kolejne 4 bajty klucza traktowane sa
  // jak liczba calkowita zapisana w konwencji Little Endian.
  for (uint i = 4; i < 12; ++i)
    state[i] = GetLittleEndianInteger(key, (i - 4) * 4)

  // Slowa 12-13 inicjowane sa przez licznik
  state[12] = (uint) (counter & 0xFFFFFFFF)
  state[13] = (uint) (counter >> 32)

  // Slowa 14-15 inicjowane sa przez IV
  state[14] = GetLittleEndianInteger(nonce, 0)
  state[15] = GetLittleEndianInteger(nonce, 4)
  return state
```

W kolejnym kroku tworzony jest stan przetworzony przez wykonanie 20 tzw. rund algorytmu

```
uint[] GetTransformedState(uint[] inputState)
  // Stan nieprzetworzony bedzie jeszcze potrzebny,
  // dlatego operacje przeprowadzone zostana na jego kopii.
  uint[] returnState = inputState.Copy()

  // Kazde wykonanie petli wykonuje dwie rundy algorytmu:
  // kolumnowa i przekatna. Oznacza to wykonanie funkcji
  // QuarterRound dla kazdej kolumny i kazdej przekotnej,
  // jezeli stan algorytmu zobrazuje sie jako macierz 4x4
  for (int round = 0; round < 20; round += 2)

    // Runda kolumnowa skladajaca sie z 4 rund "cwiartkowych"
    for (byte i = 0; i < 4; ++i)
      byte[] indexes = { i, i + 4, i + 8, i + 12 }
      QuarterRound(returnState, indexes)

    // Runda przekatna skladajaca sie z 4 rund "cwiartkowych"
    QuarterRound(returnState, { 0, 5, 10, 15 })
    QuarterRound(returnState, { 1, 6, 11, 12 })
    QuarterRound(returnState, { 2, 7, 8, 13 })
    QuarterRound(returnState, { 3, 4, 9, 14 })
  return returnState
```

Funkcja QuarterRound czyli tzw. *runda ćwiartkowa* wygląda następująco:

```
QuarterRound(uint[] state, byte[] indexes)
  uint a = block[indexes[0]]
  uint b = block[indexes[1]]
  uint c = block[indexes[2]]
  uint d = block[indexes[3]]
```

```
a += b; d ^= a; d = RotateLeft(d, 16)
c += d; b ^= c; b = RotateLeft(b, 12)
a += b; d ^= a; d = RotateLeft(d, 8)
c += d; b ^= c; b = RotateLeft(b, 7)

block[indexes[0]] = a
block[indexes[1]] = b
block[indexes[2]] = c
block[indexes[3]] = d
```

Po wykonaniu funkcji `GetTransformedState` każda liczba z pierwotnego stanu dodana zostanie do odpowiadającej jej liczby ze stanu przetworzonego.

```
for (int i = 0; i < initialState.Length; ++i)
  transformedState[i] += initialState[i]
```

W kolejnym kroku stan przetworzony, czyli wektor 16 liczb 32 bitowych zamieniony zostaje na ciąg bajtów (zapisując liczby w formacie Little Endian). Tak uzyskany 512 bitowy klucz po przeprowadzeniu operacji XOR z tekstem jawnym, tworzy szyfrogram (lub odwrotnie, przeprowadzając operację XOR z szyfrogramem, odtwarza pierwotne dane).

Wszystkie opisane kroki są powtarzane aż do zaszyfrowania lub odszyfrowania wszystkich danych, zmieniany jest jedynie parametr `counter` w funkcji `GetState` - licznik zwiększany jest o 1 przy każdym nowym, 64 bajtowym bloku danych do przetworzenia.

Literatura

[1] "The Legion of the Bouncy Castle C# Cryptography APIs." `https://www.bouncycastle.org/csharp/`. Dostęp: 9.11.2017.

[2] "Prohibiting RC4 Cipher Suites." `https://tools.ietf.org/html/rfc7465`. Dostęp: 9.11.2017.

[3] "Chacha20 and Poly1305 for IETF Protocols." `https://tools.ietf.org/html/rfc7539`. Dostęp: 9.11.2017.