

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

Zaawansowane metody kryptografii i ochrony informacji

Autorzy: Tomasz Bocheński, Carlos Zaldivar Batista

Prowadzący: Albert Sitek

Semestr 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 języku 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 zdadne 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]
    // Słowa 0-3 są inicjowane przez określone w RFC stałe.
    for (int i = 0; i < 4; ++i)
        state[i] = Constants[i]
```

```

// Słowa 4–11 są inicjowane przez klucz.
// Każde kolejne 4 bajty klucza traktowane są
// jak liczba całkowita zapisana w konwencji Little Endian.
for (uint i = 4; i < 12; ++i)
    state[i] = GetLittleEndianInteger(key, (i - 4) * 4)

// Słowa 12–13 inicjowane są przez licznik
state[12] = (uint) (counter & 0xFFFFFFFF)
state[13] = (uint) (counter >> 32)

// Słowa 14–15 inicjowane są 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 będzie jeszcze potrzebny,
// dlatego operacje przeprowadzone zostaną na jego kopii.
uint[] returnState = inputState.Copy()

// Każde wykonanie petli wykonuje dwie rundy algorytmu:
// kolumnowa i przekatna. Oznacza to wykonanie funkcji
// QuarterRound dla każdej kolumny i każdej przekątnej,
// jeżeli stan algorytmu zobrazuje się jako macierz 4x4
for (int round = 0; round < 20; round += 2)

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

    // Runda przekatna składająca się z 4 rund "ćwiartkowych"
    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, zmieniający jest jedynie parametr `counter` w funkcji `GetState` - licznik zwiększany jest o 1 przy każdym nowym, 64 bajtowym bloku danych do przetworzenia.

4 Zastosowania praktyczne

Zaimplementowane moduły szyfrujące działają na danych dowolnego typu i mogą służyć do szyfrowania plików. Ze względu na to, że implementacje RC4 oraz ChaCha20 zachowują się zgodnie ze standardami, odbiorca zaszyfrowanych plików nie musi posiadać przedstawionego tu oprogramowania aby je odszyfrować, w tym celu może skorzystać z powszechnie dostępnych bibliotek.

Wadą utworzonych w ramach projektu modułów szyfrujących w porównaniu do biblioteki Bouncy Castle jest to, że szyfrują lub odszyfrowują one wszystkie dane na raz, nawet jeżeli użytkownika interesuje tylko wybrany ich fragment, na przykład pierwsze 100 bajtów, stanowiące metadane pliku.

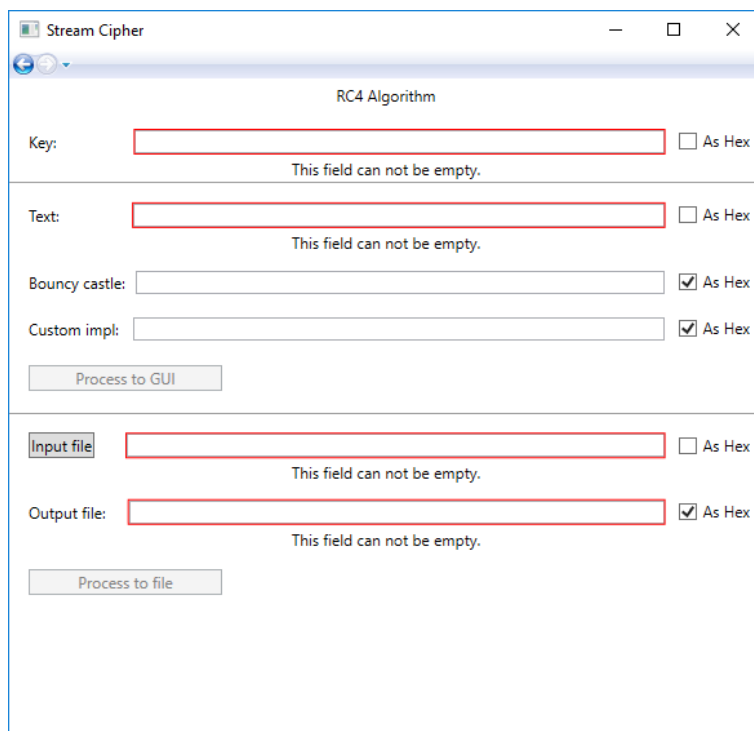
Innym problemem, powiązanym z pierwszym, są duże wymagania pamięciowe zaimplementowanych algorytmów, ponieważ nie pozwalają one na przetwarzanie wejścia w kolejnych, niewielkich blokach bajtów, ani nie wspierają one strumieni danych, a zamiast tego oczekują załadowania do pamięci operacyjnej wszystkich danych wejściowych, bez względu na ich rozmiar.

5 Instrukcja użytkowania programu

Program posiada trzy ekrany: ekran wyboru algorytmu i po jednym ekranie dla szyfrowania RC4 oraz ChaCha20. Wybór algorytmu odbywa się przez naciśnięcie jednego z dwóch przycisków na ekranie głównym.

5.1 Ekran RC4

Na rysunku 1 pokazano ekran szyfrowania z wykorzystaniem RC4. Użytkownik ma do wyboru dwa źródła danych do szyfrowania - pole tekstowe, w którym może wpisać tekst lub liczby heksadecymalne oraz plik. W przypadku wyboru pola tekstowego, wynik działania algorytmu (zarówno w implementacji własnej jak i w implementacji Bouncy Castle) wyświetla się w polach tekstowych znajdujących się niżej, a w przypadku wyboru pliku jako źródła danych, wynik szyfrowania zapisywany jest do `Custom_wybrana_nazwa_pliku` oraz `Bouncy_wybrana_nazwa_pliku`. Niezależnie od źródła danych do szyfrowania, wymagany do uzupełnienia jest dowolnej długości klucz. Po skończeniu szyfrowania/odszyfrowania tworzone są pliki z logami przedstawiające przebieg algorytmu.



Rysunek 1: Ekran szyfrowania RC4

5.2 Ekran ChaCha20

Na rysunku 2 pokazano ekran szyfrowania ChaCha20. Ekran ten wygląda i działa bardzo podobnie do uprzednio przedstawionego ekranu RC4, jedną z różnic jest obecność pola *Nounce*, w które należy wpisać 32 bitowy wektor inicjujący. Inną różnicą jest wymóg by klucz miał długość 256 bitów.

The screenshot shows a window titled "Stream Cipher" with a "ChaCha20 Algorithm" section. It contains several input fields and checkboxes:

- Key:** A text input field with a red border and the message "This field can not be empty." Below it is a checkbox labeled "As Hex" which is unchecked.
- Nonce:** A text input field with a red border and the message "This field can not be empty." Below it is a checkbox labeled "As Hex" which is unchecked.
- Text:** A text input field with a red border and the message "This field can not be empty." Below it is a checkbox labeled "As Hex" which is unchecked.
- Bouncy castle:** A text input field with a checkbox labeled "As Hex" which is checked.
- Custom impl:** A text input field with a checkbox labeled "As Hex" which is checked.
- Input file:** A text input field with a red border and the message "This field can not be empty." Below it is a checkbox labeled "As Hex" which is unchecked.
- Output file:** A text input field with a red border and the message "This field can not be empty." Below it is a checkbox labeled "As Hex" which is checked.

Buttons labeled "Process to GUI" and "Process to file" are located below the input fields.

Rysunek 2: Ekran szyfrowania ChaCha20

5.3 Logi

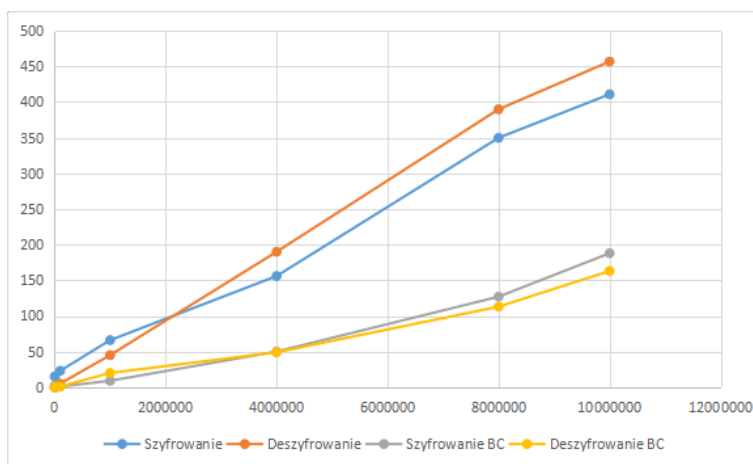
Po każdym uruchomionym szyfrowaniu/desyfrowaniu tworzony jest plik z logami (RC4.log lub ChaCha20.log). Celem logów nie jest notowanie każdej pojedynczej operacji, ponieważ przy operacjach powtarzających się w długich pętlach uczyniłoby to je nieczytelnymi. Zamiast tego logi mają za zadanie przedstawić działanie algorytmów w sposób możliwie zrozumiały, stąd unikanie redundancji i umieszczanie w logach stosownych komentarzy.

6 Testy

Przeprowadzono testy na losowych plikach tekstowych o wielkościach 1000, 10 000, 100 000, 1 000 000, 4 000 000, 8 000 000 i 10 000 000 bajtów. Testy polegały na zaszyfrowaniu oraz odszyfrowaniu wszystkich danych przez algorytmy RC4 i ChaCha20, zarówno w wersjach utworzonych w ramach tego projektu jak i w wersjach z biblioteki Bouncy Castle oraz porównaniu wyników oraz czasów tych operacji.

Przy wszystkich szyfrowaniach algorytmem ChaCha20 skorzystano z tego samego klucza oraz wektora inicjującego. Podobnie postąpiono przy szyfrowaniu RC4, do którego użyto 40 bajtowego klucza, czyli o tej samej długości co suma długości klucza i IV z algorytmu ChaCha20.

W każdym przypadku wyniki działania własnych implementacji algorytmów szyfrujących były identyczne z wynikami zwróconymi przez bibliotekę Bouncy Castle, co pozwala stwierdzić, że udało się uzyskać poprawność stworzonego oprogramowania. Widoczne różnice między implementacjami dotyczyły jednak ich wydajności - Bouncy Castle wykonywało szyfrowanie/deszyfrowanie wyraźnie szybciej. Na rysunku 3 przedstawiono wykres czasu przeprowadzenia operacji szyfrowania oraz deszyfrowania (w milisekundach) od wielkości danych (w bajtach) przy użyciu szyfrowania RC4. Na wykresie widać, że Bouncy Castle jest kilkakrotnie wydajniejszy.

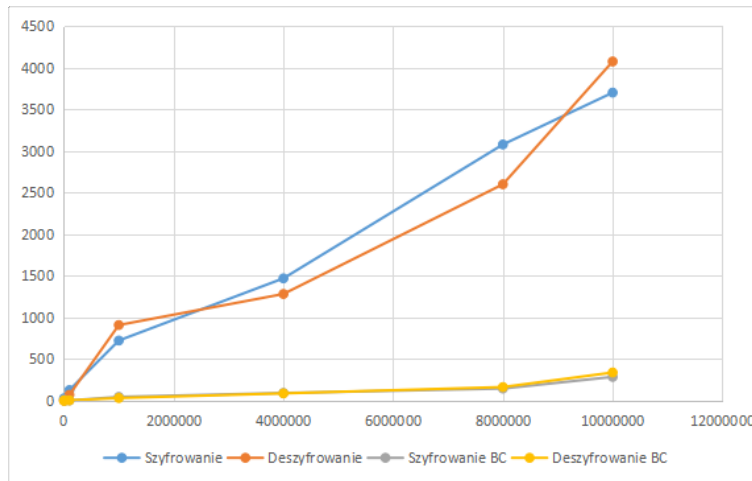


Rysunek 3: Porównanie wydajności różnych implementacji szyfrowania RC4

Na rysunku 4 pokazano z kolei porównanie wydajności implementacji ChaCha20. W tym wypadku różnica jest jeszcze większa - Bouncy Castle jest kilkunastokrotnie szybszy.

Z obu wykresów można wyciągnąć jeszcze kilka innych wniosków:

1. RC4 w obu implementacjach jest wyraźnie szybszy od ChaCha20. Gdyby szyfr ten nie był obecnie uznawany za niebezpieczny, byłby on praktycz-



Rysunek 4: Porównanie wydajności różnych implementacji szyfrowania ChaCha20

niejszym wyborem.

2. Czasy szyfrowania i deszyfrowania są podobne, co było spodziewane, ponieważ są to w istocie te same działania.

7 Podsumowanie

Wybrane szyfry strumieniowe okazały się stosunkowo proste w implementacji - kod szyfru RC4 zajął niecałe 90 linii a ChaCha20 około 160. ChaCha20 jest przy tym szybsze od popularnego szyfrowania AES[4] i według stanu obecnej wiedzy jest szyfrowaniem bezpiecznym. Łatwość implementacji, szybkość i, w przypadku algorytmów z projektu ESTREAM takich jak ChaCha20, bezpieczeństwo, pozwalają spekulować, że szyfry strumieniowe znajdą w przyszłości szersze zastosowanie.

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.

- [4] “ChaCha, a variant of Salsa20.” <https://cr.yp.to/chacha/chacha-20080128.pdf>. Dostęp: 18.11.2017.