Podstawy teoretyczne kryptografii i ochrony informacji

Dokumentacja projektu

Temat:
Programowa implementacja algorytmu szyfrowania RSA.

Autor:
Jakub Matraszek
nr indeksu: 203733

Spis treści

1 Założenia projektowe **2**

2 Teoria **2**
2.1 Algorytm RSA . . . 2
2.2 Klucze: publiczny i prywatny . . . 2
2.3 Szyfrowania i deszyfrowanie . . . 2
2.3.1 Szyfrowanie . . . 2
2.3.2 Deszyfrowanie . . . 3
2.4 Tryby szyfrowania . . . 3
2.4.1 ECB - Electronic codebook . . . 3
2.4.2 CBC - Cipher-block chaining . . . 4
2.4.3 CFB - Cipher feedback . . . 4
2.4.4 CTR - Counter . . . 5

3 Realizacja praktyczna **6**
3.1 Wykorzystane biblioteki . . . 6
3.2 Parser opcji - lib/options.rb . . . 7
3.3 Moduł metod matematycznych - lib/mathematics.rb . . . 7
3.4 Klasa klucza - lib/key.rb . . . 7
3.5 Klasa wiadomości - lib/text.rb . . . 8
3.6 Program główny - rubyrsa.rb . . . 9

4 Testowanie **9**
4.1 Test poprawności generowania klucza . . . 9
4.2 Test poprawności szyfrowania danych . . . 10
4.3 Test szybkości szyfrowania/deszyfrowania . . . 10

5 Podsumowanie **10**
5.1 Braki/możliwości rozwoju . . . 11

6 Bibliografia **11**

Spis rysunków

1 Szyfrowanie w trybie ECB . . . 3
2 Deszyfrowanie w trybie ECB . . . 3
3 Szyfrowanie w trybie CBC . . . 4
4 Deszyfrowanie w trybie CBC . . . 4
5 Szyfrowanie w trybie CFB . . . 5
6 Deszyfrowanie w trybie CFB . . . 5
7 Szyfrowanie w trybie CTR . . . 6
8 Deszyfrowanie w trybie CTR . . . 6

1 Założenia projektowe

Projekt miał spełniać następujące założenia:

- implementuje algorytm RSA
- pozwala szyfrować i deszyfrować wiadomości
- implementuje 4 tryby szyfrowania (wybrano: ECB, CBC, CFB i CTR)
- generuje klucz i zapisuje go na dysku do późniejszego użycia
- możliwość wprowadzenia danych ze standardowego wejścia bądź wczytania z pliku
- możliwość zapisania danych do pliku bądź wyprowadzenia na standardowe wyjście

2 Teoria

2.1 Algorytm RSA

Algorytm RSA jest jednym z pierwszych i jednym z najpopularniejszych asymetrycznych algorytmów szyfrowania. Opiera się na praktycznej niemożliwości faktoryzacji liczb złożonych (jeśli są odpowiednio długie).

2.2 Klucze: publiczny i prywatny

Generowanie nowej pary kluczy przebiega w sposób następujący:

1. Wylosowanie dwóch liczb pierwszych p i q. Bezpieczeństwo klucza możemy zmaksymalizować dobierając tak p i q, aby miały zbliżoną długość w bitach, ale znacząco różniły się wartościami.
2. Obliczamy $n = p * q$.
3. Obliczamy wartość funkcji Eulera dla n: $\varphi(n) = (p-1) * (q-1)$.
4. Losujemy takie e $(1 < e < \varphi(n))$, które jest względnie pierwsze z wartością funkcji Eulera dla n, tzn. $NWD(e, \varphi(n)) = 1$.
5. Znajdujemy taką liczbę d, że $e * d = 1 (mod \varphi(n))$
6. Klucz publiczny definiujemy jako parę liczb (n, e), a klucz prywatny jako (n, d).

2.3 Szyfrowania i deszyfrowanie

RSA jest szyfrem blokowym, więc na początku dzielimy wiadomość m na bloki m_i o długości nie większej niż n.

2.3.1 Szyfrowanie

Otrzymane bloki m_i szyfrujemy według wzoru: $c_i = m_i^e \ mod \, n$.

2.3.2 Deszyfrowanie

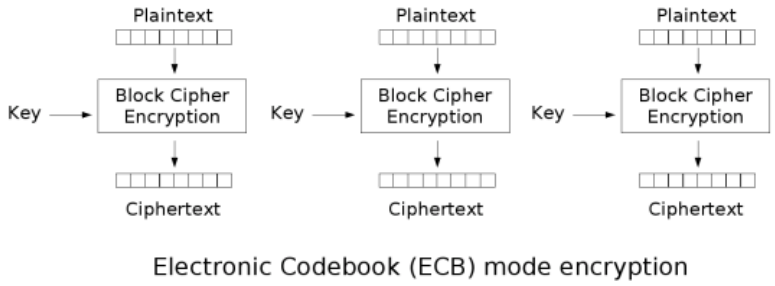
Zaszyfrowane bloki c_i odszyfrowujemy według wzoru: $m_i = c_i^d \ mod \, n$.

2.4 Tryby szyfrowania

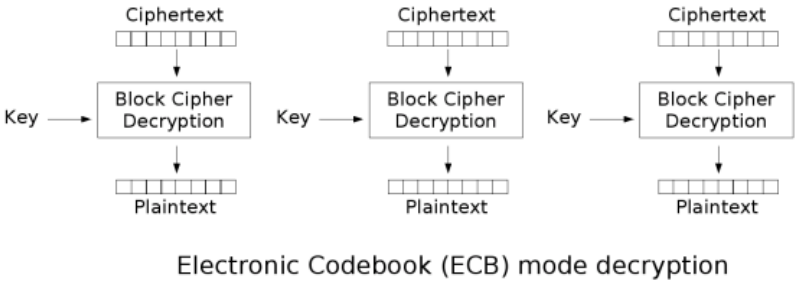
W RSA wiadomość jest dzielona na bloki i (de)szyfrowane są dane bloki. W zależności od sposobu obliczenia kolejnych bloków wiadomości zaszyfrowanej/jawnej możemy wyróżnić kilka trybów. W projekcie zaimplementowano poniższe 4:

2.4.1 ECB - Electronic codebook

Najprostszy tryb szyfrowania. Każdy blok jest (de)szyfrowany oddzielnie.
Tryb ten nie jest uważany za bezpieczny, ponieważ nie wprowadza elementu



Rysunek 1: Szyfrowanie w trybie ECB
$c_i = m_i^e \ mod \, n$

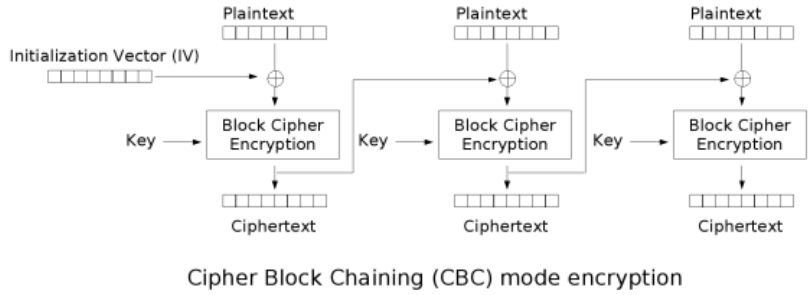


Rysunek 2: Deszyfrowanie w trybie ECB
$m_i = c_i^d \ mod \, n$

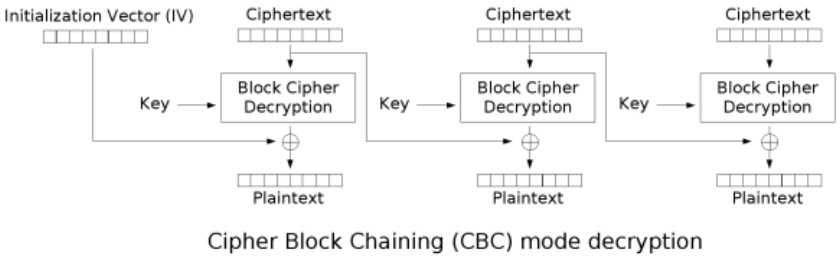
losowości - przy szyfrowaniu dużej wiadomości podzielonej na małe bloki szyfruje same dane, ale nie ukrywa ewentualnych regularności w wiadomości. W związku z tym zazwyczaj korzysta się z innego trybu szyfrowania.

2.4.2 CBC - Cipher-block chaining

Tryb, w którym przed potęgowaniem modulo n bloku dokonuje się na nim operacji XOR z poprzednim blokiem zaszyfrowanego tekstu. Dla pierwszego bloku w operacji XOR używa się wektora inicjalizacyjnego (IV). W przypadku deszyfrowania blok po potęgowaniu modulo n poddaje się operacji XOR z poprzednim blokiem zaszyfrowanym. Dla pierwszego bloku używa się w operacji XOR wektora inicjalizacyjącego.



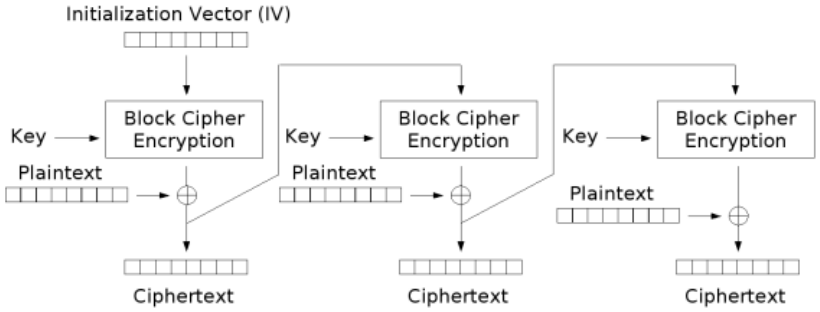
Rysunek 3: Szyfrowanie w trybie CBC
$c_i = (m_i \oplus c_{i-1})^e \; mod \, n; c_0 = IV$



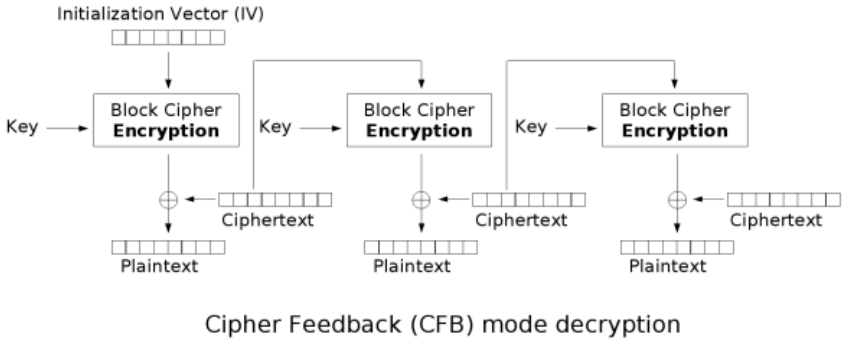
Rysunek 4: Deszyfrowanie w trybie CBC
$m_i = c_i^d \oplus c_{i-1} \; mod \, n; c_0 = IV$

2.4.3 CFB - Cipher feedback

Tryb, w którym w przypadku szyfrowania operacji potęgowania modulo n poddaje się poprzedni blok zaszyfrowanego tekstu (lub wektor IV), a następnie dokonuje operacji XOR z aktualnie szyfrowanym blokiem tekstu. W przypadku deszyfrowania operacji potęgowania modulo n poddaje się poprzedni blok zaszyfrowanego tekstu (lub wektor IV), a następnie dokonuje operacji XOR z aktualnie deszyfrowanym blokiem tekstu.



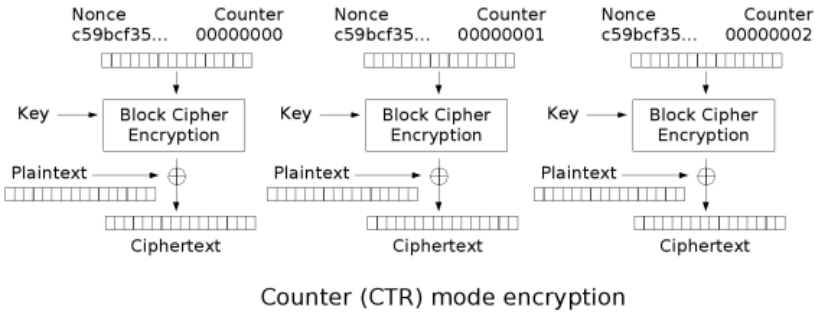
Rysunek 5: Szyfrowanie w trybie CFB
$c_i = c_{i-1}^e \oplus m_i \; mod \, n; c_0 = IV$



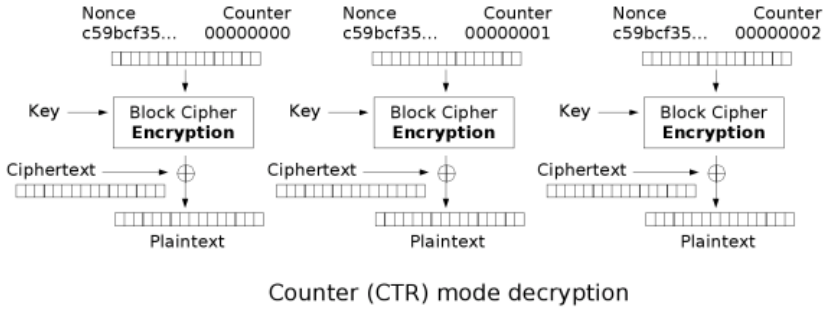
Rysunek 6: Deszyfrowanie w trybie CFB
$m_i = c_{i-1}^e \oplus c_i \; mod \, n; c_0 = IV$

2.4.4 CTR - Counter

Tryb, w którym używany jest licznik. Licznik może być każdą funkcją, która daje gwarancję, że wynikowa sekwencja nie powtarza się przez długi czas. Najprostszą taką funkcją jest zwykły licznik. W tym trybie szyfrując wiadomość operacji potęgowania modulo n poddajemy wartość licznika, następnie wynik poddajemy operacji XOR z aktualnie szyfrowanym blokiem tekstu jawnego. Deszyfrując wiadomość operacji potęgowania modulo n poddajemy również wartość licznik, następnie wynik poddajemy operacji XOR z aktualnym blokiem tekstu tajnego.



Rysunek 7: Szyfrowanie w trybie CTR
$c_i = ctr^e \oplus m_i \; mod \, n$



Rysunek 8: Deszyfrowanie w trybie CTR
$m_i = ctr^e \oplus c_i \; mod \, n$

3 Realizacja praktyczna

Implementację wykonano w języku Ruby w wersji 1.8.7 (http://www.ruby-lang.org/). Działa pod systemem Linux, jednak powinna poprawnie działać na wszystkich systemach, na których istnieje możliwość uruchomienia interpretatora Ruby MRI/CRuby (być może innych wersji również).

3.1 Wykorzystane biblioteki

W projekcie wykorzystano jedną zewnętrzną bibliotekę - Ruby-LibTomMath (http://copiousfreetime.rubyforge.org/ruby-libtommath/). Jest to biblioteka udostępniająca dla języka Ruby otwartoźródłową bibliotekę LibTomMath. Wykorzystana została klasa Prime, pozwalająca wygenerować liczbę pierwszą p o zadanej długości bitowej,taką, że $(p-1)/2$ również jest liczbą pierwszą. Dodatkowo można zarządać aby wygenerowana liczba pierwsze spałniała kongruencję z $3 mod 4$. Biblioteka tak znacząco rozszerza możliwości klasy Prime znajdującej się w module "mathn" standardowej biblioteki języka Ruby. Pozwala również na szybsze generowanie liczb pierwszych.

3.2 Parser opcji - lib/options.rb

Parser opcji tworzy zmienną globalną $options będącą strukturą opisującą opcje działania programu. Struktura zawiera następujące pola (wraz z domyślną wartością):

- bits (64) - liczba bitów klucza
- chunk_size (bits/8) - rozmiar bloku
- gen_key (false) - flaga określająca czy generować nową parę kluczy
- mode (cbc) - tryb szyfrowania
- direction (" ") - kierunek szyfrowania (domyślnie: szyfrowanie)
- infile (stdin) - plik wejściowy (domyślnie: standardowe wejście)
- outfile (stdout) - plik wyjściowy (domyślnie: standardowe wyjście).

Parser dokonuje również parsowania opcji linii komend, w razie potrzeby ustawiając odpowiednie pola struktury $options na wartości podane przez użytkownika.

3.3 Moduł metod matematycznych - lib/mathematics.rb

Moduł implementuje metody matematyczne używane przez klasy implementujące klucz oraz wiadomość:

- Funkcja Eulera $\varphi(n)$ - euler_func(p,q)
- Potęgowanie modulo n - mod(base, pow, mod)
- Rozszerzony algorytm Euklidesa - ext_euc(a,b)
- Największy wspólny dzielnik $NWD(a, b)$ - gcd(a,b)
- Metodę losowania liczby e - find_e(fi)
- Metoda znalezienia liczby d będącej odwrotnością e w $Z_{\varphi(n)}$ - find_d(e,fi)

Przy implementowaniu algorytmów matematycznych korzystano z książki A. Menezes, P. van Oorschot oraz S. Vanstone - Handbook of Applied Cryptography, CRC Press, 1996.

3.4 Klasa klucza - lib/key.rb

Klasy klucza implementuje dwie metody - initialize i write_key.

- initialize - metoda konstruktora. Wywoływana przy tworzeniu nowego obiektu klasy Key. W przypadku podania opcji linii komend -n lub –genkey metoda utworzy nowy klucz korzystając z biblioteki Ruby-LibTomMath oraz metod zainplementowanych w module Mathematics. Klucz zostanie zapisany do pliku o nazwie .key.txt. W przypadku braku opcji uruchamiających generowanie klucza metoda initialize wczyta klucz z pliku .key.txt.
- write_key - metoda wywoływana w celu zapisania wygenerowanego klucza do pliku o nazwie .key.txt

3.5 Klasa wiadomości - lib/text.rb

Klasa wiadomości implementuje operowanie wiadomością - szyfrowania oraz deszyfrowanie oraz kilka pomocniczych metod. Klasa Text dziedziczy po klasie Array (tablica). Utworzony obiekt klasy Text jest de facto tablicą o jednym elemencie będącym łańcuchem znaków (wiadomość do zaszyfrowania). Pozwala to na łatwy podział na bloki (tablica będzie zawierać łańcuchy znaków będące blokami wiadomości) oraz prostotę implementacji metod (de)szyfrujących (iterowanie po elementach tablicy i wykonywanie na nich odpowiednich operacji). Dzięki takiemu ujęciu nie jest problemem użycie wektora IV - jest on umieszczany na końcu tablicy i można się do niego odwołać (dzięki właściwości języka Ruby) poprzez indeks -1. Zaimplementowane metody klasy Text:

- initialize - metoda konstruktowa. Pozwala utworzyć obiekt klasy Text z jawnie podanym tekstem wiadomości (należy wtedy wywołać Text.new("tekst wiadomości") bądź z tekstem wczytanym z pliku (wywołanie Text.new bez parametru). Jeżeli kierunek działania algorytmu to szyfrowanie to zostanie utworzona tablica o jednym elemencie, jeżeli kierunek to deszyfrowanie to zostanie utworzona tablica o ilości elementów odpowiadajej ilości wczytywanych bloków.
- rotate - pomocnicza metoda zmieniająca kolejność bloków (używana w deszyfrowaniu w trybie CBC)
- create_chunks! - pomocnicza metoda dzieląca wiadomość na bloki. Zgodnie z konwencją przyjętą w języku Ruby wykrzyknik na końcu nazwy metody oznacza modyfikację obiektu, na rzecz którego metoda jest wywoływana.
- to_bignum(s) - pomocnicza metoda zamieniająca łańcuch znaków s na liczbę Bignum.
- to_string(n) - pomocnicza metoda zamieniająca liczbę Bignum na łańcuch znaków s.
- chunks_to_bignum! - pomocnicza metoda konwertująca wszystkie bloki danej wiadomości na liczby Bignum z łańcuchów znaków
- chunks_to_string! - pomocnicza metoda konwertująca wszystkie bloki danej wiadomości na łańcuchy znaków z liczby Bignum
- to_output - pomocnicza metoda wypisująca dane na wyjście - wyjściem może być plik bądź standardowe wyjście (w zależności od użytych opcji linii komend)
- ecb_crypt!(key) - pomocnicza (do (de)szyfrowania powinna zostać użyta opisana niżej metoda crypt!(key), ponieważ uwzględnia opcje linii komend i wywołuje odpowiednią metodę *_(de)crypt!) metoda implementująca algorytm szyfrowania przedstawiony w sekcji 2.4.1
- ecb_decrypt!(key) - pomocnicza metoda implementująca algorytm deszyfrowania przedstawiony w sekcji 2.4.1
- cbc_crypt!(key) - pomocnicza metoda implementująca algorytm szyfrowania przedstawiony w sekcji 2.4.2

- cbc_decrypt!(key) - pomocnicza metoda implementująca algorytm deszyfrowania przedstawiony w sekcji 2.4.2
- cfb_crypt!(key) - pomocnicza metoda implementująca algorytm szyfrowania przedstawiony w sekcji 2.4.3
- cfb_decrypt!(key) - pomocnicza metoda implementująca algorytm deszyfrowania przedstawiony w sekcji 2.4.3
- ctr_crypt!(key) - pomocnicza metoda implementująca algorytm szyfrowania przedstawiony w sekcji 2.4.4
- ctr_decrypt!(key) - pomocnicza metoda implementująca algorytm deszyfrowania przedstawiony w sekcji 2.4.4
- crypt!(key) - główna metoda klasy Text. Metoda crypt! wywołuje odpowiednią metodę szyfrującą/deszyfrującą, która została zdefiniowana dla klasy Text. Rozwiązanie takie pozwala na łatwą rozbudowę kodu przy dodaniu nowego trybu - należy go uwzględnić w pliku lib/options.rb oraz zaimplementować odpowiednie metody szyfrowania i deszyfrowania - sama metoda crypt! nie musi być modyfikowana. Jest to o wiele łatwiejsze rozwiązanie niż np. znana z języka C tablica wskaźników na funkcje.

3.6 Program główny - rubyrsa.rb

Program główny jest bardzo prosty - jedyne co robi, to dołącza niezbędne pliki, tworzy nowy obiekt klasy Key, tworzy obiekt klasy Text oraz wywołuję metodę obiektu klasy Text o nazwie crypt!, jako parametr podając obiekt klasy Key.

4 Testowanie

W celu przetestowania poprawności zaimplementowanych algorytmów zaimplementowano dwa testy jednostkowe: generowania klucza oraz szyfrowania danych.

4.1 Test poprawności generowania klucza

Test ten opiera się na założeniu, że dla poprawnie wygenerowanego klucza zachodzi równość $e * d = 1 \ (mod\, \varphi(n)$. Zaimpelementowany został test jednostkowy (plik test/tc_key.rb) sprawdzający czy zachodzi powyższa równość. Wygenerowano 1000 kluczy, obliczono wyżej wymieniony iloczyn i porównano z wartością 1. Oto uzyskane wyniki:

```
$ ruby test/tc_key.rb
Loaded suite test/tc_key
Started
.
Finished in 126.908223 seconds.
1 tests, 1000 assertions, 0 failures, 0 errors
```

Wyniki powyższego testu pozwalają stwierdzić, że algorytm generowania klucza jest poprawny.

4.2 Test poprawności szyfrowania danych

Test ten opiera się na założeniu, że dla poprawnie zaimplementowanego algorytmu (de)szyfrowania otrzymamy początkowy tekst wiadomości po zaszyfrowaniu oraz odszyfrowaniu jej.. Zaimpelementowany został test jednostkowy (plik test/tc_key.rb) sprawdzający czy zachodzi powyższa właściwość. Wygenerowano 100 wiadomości dla każdego trybu szyfrowania (a więc w sumie 400 asercji), następnie każdą wiadomość zaszyfrowano oraz odszyfrowano i porównano z oryginalnie wygenerowaną wiadomością. Oto uzyskane wyniki:

```
$ ruby test/tc_text.rb
Loaded suite test/tc_text
Started
....
Finished in 58.806893 seconds.
4 tests, 400 assertions, 0 failures, 0 errors
```

Również w tym przypadku wyniki powyższych testów pozwalają stwierdzić, że algorytmy szyfrowania i deszyfrowania zostały zaimplementowane poprawnie.

4.3 Test szybkości szyfrowania/deszyfrowania

Dokonano testu szyfrowania oraz deszyfrowania na 60-kilobajtowym pliku tekstym o nazwie triangle.txt:

```
$ time ./rubyrsa.rb -i triangle.txt -o triangle-crypted.txt
real    0m6.708s
user    0m6.660s
sys     0m0.036s

$ time ./rubyrsa.rb -d -i triangle-crypted.txt -o triangle2.txt
real    0m6.495s
user    0m6.460s
sys     0m0.024s
```

Dokonano również sprawdzenia czy proces szyfrowania i deszyfrowania nie wprowadził przekłamań do tekstu - wywołanie

```
$diff triangle.txt traingle2.txt
```

nie zwróciło żadnych różnic.
Zainplementowany algorytm nie jest najszybszy - z pewnoscią można by go zoptymalizować.

5 Podsumowanie

Zaimplementowano poprawnie działający algorytm szyfrowania RSA. Szyfrowanie może odbywać się w jednym z czterech trybów (ECB, CBC, CFB, CTR). Użytkownik może wprowadzać dane z pliku bądź standardowego wejścia. Dane mogą zostać zapisane do pliku bądź wyprowadzone na standardowe wyjście. Zgodnie z przeprowadzonymi testami algorytm generowania klucza, algorytm szyfrowania/deszyfrowania oraz tryby szyfrowania zostały zaimplementowane poprawnie.

5.1 Braki/możliwości rozwoju

- Projekt jedynie prezentuje poprawność implementacji algorytmu, więc nie zaimplementowano możliwości importu kluczy publicznych innych użytkowników.
- Nie zaimplementowano algorytmu podpisu cyfrowego - byłaby to stosunkowo prosta modyfikacja, jako, że algorytm podpisu różni się od algorytmu szyfrowania jedynie zamianą kluczy.
- Nie zaimplementowano sprawdzenia czy p i q są od siebie odległe wartościami, co może prowadzić do osłabienia siły klucza.
- Projekt nie jest kompatybilny ze standardem OpenPGP.

6 Bibliografia

1. A. Menezes and P. van Oorschot and S. Vanstone - Handbook of Applied Cryptography, CRC Press, 1996
2. Morris Dworkin - Recommendation for Block Cipher Modes of Operation. Methods and Techniques, National Institute of Standart and Technology, 2001
3. http://en.wikipedia.org/wiki/RSA
4. http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation