Przełamywanie szyfru RSA

dokumentacja do projektu z przedmiotu „Teoria Obliczeń i Złożoności Obliczeniowej”

Marcin Rociek
Informatyka, III rok

1 lutego 2001

Spis treści

1 Wstęp **2**

2 Systemy kryptograficzne **2**
2.1 Systemy z pojedynczym kluczem 2
2.2 Systemy z kluczem publicznym 2

3 Funkcje jednokierunkowe **2**

4 Algorytm RSA **3**
4.1 Tworzenie kluczy 3
4.2 Szyfrowanie wiadomości 3
4.3 Odszyfrowanie wiadomości 3
4.4 Przykład działania 4
4.5 Bezpieczeństwo 4

5 Generowanie dużych liczb pierwszych **4**
5.1 Test Fermata 5
5.2 Test Millera-Rabina 5

6 Generowanie klucza prywatnego **5**
6.1 Algorytm Euklidesa 5
6.2 Rozszerzony algorytm Euklidesa 6

7 Faktoryzacja **7**
7.1 Metoda Pollarda 7
7.2 Złożoność obliczeniowa 7

8 Implementacja **8**
8.1 Działanie programu 8
8.2 Opcje 8
8.3 Opis implementacji 8
8.4 Wyniki *łamania* 8
8.5 Sprzęt 9

A Kod źródłowy **11**

1 Wstęp

RSA jest algorytmem szyfrowania z kluczem publicznym, opracowanym w 1977 roku przez trójkę profesorów z uniwersytetu MIT: Ronalda L. Rivesta, Adi Shamira oraz Leonarda M. Adlemana (jego nazwa pochodzi właśnie od nazwisk autorów). Algorytm RSA korzysta z pewnych właściwości wielkich liczb, które uniemożliwiają jego *złamanie*.

Wykorzystywany jest między innymi w PGP (Pretty Good Privacy) z kluczem o długości 512, 1024 lub 1280 bitów.

Po 17 latach od opatentowania, we wrześniu 2000 roku, algorytm ten stał się własnością publiczną (*public domain*).

2 Systemy kryptograficzne

System kryptograficzny to jakiś algorytm, który potrafi zamienić dane wejściowe w pewną nieodczytywalną, zaszyfrowaną treść, oraz przekształcić zaszyfrowane dane spowrotem do oryginalnej postaci. W celu zaszyfrowania danych, części szyfrującej algorytmu podajemy wejściowe dane oraz klucz szyfrujący. Aby zaszyfrowane dane odszyfrować, musimy podać algorytmowi zaszyfrowane dane oraz odpowiedni klucz deszyfrujący. Klucz jest po prostu jakąś tajną liczbą, bądź zestawem liczb. W zależności od systemu kryptograficznego liczby te mogą być losowe lub też mogą być wyznaczone za pomocą odpowiednich reguł matematycznych.

2.1 Systemy z pojedynczym kluczem

W systemach kryptograficznych z pojedynczym kluczem, klucze szyfrujący i deszyfrujący są takie same. Istnieją dwie funkcje: funkcja szyfrująca, oraz odwrotna do niej funkcja deszyfrująca. Ponieważ zarówno nadawca wiadomości jak i odbiorca używają tego samego klucza, muszą najpierw wymienić go między sobą poprzez bezpieczne kanały, aby móc z niego korzystać przesyłając zaszyfrowane wiadomości poprzez kanały nie zabezpieczone.

Wadą systemów z pojedynczym kluczem jest to, że wymagają one, aby obie strony znały klucz przed każdą transmisją. Co więcej, ponieważ nikt poza adresatem zaszyfrowanej wiadomości nie powinien mieć możliwości odszyfrowania transmisji, trzeba stworzyć różne klucze dla wszystkich osób (grup, firm itp.), do których są przesyłane zaszyfrowane informacje. Niewygoda z utrzymaniem dużej liczby pojedynczych kluczy jest oczywista. Poza tym jeśli do wymiany samego klucza jest używany bezpieczny kanał, to można by po prostu przez ten sam kanał transmitować dane.

2.2 Systemy z kluczem publicznym

W przeciwieństwie do systemów z kluczem pojedynczym, w systemach z kluczem publicznym istnieją dwa klucze: klucz *prywatny* oraz klucz *publiczny*. Klucz publiczny można swobodnie przekazywać komukolwiek (kolegom, współpracownikom itp.), natomiast klucz prywatny powinien być przechowywany w bezpiecznym miejscu i nie udostępniany nikomu.

W systemie takim osoba A, która chce przekazać zaszyfrowaną wiadomość osobie B, wykorzystuje do tego celu klucz publiczny udostępniony przez osobę B. Po zaszyfrowaniu wiadomości tylko osoba B może ją odszyfrować, wykorzystując w tym celu swój klucz prywatny (nawet nadawca A nie może już odszyfrować zaszyfrowanej wiadomości). Analogicznie, jeśli osoba B chce przekazać poufną wiadomość osobie A, szyfruje ją przy pomocy klucza publicznego udostępnionego przez osobę A.

Ponieważ publiczny klucz szyfrujący nie może odszyfrować zaszyfrowanej wiadomości, jego znajomość w niczym nie pomoże osobom trzecim, pragnącym przeczytać zaszyfrowaną wiadomość. Odczytać wiadomość może tylko i wyłącznie adresat, przy pomocy swojego klucza prywatnego.

3 Funkcje jednokierunkowe

Wyzwaniem dla twórców systemów kryptograficznych z kluczem publicznym jest opracowanie takiego systemu, w którym niemożliwe jest odgadnięcie klucza prywatnego na podstawie klucza publicznego. Dokonać tego można przy pomocy tzw. funkcji jednokierunkowych. Funkcja taka w stosunkowo prosty sposób oblicza wynik dla pewnych wartości wejściowych, lecz znając tylko wynik niezwykle trudno jest obliczyć wartość

wejściową. W sensie matematycznym: dla pewnej wartości x obliczenie $f(x)$ jest proste, natomiast odgadnięcie wartości x tylko na podstawie wartości $f(x)$ jest niezwykle trudne.

Okazuje się, że mnożenie może pełnić rolę funkcji jednokierunkowej. Wymnożenie dwóch liczb pierwszych jest bardzo proste (zwłaszcza dla komputera), natomiast rozłożenie na czynniki pierwsze większości bardzo dużych liczb jest niezwykle czasochłonne.

Przykładowo rozłożenie na czynniki pierwsze liczby 15 jest proste: $3 \cdot 5$, natomiast liczba 91723948197324 sprawi już trochę więcej kłopotu, tym bardziej liczba 155- czy 200-cyfrowa. Rozłożenie liczby na czynniki pierwsze składa się z pewnej ilości kroków, która to ilość zwiększa się wykładniczo wraz z wzrostem wielkości liczby. Oznacza to, że jeśli liczba jest wystarczająco duża, jej faktoryzacja (rozłożenie na czynniki pierwsze) może zająć lata.

4 Algorytm RSA

W większości przypadków użytkownik szyfruje wiadomości tekstowe, istotne jest jednak to, że komputer realizuje szyfrowanie wykonując szereg operacji matematycznych. Wiadomość musi być więc zamieniona na postać liczbową przed szyfrowaniem (oczywiście po odszyfrowaniu musi być zamieniona spowrotem na oryginalną wiadomość).

Algorytm RSA realizuje asymetryczne kodowanie z kluczami publicznym i prywatnym. Asymetryczna funkcja kodująca wykorzystuje opisane wcześniej funkcje jednokierunkowe, zapewniające bezpieczeństwo RSA, które wynika przede wszystkim z dużych trudności w faktoryzacji wielkich liczb.

4.1 Tworzenie kluczy

Tworzenie pary kluczy przebiega następująco:

1. Wybieramy losowo dwie duże liczby pierwsze p i q
2. Obliczamy ich iloczyn
$$n = p \cdot q$$
3. Wybieramy losowo klucz szyfrujący e taki, żeby liczby e i $(p-1)\cdot(q-1)$ były względnie pierwsze (tzn. nie miały wspólnych podzielników)
4. Wyznaczamy klucz deszyfrujący d będący odwrotnością e modulo $(p-1)\cdot(q-1)$, tzn.
$$(e \cdot d) \bmod (p-1)\cdot(q-1) = 1$$

Liczby e i n tworzą klucz publiczny, d jest kluczem prywatnym. Po obliczeniu n, e i d liczby p i q nie są już potrzebne (powinny zostać wymazane).

4.2 Szyfrowanie wiadomości

Aby otrzymać zaszyfrowaną wiadomość c z pewnej wiadomości m, podnosimy wiadomość m do potęgi e modulo n, tzn:
$$c = m^e \bmod n$$

W praktyce wiadomość musi zostać podzielona wcześniej na bloki o jednoznacznej reprezentacji modulo n.

4.3 Odszyfrowanie wiadomości

W celu odszyfrowania wiadomości, podnosimy zaszyfrowaną wiadomość c do potęgi d modulo n:

$$c^d \bmod n = m^{e\cdot d} \bmod n = m^1 \bmod n = m$$

4.4 Przykład działania

Przykładowe działanie algorytmu RSA:

1. Wybieramy losowo dwie liczby pierwsze: $p = 13$, $q = 11$ (w praktyce są one oczywiście dużo większe).
2. Wyliczamy $n = 13 \cdot 11 = 143$.
3. $(p-1)\cdot(q-1) = 12 \cdot 10 = 120 \rightarrow$ wybieramy klucz publiczny $e = 7$ (120 i 7 są względnie pierwsze).
4. Wyliczamy klucz prywatny $d = 103$ ($(7 \cdot 103) \bmod 120 = 721 \bmod 120 = 1$).

Przykładowa „wiadomość” 40 zostaje zaszyfrowana: $40^7 \bmod 143 = 105$.
Zaszyfrowana wiadomość 105 zostaje odszyfrowana: $105^{103} \bmod 143 = 40$.

4.5 Bezpieczeństwo

Możliwe jest uzyskanie prywatnego klucza d na podstawie klucza publicznego (e,n), poprzez faktoryzację n na czynniki pierwsze p i q. Aby wyliczyć d (klucz prywatny) trzeba znać wartość $(p-1)\cdot(q-1)$, a do znajomości tejże potrzeba znać właśnie wartości p i q. W podanym powyżej przykładzie, osoba trzecia, próbująca odszyfrować wiadomość zna tylko wartość $p \cdot q = 143$, nie wie natomiast ile wynosi $(p-1)\cdot(q-1)$.

Rozłożenie na czynniki pierwsze liczby 143 jest stosunkowo proste: 13 i 11. Jednakże w przypadku bardzo dużych liczb (np. 200-cyfrowych) faktoryzacja jest bardzo trudna.

5 Generowanie dużych liczb pierwszych

Tworzenie kluczy w algorytmie RSA opiera się na znalezieniu na początku dwóch wielkich liczb pierwszych p i q. Jeśli nie było by to stosunkowo proste, algorytm RSA byłby zbyt trudny do zaimplementowania. Jak się okazuje istnieją techniki pozwalające na generowanie losowych liczb pierwszych.

Najpierw generujemy jakąś losową liczbę a, następnie za pomocą sita Erastotenesa o określonej wielkości „poprawiamy” liczbę a tak, aby nie była podzielna przez znane początkowe liczby pierwsze (2, 3, 5, 7, 11, 13, 17).

Przykładowo mamy sito o 20 pozycjach, na początku wypełniamy je zerami:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Losujemy jakąś liczbę a, np. 793472. Najpierw sprawdzamy, czy jest podzielna przez 2, tj. obliczamy resztę z dzielenia $793472/2$. Reszta wynosi 0, a więc a jest podzielne przez 2. Wpisujemy więc 1 do tablicy na pozycji 0, 2, 4... (jeśli $a+0$ jest podzielne przez 2, to $a+2$ również jest, jak również $a+4$ itd.).

1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Następnie robimy to samo dla 3. Szukamy reszty z dzielenia $793472/3$. Reszta wynosi 2, a więc proponowana liczba nie jest podzielna przez 3. Jeśli natomiast do liczby dodamy 3 i odejmiemy resztę z dzielenia, otrzymamy liczbę która jest podzielna przez 3. Jeśli więc $a+1$ ($+1$, bo $3-2=1$) jest podzielne przez 3, to do tablicy wstawiamy jedynki na pozycji 1, oraz co trzecią następną:

1	**1**	1	0	**1**	0	1	**1**	1	0	**1**	0	1	**1**	1	0	**1**	0	1	**1**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Następnie kontynuujemy to działanie dla wszystkich liczb pierwszych z listy. Po wykonaniu tego dla 2, 3, 5, 7, 11, 13 i 17 tablica wygląda następująco:

1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	0	1	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Po tej operacji odsiane zostały liczby, które na pewno nie są pierwszymi. Jeśli jakaś pozycja i tablicy zawiera 0, oznacza to, że nie znaleźliśmy czynnika liczby $a+i$ (co jeszcze wcale nie oznacza, że ten czynnik nie istnieje). Oczywiście wielkość tej tablicy nie jest ograniczona, można więc zbudować tablicę powiedzmy 1000-elementową.

Następnym krokiem w poszukiwaniu liczby pierwszej może być zastosowanie jakiegoś testu probabilistycznego, np. takiego jak opisane poniżej. Ponieważ jednak są one bardzo czasochłonne, lepiej jest wcześniej właśnie „odsiać” niepożądanych kandydatów przez sito Erastotenesa.

5.1 Test Fermata

Dla danej liczby a (kandydata na liczbę pierwszą) oraz znanej liczby pierwszej b obliczamy:

$$f = b^{a-1} \bmod a$$

Jeśli $f \neq 1$ to a nie jest liczbą pierwszą, jeśli zaś $f = 1$ to a jest najprawdopodobniej liczbą pierwszą (prawdopodobieństwo, że a nie jest liczbą pierwszą dla testu Fermata wykonanego dla b =3, 5, 7, 11 jest równe w przybliżeniu 10^{-22}).

Kontynuując przykład z liczbą 793472, wykonujemy ten test dla liczby $793472+5$ (5 jest pierwszą pozycją w sicie, na której jest 0). Obliczamy $3^{793476} \bmod 793477 = 355221$, a więc 793477 na pewno nie jest liczbą pierwszą. Próbujemy więc $793472+11$: $3^{793482} \bmod 793483 = 16293$. Kolejna próba dla liczby $793472+15$: $3^{793486} \bmod 793487 = 1$, a więc 793487 może być liczbą pierwszą. Dla pewności można wykonać ten test dla 5, 7 i 11.

5.2 Test Millera-Rabina

Test Millera-Rabina jest bardziej skomplikowany, algorytm składa się z następujących kroków:

1. Wybieramy k liczb losowych b z przedziału $0 < b < a$.
2. Jeśli $b^{a-1} \bmod a$ jest różne od 1, to liczba a nie jest liczbą pierwszą (test Fermata).
3. Obliczamy $a-1 = 2^s \cdot t$, gdzie t jest nieparzyste.
4. Jeśli $b^t = 1 \bmod a$, przechodzimy do następnej liczby b.
5. Jeśli b^t jest różne od 1 mod a, obliczamy kwadrat b^t modulo a, następnie kwadrat z kwadratu itd., aż dostaniemy $b^{2^s \cdot t} = b^{a-1} = 1$. Jeśli otrzymany ciąg liczb nie zawiera $-1 = a-1 \bmod a$, to liczba a nie jest pierwsza, w przeciwnym wypadku przechodzimy do następnej liczby b.
6. Jeśli liczba a jest pierwsza dla wszystkich k liczb b, to jest ona liczbą pierwszą z prawdopodobieństwem błędu $1/4^k$. Jeśli $k > 50$, prawdopodobieństwo błędu jest mniejsze niż 10^{-30}.

6 Generowanie klucza prywatnego

Zgodnie z opisem algorytmu RSA, generowanie klucza prywatnego d polega na obliczeniu odwrotności $e \bmod (p-1)\cdot(q-1)$. Znakomicie do tego celu nadaje się rozszerzony algorytm Euklidesa opisany poniżej.

6.1 Algorytm Euklidesa

Podstawowy algorytm Euklidesa znajduje największy wspólny dzielnik (NWD) dwóch liczb naturalnych. NWD jest największą liczbą dzielącą bez reszty dwie liczby. Przykładowo NWD liczb 12 i 16 jest liczba 4. Żadna liczba większa od 4 nie jest dzielnikiem zarówno 12 jak i 16. Liczba 6 jest większa od 4 i jest dzielnikiem liczby 12, ale nie jest dzielnikiem 16.

Algorytm Euklidesa wygląda następująco: dla dwóch liczb naturalnych (u, v):

1. Sprawdzamy, czy v jest dzielnikiem u: obliczamy u/v. Jeśli nie istnieje reszta z dzielenia, to NWD=v, w przeciwnym wypadku przechodzimy do kroku 2.
2. Podstawiamy za u resztę z dzielenia z kroku 1. Jeśli reszta z dzielenia u/v (co można zapisać $u \bmod v$) jest równa 1, to liczby są liczbami względnie pierwszymi i NWD=1. W przeciwnym wypadku powracamy do kroku 1 z liczbami $(v, u \bmod v)$.

Przykład, szukamy NWD(945, 217):

Krok 1: 945/217 = 4 reszty 77
Krok 2: szukamy NWD(217, 77)
Krok 1: 217/77 = 2 reszty 63
Krok 2: szukamy NWD(77, 63)
Krok 1: 77/63 = 1 reszty 14
Krok 2: szukamy NWD(63, 14)
Krok 1: 63/14 = 4 reszty 7
Krok 2: szukamy NWD(14, 7)
Krok 1: 14/7 = 2 reszty 0, zatem NWD(945, 217)=7

6.2 Rozszerzony algorytm Euklidesa

Istnieje wiele modyfikacji tego algorytmu, lecz jedno rozszerzenie jest szczególnie interesujące: podczas obliczania NWD(u, v) możemy również znaleźć u' i v', takie, że:

$$u \cdot u' + v \cdot v' = NWD(u, v)$$

W kroku 1 algorytmu Euklidesa sprawdzaliśmy, czy v jest NWD. Jeśli tak jest, to $u' = 0$, zaś $v' = 1$.

$$u \cdot 0 + v \cdot 1 = v$$

Jeśli tak nie jest, szukamy NWD(v, u mod v). Jeśli u mod v jest NWD, to $u' = 1$, zaś v' będzie równe części całkowitej z u/v ze zmienionym znakiem.

$$u \cdot 1 - v \cdot \text{ int}(u/v) = u \bmod v.$$

To zaś może być definicją u mod v, czyli tego samego, czym jest reszta z dzielenia. Przykładowo, jeśli szukaliśmy NWD(945, 217), sprawdzaliśmy 217, a następnie szukaliśmy NWD(217, 945 mod 217). A 945 mod 217 wynosi

$$945 \cdot 1 - 217 \cdot \text{ int}(945/217) = 945 - 217 \cdot 4 = 945 - 868 = 77$$

Kontynuując, po każdych dwóch krokach algorytmu Euklidesa znajdujemy u' i v' z równania

$$u \cdot u' + v \cdot v' = \text{proponowane NWD}$$

Rozszerzony algorytm Euklidesa wygląda następująco:

1. Inicjalizacja:
   $(u_1, u_2, u_3) = (1, 0, u)$
   $(v_1, v_2, v_3) = (0, 1, v)$
2. Jeśli $v_3 = 0$, to $u' = u_1$ i $v' = u_2$, koniec algorytmu. W przeciwnym wypadku przechodzimy do kroku 3.
3. Obliczamy:
   $q = \text{ int}(u_3/v_3)$
   $(t_1, t_2, t_3) = (u_1, u_2, u_3) - q \cdot (v_1, v_2, v_3)$.
   Następnie podstawiamy:
   $(u_1, u_2, u_3) = (v_1, v_2, v_3)$
   $(v_1, v_2, v_3) = (t_1, t_2, t_3)$
   i powracamy do kroku 2.

W przykładzie z liczbami 945 i 217, pierwsze dwie iteracje tego algorytmu wyglądały by mniej więcej w sposób następujący:

u_i	v_i	u_i - $q \cdot v_i$	t_i
1	0	1 - $4 \cdot 0$	1
0	1	0 - $4 \cdot 1$	-4
945	217	945 - $4 \cdot 217$	77
0	1	0 - $2 \cdot 1$	-2
1	-4	1 - $2 \cdot (-4)$	9
217	77	217 - $2 \cdot 77$	63

Kontynuując otrzymujemy $u' = -14$ i $v' = 61$.

$$945 \cdot (-14) + 217 \cdot 61 = (-13,230) + 13,237 = 7$$

Problem stanowią jednak liczby ujemne. Ponieważ chcieliśmy znaleźć odwrotność modulo pewnej liczby, nasz wynik musi być liczbą nieujemną. Rozwiązaniem jest użycie arytmetyki modularnej: znaleźliśmy u' i v', teraz znajdziemy $coeffU = u' \bmod v$ i $coeffV = v' \bmod u$. Jeśli u' jest ujemne, to $u' \bmod v$ jest równe $(u' + v) \bmod v$. W naszym przykładzie:

$$coeffU = -14 \bmod 217 = (-14 + 217) \bmod 217 = 203$$

$$coeffV = 61 \bmod 945 = 61$$

Okazuje się, że

$$(coeffU \cdot u) \bmod v = (coeffV \cdot v) \bmod u = NWD(u,v).$$

W przykładzie

$$(203 \cdot 945) \bmod 217 = 191{,}835 \bmod 217 = (191{,}835 - 884 \cdot 217) = 191{,}835 - 191{,}828 = 7 = NWD(945,217)$$

$$(61 \cdot 217) \bmod 945 = 13{,}237 \bmod 945 = (13{,}237 - 14 \cdot 945) = 13{,}237 - 13{,}230 = 7 = NWD(945,217)$$

Gdyby u i v były liczbami względnie pierwszymi, wtedy NWD(u, v) byłby równy 1. Zatem

$$(coeffU \cdot u) \bmod v = (coeffV \cdot v) \bmod u = 1.$$

To by zaś oznaczało, że $coeffU$ jest odwrotnością u modulo v, zaś $coeffV$ jest odwrotnością v modulo u.

W algorytmie RSA prywatny klucz d jest odwrotnością e (klucza publicznego) modulo $(p-1) \cdot (q-1)$. To zaś oznacza, że do wyznaczenia d idealnie nadaje się właśnie rozszerzony algorytm Euklidesa.

7 Faktoryzacja

Jednym ze sposobów rozłożenia liczby na czynniki pierwsze jest metoda Pollarda, opracowana w latach 70 przez Johna M. Pollarda, później ulepszona przez R. P. Brenta.

7.1 Metoda Pollarda

Przyjmując, że n jest liczbą, którą chcemy rozłożyć na czynniki pierwsze, algorytm przedstawia się następująco:

1. Wybieramy losowo dwie liczby $(\bmod n)$, x i y.
2. Czy różnica $x - y$ jest równa $0(\bmod n)$?
   - Tak - znaleźlimy czynnik, a mianowicie NWD($x - y, n$).
   - Nie - wracamy do kroku 1.

Oczywiście istnieją metody na szybkie generowanie „losowych” liczb $(\bmod n)$. Najprostszą metodą jest iterowanie z użyciem jakiegoś nieredukującego się wielomianu (z wyjątkiem $x^2 - 2$). Metoda Brenta wykorzystuje $x^2 - c, c \neq 2$.

7.2 Złożoność obliczeniowa

Teoretycznie metoda ta ma złożoność $O(\sqrt{r})$, gdzie r jest największym czynnikiem pierwszym wchodzącym w skład n. Przyjmując l jako długość (w bitach) klucza algorytmu RSA, liczby pierwsze p i q generujące klucz mają po $l/2$ bitów. Największy czynnik pierwszy w $n = p \cdot q$ będzie więc rzędu $2^{l/2}$. Złożoność faktoryzacji klucza publicznego będzie więc wynosić

$$O(\sqrt{2^{l/2}}) = O(2^{l/4}).$$

8 Implementacja

8.1 Działanie programu

Zaimplementowany program tworzy parę kluczy, następnie szyfruje za pomocą algorytmu RSA przykładową daną liczbową, która następnie jest odszyfrowana. Kolejnym krokiem jest odszyfrowanie danej tylko na podstawie klucza publicznego (faktoryzacja). Czas łamania jest mierzony i może być również wypisany na wyjście. Przykładowy wynik działania programu:

```
--------------------------
Ilosc bitow: 80
p = 898835777197
q = 844421896817
(p-1)(q-1) = 758996611905929877807936
Klucze:
d = 675528912340591733511595
e = 217026572659662567538627
n = 758996611907673135481949
Wiadomosc: 1234567890
Wiadomosc zaszyfrowana: 655561770502164031523330
Wiadomosc odszyfrowana: 1234567890

p = 898835777197
q = 844421896817
d = 675528912340591733511595
Wiadomosc 'zlamana': 1234567890
Czas lamania = 7.73s
```

8.2 Opcje

`-b` *n*	Określenie długości klucza w bitach (domyślnie 64).
`-t`	Wypisywanie czasu łamania klucza publicznego.
`-v`	Wypisywanie liczb pierwszych p i q, za pomocą których tworzone są klucze.

8.3 Opis implementacji

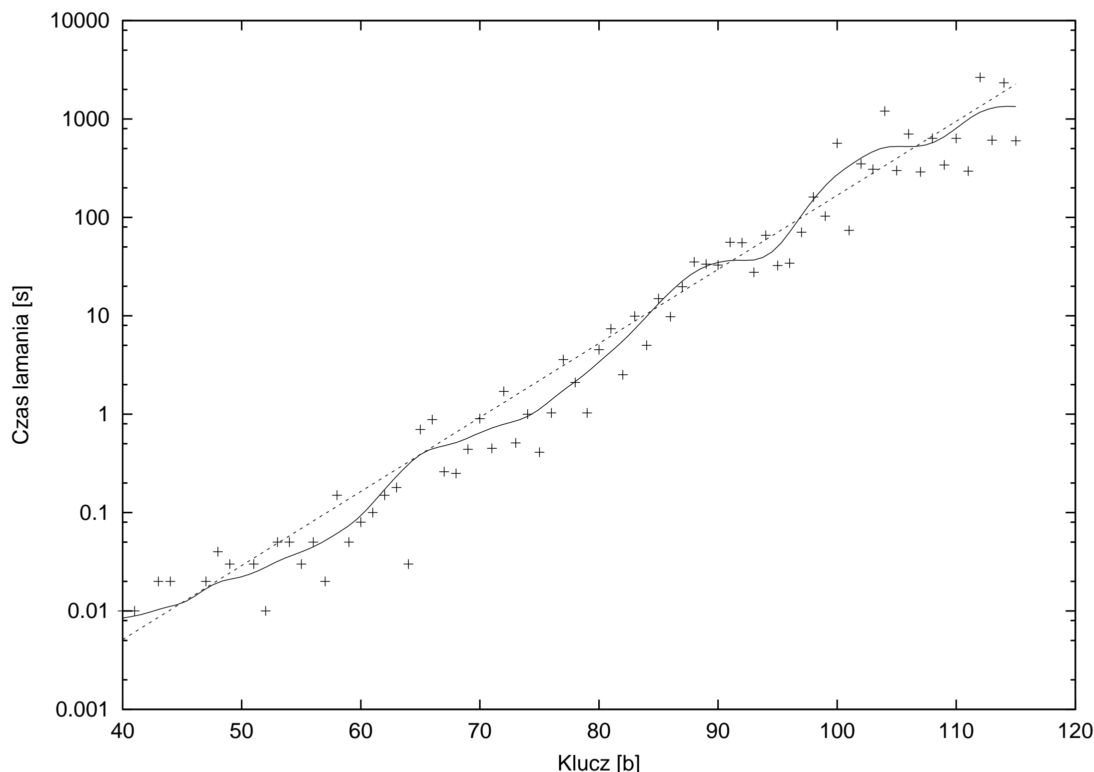
Program został zaimplementowany w języku C z wykorzystaniem biblioteki GMP 3.1.1 (GNU Multiple Precision Arithmetic Library). Biblioteka ta umożliwia operowanie na liczbach o dowolnej długości i oprócz podstawowych operacji arytmetycznych oferuje szeroką gamę dodatkowych funkcji. Wśród nich znajdują się funkcje przydatne przy implementowaniu RSA, jak np.: funkcja sprawdzająca czy dana liczba jest pierwszą, wykorzystując w tym celu test Millera-Rabina (`mpz_probab_prime_p()`), funkcja realizująca rozszerzoną wersję algorytmu Euklidesa (`mpz_gcdext()`), czy funkcja rozkładająca liczbę na czynniki pierwsze metodą faktoryzacji Pollarda (`factor()`).

8.4 Wyniki *łamania*

Wyznaczanie czasu łamania szyfru RSA pozwoliło na praktyczne określenie złożoności obliczeniowej tej czynności. Wykonano serię pomiarów czasu łamania klucza w zależności od długości klucza. Wynik pokazany jest na rysunku 1. Punktami oznaczono kolejne wyniki pomiarów, linia falista jest aproksymacją tychże wyników. Linia prosta natomiast przedstawia wykres funkcji

$$y = a \cdot 2^{\frac{x}{4}}, \qquad \text{gdzie } a = \frac{1}{2 \cdot 10^5}$$

Stała a została tak dobrana, aby wykres tej funkcji znajdował się w obszarze wyników pomiarów (ponieważ wykres jest w skali logarytmicznej, mnożenie funkcji wykładniczej przez stałą powoduje przesuwanie jej



Rysunek 1: Wielkość czasu łamania w zależności od długości klucza (skala logarytmiczna!).

wykresu w pionie). Czas łamania szyfru RSA w zależności od długości klucza jest więc równy

$$t = \frac{2^{\frac{L}{4}}}{2 \cdot 10^5} \quad [s]$$

gdzie L jest długością klucza w bitach.

Dla klucza o długości 100 bitów, czas ten wynosi więc 2 minuty 47 sekund. Dla 128 bitów - prawie 6 godzin. Dla 200 bitów ponad 178 lat, natomiast dla 300 bitów prawie 6 miliardów lat. Rozsądną wartością długości klucza, którego dałoby się przełamać na domowym PC są więc okolice 128 bitów.

Należy jednak wziąć pod uwagę następujące rzeczy:

- Podane wartości czasów są wartościami średnimi. Jak widać na wykresie, niejednokrotnie wyniki pomiarów znajdują się ponad wykresem funkcji wykładniczej, a to oznacza, że wartości mogą być *kilkakrotnie* większe (albo kilkakrotnie mniejsze - dla przeciwnego przypadku).
- Wzór został wyznaczony na podstawie badań tylko jednej funkcji faktoryzującej, związanej tylko z jedną biblioteką (GMP).
- Złamane zostały tylko klucze o długości do 115 bitów.

8.5 Sprzęt

Wszystkie testy były przeprowadzane na komputerze o następujących parametrach:

- procesor Intel Celeron 433Mhz
- 128MB pamięci operacyjnej
- 128kB pamięci podręcznej
- system operacyjny Linux (Slackware 7.0, jądro 2.2.13)

Bibliografia

[1] Steve Burnett; *The Mathematics of RSA*, 1996.
http://dslab.csie.ncu.edu.tw/~lucas/rsamath.HTM

[2] *RSA Algorithm Description*.
http://www.stanford.edu/~meverett/cs/rsa_description.html

[3] *Szyfrowanie*.
http://www.wsi.edu.pl/~dratewka/security/szyfrowanie.htm

[4] Francis Litterio; *The Mathematical Guts of RSA Encryption*.
http://world.std.com/~franl/crypto/rsa-guts.html

[5] *The RSA Algorithm*.
http://kulichki-win.rambler.ru/cryptology/rsa.html

[6] Steve Burnett; *How RSA works*.
http://www.kernel-panic.org/jaqque/rsamath.htm

[7] Grzegorz Blinowski; *Bezpieczeństwo danych w Internecie - Poczta elektroniczna*.
http://www.cc.com.pl/security/bezpmail.html

[8] *Carmichael numbers and Miller-Rabin Test*, 1999.
http://www.ma.umist.ac.uk/avb/117ws9.html

[9] Paul Herman; *Pollard's Rho Method*, 1997.
http://www.frenchfries.net/paul/factoring/theory/pollard.rho.html

A Kod źródłowy

```
/*
 * rsa.c - Przelamywanie szyfru RSA
 *
 * Marcin Rociek
 */
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>
#include "gmp.h"
#include "factorize.h"
#define TABLE_SIZE 30000
#define MAX_PRIME 1000

int verbose = 0;

/*
 * Funkcja wypisujaca liczbe
 */
void show(char *text, mpz_t number)
{
  printf("%s", text);
  mpz_out_str(stdout, 10, number);
  printf("\n");
}

/*
 * Funkcja znajdujaca najblizsza liczbe pierwsza
 * wieksza lub rowna danej (dowolnej) liczbie x
 * (na podstawie przykladu z GMP: demos/primes.c)
 */
int nearest_prime(mpz_t x)
{
  char *table;                        /* sito */
  ulong i, j, last, found;

  /* alokacja pamieci na sito */
  table = malloc(TABLE_SIZE);
  if (!table)
    return 0;

  /* znalezienie liczby pierwszej >= x */
  if ((mpz_get_ui(x) & 1) == 0)
    mpz_add_ui(x, x, 1);
  found = 0;
  while (!found)
  {
    memset(table, 1, TABLE_SIZE);
    for (i = 3; i < MAX_PRIME; i += 2)
    {
      ulong start;
      start = mpz_tdiv_ui(x, i);
      if (start != 0)
        start = i - start;

      if (mpz_cmp_ui(x, i - start) == 0)
        start += i;
      for (j = start; j < TABLE_SIZE; j += i)
        table[j] = 0;
    }
    last = 0;
    for (j = 0; j < TABLE_SIZE; j += 2)
    {
      if (table[j] == 0)
        continue;
      mpz_add_ui(x, x, j - last);
      last = j;
      if (mpz_probab_prime_p(x, 3))
      {
        found = 1;
        break;
      }
    }
    if (!found)
      mpz_add_ui(x, x, TABLE_SIZE - last);
  }

  /* zwolnienie pamieci */
  free(table);

  return 1;
}
```

```
/*
 * Funkcja tworzaca klucze (liczby e, d i n)
 */
void create_keys(mpz_t e, mpz_t d, mpz_t n, int bits)
{
  mpz_t p, q, p1q1, tmp1, tmp2;
  gmp_randstate_t state;              /* stan generatora liczb losowych */
  struct timeval tv;                  /* odczyt czasu */

  /* inicjalizacja generatora liczb losowych */
  gmp_randinit(state, GMP_RAND_ALG_LC, 128);
  gettimeofday(&tv, NULL);
  gmp_randseed_ui(state, (tv.tv_sec*tv.tv_usec));

  /* inicjalizacja zmiennych */
  mpz_init(p);
  mpz_init(q);
  mpz_init(p1q1);
  mpz_init(tmp1);
  mpz_init(tmp2);

  /* losowe wygenerowanie liczb pierwszych p i q */              /* krok 1 */
  mpz_urandomb(p, state, bits >> 1);
  mpz_urandomb(q, state, bits - (bits >> 1));
  nearest_prime(p);
  nearest_prime(q);

  if (verbose)
  {
    show("p = ", p);
    show("q = ", q);
  }

  /* wyliczenie n=pq */                                          /* krok 2 */
  mpz_mul(n, p, q);

  /* wyliczenie (p-1)(q-1) */                                    /* krok 3 */
  mpz_sub_ui(p, p, 1);
  mpz_sub_ui(q, q, 1);
  mpz_mul(p1q1, p, q);

  if (verbose)
    show("(p-1)(q-1) = ", p1q1);

  /* wyznaczenie e */
  mpz_set(tmp1, n);
  mpz_tdiv_ui(tmp1, 2);
  mpz_add_ui(tmp1, tmp1, 10);
  mpz_urandomm(e, state, tmp1);
  for (; ; )
  {
    mpz_gcd(tmp1, e, p1q1);
    if (mpz_cmp_ui(tmp1, 1) == 0)
      break;
    mpz_add_ui(e, e, 1);
  }

  /* wyznaczenie d */                                            /* krok 5 */
  mpz_gcdext(tmp1, d, NULL, e, p1q1);
  mpz_mod(d, d, p1q1);

  /* deinicjalizacja generatora liczb losowych */
  gmp_randclear(state);

  /* deinicjalizacja zmiennych */
  mpz_clear(p);
  mpz_clear(q);
  mpz_clear(p1q1);
  mpz_clear(tmp1);
  mpz_clear(tmp2);
}

/*
 * Funkcja szyfrujaca.
 * we: m - dana do zaszyfrowania
 *     e, n - klucz szyfrujacy
 * wy: c - dana zaszyfrowana
 */
void encrypt(mpz_t c, mpz_t m, mpz_t e, mpz_t n)
{
  /* podniesienie m do potegi e modulo n */
  mpz_powm(c, m, e, n);
}

/*
 * Funkcja deszyfrujaca.
```

```
 * we: c - dana zaszyfrowana
 *     d, n - klucz deszyfrujacy
 * wy: m - dana odszyfrowana
 */
void decrypt(mpz_t m, mpz_t c, mpz_t d, mpz_t n)
{
  /* podniesienie c do potego e modulo n */
  mpz_powm(m, c, d, n);
}

/*
 * Funkcja 'lamiaca' szyfr
 * we: c - dana zaszyfrowana
 *     e, n - klucz publiczny (szyfrujacy)
 * wy: m - dana odszyfrowana
 */
void attack(mpz_t m, mpz_t c, mpz_t e, mpz_t n)
{
  mpz_t p, q, p1q1, tmp;
  mpz_t d, n2;
  int n_ = 0;

  mpz_init(p);
  mpz_init(q);
  mpz_init(p1q1);
  mpz_init(tmp);
  mpz_init(d);
  mpz_init(n2);

  mpz_set(n2, n);

  /* rozlozenie n na czynniki pierwsze (czyli p i q) */
  factor(n, 0, p, q, &n_);

  if (verbose)
  {
    puts("");
    show("p = ", p);
    show("q = ", q);
  }

  /* wyliczenie (p-1)(q-1) */
  mpz_sub_ui(p, p, 1);
  mpz_sub_ui(q, q, 1);
  mpz_mul(p1q1, p, q);

  /* wyznaczenie d (klucza prywatnego) */
  mpz_gcdext(tmp, d, NULL, e, p1q1);
  mpz_mod(d, d, p1q1);

  if (verbose)
    show("d = ", d);

  /* podniesienie c do potegi d modulo n */
  mpz_powm(m, c, d, n2);

  mpz_clear(p);
  mpz_clear(q);
  mpz_clear(p1q1);
  mpz_clear(tmp);
  mpz_clear(d);
  mpz_clear(n2);
}

/*
 * Glowna funkcja prezentacyjna
 */
int rsa_show(int bits, int showtime)
{
  mpz_t e, d, n;                        /* klucze */
  mpz_t m;                              /* wiadomosc */
  mpz_t c;                              /* wiadomosc zaszyfrowana */
  mpz_t m2;                             /* wiadomosc po odszyfrowaniu */
  mpz_t m3;                             /* wiadomosc odszyfrowana (zlamana) */
  clock_t startc, stopc;                /* do mierzenia czasu lamania */
  time_t startt, stopt;
  float secs;

  /* inicjalizacja zmiennych */
  mpz_init(n);
  mpz_init(e);
  mpz_init(d);
  mpz_init(m);
  mpz_init(m2);
  mpz_init(m3);
  mpz_init(c);
```

```
  printf("---------------------------\n");
  printf("Ilosc bitow: %d\n", bits);

  /* tworzenie kluczy */
  create_keys(e, d, n, bits);
  printf("Klucze:\n");
  show("d = ", d);
  show("e = ", e);
  show("n = ", n);

  /* tworzenie jakiejs przykladowej danej wejsciowej */
  mpz_set_str(m, "1234567890", 10);
  mpz_mod(m, m, n);
  show("Wiadomosc: ", m);

  /* szyfrowanie */
  encrypt(c, m, e, n);
  show("Wiadomosc zaszyfrowana: ", c);

  /* odszyfrowanie */
  decrypt(m2, c, d, n);
  show("Wiadomosc odszyfrowana: ", m2);

  /* lamanie */
  startc = clock();
  startt = time(NULL);
  attack(m3, c, e, n);
  stopc = clock();
  stopt = time(NULL);
  show("Wiadomosc 'zlamana': ", m3);

  secs = (float)(stopt - startt);
  if (secs<100)
    secs = (float)(stopc - startc)/CLOCKS_PER_SEC;

  if (showtime)
    printf("Czas lamania = %.2fs\n", secs);

  /* deinicjalizacja zmiennych */
  mpz_clear(n);
  mpz_clear(e);
  mpz_clear(d);
  mpz_clear(m);
  mpz_clear(m2);
  mpz_clear(m3);
  mpz_clear(c);

  return 1;
}

/*
 * Czesc glowna programu
 */
int main(int argc, char **argv)
{
  int i;
  int bits = 64;                        /* domyslna ilosc bitow */
  int showtime = 0;                     /* czy wypisywac czas lamania */

  /* skanowanie parametrow */
  for (i=0; i<argc; i++)
  {
    if (argv[i][0]!='-')
      continue;
    switch (argv[i][1])
    {
    case 'b':                           /* ustawienie ilosci bitow */
    case 'B':
      if ((i + 1)<argc)
        sscanf(argv[i + 1], "%d", &bits);
      break;
    case 'v':                           /* wypisywanie p i q */
    case 'V':
      verbose = 1;
      break;
    case 't':                           /* wypisywanie czasu lamania */
    case 'T':
      showtime = 1;
      break;
    }
  }

  /* odpalenie glownej funkcji */
  rsa_show(bits, showtime);
}
```