Politechnika Białostocka

Sprawozdanie z pracowni specjalistycznej

Bezpieczeństwo Sieci Komputerowych

Temat: **Kryptosystemy symetryczne – algorytmy: DES, RSA.**

Wykonujący ćwiczenie: Paweł Szapiel

Studia dzienne

Kierunek: **INFORMATYKA**

Semestr: **VI**

Grupa zajęciowa: **6**

Prowadzący ćwiczenie: mgr inż. Dariusz Jankowski

Data wykonania ćwiczenia: **11.04.2022**

Zadanie 1. RSA

1. Treść zadania

Wykonaj program realizujący szyfrowanie i deszyfrowanie z wykorzystaniem algorytmu RSA.

1. Generowanie pary kluczy prywatnych i publicznych - 2 pkt
2. Obliczanie odwrotności modulo n - 2 pkt
3. Poprawne zaszyfrowanie wiadomości - 2 pkt
4. Poprawne deszyfrowanie wiadomości - 2 pkt
5. Obsługa plików binarnych - 1 pkt

2. Implementacja

Proces implementacji można podzielić na 3 etapy:

1. Wygenerowanie dwóch liczb pierwszych (dalej p i q)
2. Znalezienie odpowiednich parametrów (dalej e, d, k)
3. Szyfrowanie i deszyfrowanie.

1. Generowanie dwóch liczb pierwszych.

Do rozwiązania problemu użyłem powszechnie znanego algorytmu Rabina-Millera. Jest to test sprawdzający prawdopodobieństwo tego, że liczba jest pierwsza. W implementacji przyjmuję, że jeżeli test zakończy się powodzeniem 10-krotnie to można uznać, że liczba jest pierwsza.

Zakres generowanych liczb to 2 → 2^1023
W implementacji pomogły artykuły: [1] [2] [3] [4].

Kod źródłowy – Generowanie liczb pierwszych

```
        private static BigInteger FindPrimeNumber(int size, List<int> primesList)
        {
            BigInteger num;
            while (true)
            {
                num = GetNBitRandom(size); // Generuje pseudolosową liczbę o n bitach
                bool isDivisible = false;
                //check divisibility by pre-generated primes
                foreach (int prime in primesList)
                {
                    if (num % prime == 0 && num < prime)
                    {
                        isDivisible = true;
                        break;
                    }
                }
                if (!isDivisible) // jeżeli nie jest podzielna przez żadną z początkowych
liczb pierwszych
                {
                    if (IsMillerRabinPassed(num, 10)) // przeprowadź test millera rabina 20
razy
                        break;
                }
```

```
        }
        return num;
    }
    // pojedynczy zdany test daje statystycznie pewność 75% że liczba jest pierwsza
    private static bool IsMillerRabinPassed(BigInteger candidate, int numberOfTests)
    {
        int maxDivisionByTwo = 0;
        BigInteger evenComponent = candidate - 1;
        // dzielimy even component maksymalną liczbę razy przez dwa
        while (evenComponent % 2 == 0)
        {
            evenComponent /= 2; // shift left by one bit
            maxDivisionByTwo += 1;
        }

        RandomNumberGenerator rng = RandomNumberGenerator.Create();
        byte[] bytes = new byte[candidate.ToByteArray().LongLength];
        BigInteger randomNumber;
        // numberOfTests - im większe tym większa pewność że pierwsza
        for (int i = 0; i < numberOfTests; i++)
        {
            // generujemy losową liczbę większą od 2 ale mniejszą od evenComponent
            do
            {
                rng.GetBytes(bytes);
                randomNumber = new BigInteger(bytes);
            }
            while (randomNumber < 2 || randomNumber > candidate - 2);

            // x = randomNumber^evenComponent (mod candidate)
            BigInteger x = BigInteger.ModPow(randomNumber, evenComponent, candidate);
            if (x == 1 || x == candidate - 1)
                continue; // test zdany pomyślnie

            for (int r = 1; r < maxDivisionByTwo; r++)
            {
                // x = x^2 (mod candidate)
                x = BigInteger.ModPow(
                    x
                    , 2
                    , candidate);

                if (x == 1)
                    return false;
                if (x == candidate - 1)
                    break; // test zdany pomyślnie
            }

            if (x != candidate - 1)
                return false;
        }
        return true;
    }
```

2. Znalezienie odpowiednich parametrów e,d,k.

Dobieram parametry tak aby spełniały poniższe formuły:

```
n     = (p * q)
fi(n) = (p - 1) * (q - 1)
e     = GCD (fi(n), e) == 1 && e > 2
k     = (k * fi(n) + 1) % e == 0
```

`d     = (k * fi(n) + 1) / e`

Posiadając dwie liczby pierwsze p i q - ‘n’ i ‘fi(n)’ uzyskujemy natychmiast.
Parametr e dobieramy tak aby NWD(fi(n), e) == 1 i e > 2. Oczywiście możliwe jest uzyskanie wielu takich e, dlatego przyjmuje, że pierwsze 20 wystarczy aby wyliczyć prawidłowe d.
Liczba d musi być całkowita, dlatego potrzebne jest k. Znajduję takie k, że
`(k * fi(n) + 1) % e == 0 ,`
Następnie obliczam d ze wzoru `d = (k * fi(n) + 1) / e`
Ostatnim krokiem niejawnym (ponieważ nie przesyłamy nigdzie klucza) – jest ustalenie klucza prywatnego i publicznego.

Klucz publiczny	Klucz prywatny
d, n	e, n

Kod źródłowy – parametry e, d, k

```
        private static BigInteger[] Find_e(BigInteger fiOdN)
        {
            // znajduje 20 możliwych e mniejszych od fi(n)
            BigInteger[] possiblesE = new BigInteger[20];
            int iNum = 0;
            for (BigInteger i = new BigInteger(3);
                            iNum < 20 && i < fiOdN;
                            i = i + 2)
            {
                if (GCD(fiOdN, i) == 1)
                    possiblesE[iNum++] = i;
            }
            return possiblesE;
        }

        private static BigInteger GCD(BigInteger A, BigInteger B)
        {
            if (B.CompareTo(0) != 0)
            {
                return GCD(B, A % B);
            }
            return A;
        }

        private static void Generate_e_and_d(RsaModel m, BigInteger[] ePossibles)
        {
            for (int i = 0; i < ePossibles.Length; i++)
            {
                // dobierz k tak aby k*fi(n) + 1 było podzielne przez e
                for (int k = 0; k < 100; k++) // k może być różne i nie wpływa na trudność
złamania szyfru
                {
                    if ((k * m.fiOdN + 1) % ePossibles[i] == 0)
                    {
                        m.d = (k * m.fiOdN + 1) / ePossibles[i];
                        m.e = ePossibles[i];
                        m.k = k;
                        return;
                    }
                }
            }
        }
```

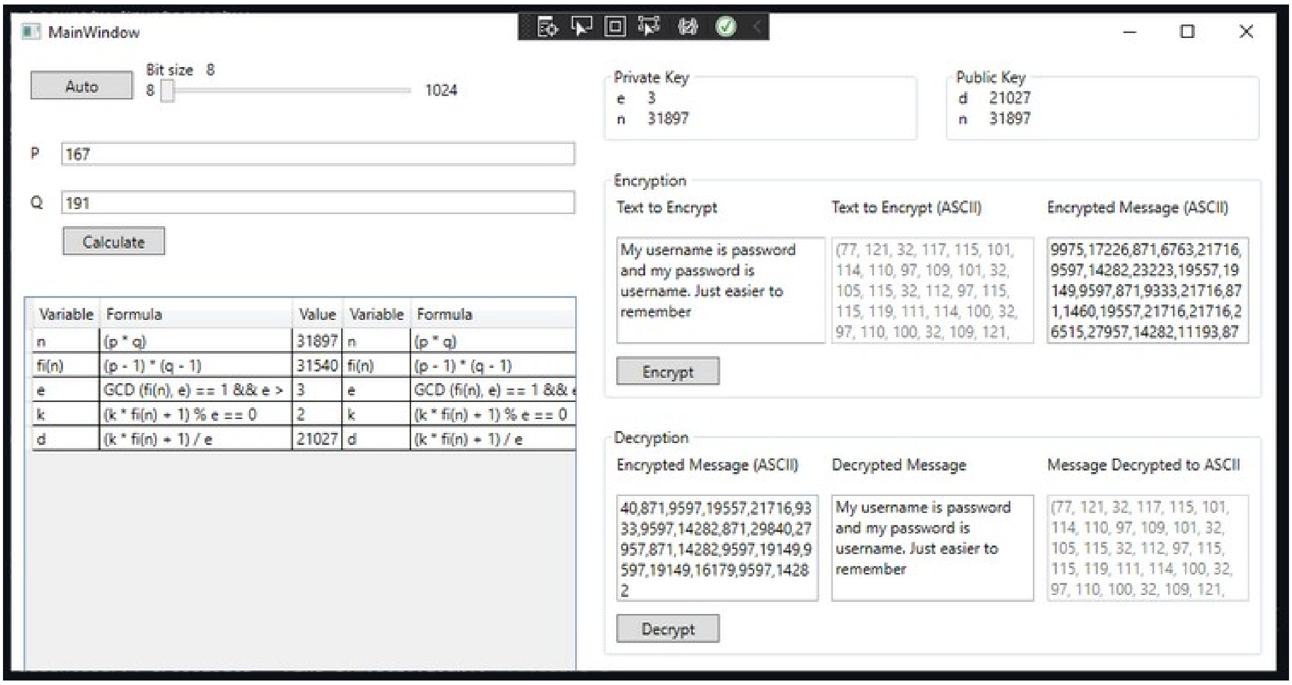
3. Szyfrowanie i deszyfrowanie

Obie powyższe operacje opierają się na podniesieniu do potęgi e lub d w modulo n.
Posiadając ciąg znaków zapisany w wartościach z tablicy ASCII można zaszyfrować oddzielnie każdy znak i zwrócić wynik.
Np. Jeżeli szyfrowanie było za pomocą e to deszyfrowanie odbędzie się używając d.

Kod źródłowy – szyfrowanie i deszyfrowanie

```
        private static BigInteger[] decryptRSA(BigInteger n, BigInteger d, BigInteger[]
encrypted)
        {
            BigInteger[] decrypted = new BigInteger[encrypted.Length];

            for (int i = 0; i < encrypted.Length; i++)
            {
                decrypted[i] = BigInteger.ModPow(encrypted[i], d, n);
            }

            return decrypted;
        }

        private static BigInteger[] encryptRSA(int[] converted, BigInteger n, BigInteger e)
        {
            BigInteger[] encrypted = new BigInteger[converted.Length];

            for (int i = 0; i < encrypted.Length; i++)
            {
                encrypted[i] = BigInteger.ModPow(converted[i], e, n);
            }

            return encrypted;
        }
```

3. GUI



Instrukcja:
1. Wybierz wielkość generowanych liczb pierwszych od 8 do 1024 bitów,
2. Lub wpisz liczby pierwsze samodzielnie.
3. Naciśnij Calculate.
4. W tabeli pokazują się wyliczone wartości, ustalone zostają klucze.
5. Wprowadź tekst do zaszyfrowania w polu ‘Text to Encrypt’
6. Wciśnij Encrypt.
7. Ukazuje się ciąg będący odpowiednikiem wiadomości w tablicy ASCII oraz zaszyfrowana wiadomość.
8. Skopiuj tekst z pola ‘Encrypted Message (ASCII)’ i wklej w pole o tej samej nazwie poniżej.
9. Wciśnij Decrypt.

Powinniśmy uzyskać zdeszyfrowaną wiadomość.

Wnioski

Największy problem sprawiło generowanie dostatecznie dużych liczb pierwszych. Własna implementacja, ze względu na zbyt dużą złożoność, nie jest w stanie generować liczb używanych komercyjnych 4096 bit, z prawdopodobieństwem 1/2^128.

Kolejny problem stanowiło dobranie liczby d i e w taki sposób, aby liczba d była całkowita. W rozwiązaniu pomogło ustalenie liczby k i założenie że nie może być większa od 100 oraz ograniczenie możliwych e do max. 20.

W wybraniu odpowiedniej technologii utwierdził mnie typ BigInteger, który znacząco ułatwił operacje arytmetyczne na liczbach.

Źródła:

[1] https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test
[2] https://www.geeksforgeeks.org/how-to-generate-large-prime-numbers-for-rsa-algorithm/
[3] https://www.geeksforgeeks.org/sieve-of-eratosthenes/
[4] http://rosettacode.org/wiki/Miller%E2%80%93Rabin_primality_test#C.23
[5] https://pl.khanacademy.org/computing/computer-science/cryptography
[6] https://pl.wikipedia.org/wiki/RSA_(kryptografia)