

1. Цель работы:

Целью нашей работы является изучение алгоритмов с открытым ключом. В данном случае рассматривается ассиметричный алгоритм шифрования RSA. Помимо изучения алгоритма, в лабораторной работе рассматривается постановка цифровой подписи (при постановке подписи использовать алгоритм MD5) с помощью RSA, а также атака повторным шифрованием.

2. Описание алгоритма и пояснения к заданиям:

RSA (аббревиатура от фамилий Rivest, Shamir и Adleman) — криптографический алгоритм с открытым ключом, основывающийся на вычислительной сложности задачи факторизации больших целых чисел.

Алгоритм создания публичного и секретного ключей

- 1) Выбираются два различных случайных простых числа p и q заданного размера
- 2) Вычисляется их произведение $n = p * q$, которое называется модулем
- 3) Вычисляется значение функции Эйлера от числа n :
$$\varphi(n) = (p - 1) * (q - 1)$$
- 4) Выбирается целое число e ($1 < e < \varphi(n)$), взаимно простое с $\varphi(n)$
- 5) Вычисляется число мультипликативно обратное к числу e по модулю n , то есть число, удовлетворяющее сравнению:
$$d * e \equiv 1 \pmod{\varphi(n)}$$
- 6) Пара (e, n) публикуется в качестве открытого ключа RSA
- 7) Пара (d, n) играет роль закрытого ключа RSA

Шифрование и расшифровка

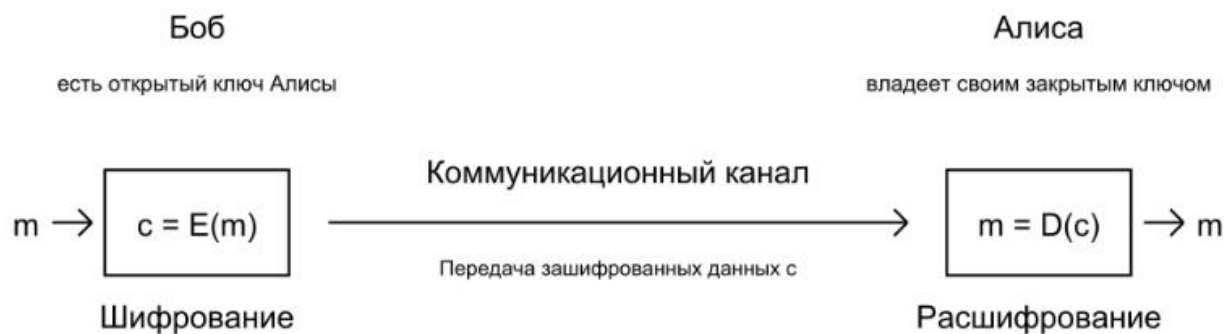


Рис.1 – Схема шифрования и расшифровки

1. Шифрование:

- 1) Взять открытый ключ (e, n) Алисы
- 2) Взять открытый текст m
- 3) Зашифровать сообщение, с помощью открытого ключа:
$$c = E(m) = m^e \bmod n$$

2. Расшифровка:

- 1) Принять зашифрованное сообщение c
- 2) Взять секретный ключ (d, n)
- 3) Применить закрытый ключ для расшифровки сообщения
$$m = D(c) = c^d \bmod n$$

Цифровая подпись



Рис.2 – Схема постановки цифровой подписи

Учитывая, что для цифровой подписи мы используем алгоритм MD5, то итоговый процесс не сильно отличается от шифрования/расшифровки:

1. Постановка подписи:

- 1) Взять открытый текст m
- 2) Создать цифровую подпись s , с помощью секретного ключа

$$s = S(m) = m^d \bmod n$$

3) Передать сообщение и подпись

2. Проверка подписи:

1) Принять зашифрованное сообщение и подпись

2) Взять открытый ключ

3) Вычислить прообраз сообщения из подписи

$$m' = P(s) = s^e \bmod n$$

4) Проверить подлинность подписи и неизменность сообщения сравнив m и m'

Атака повторным шифрованием

Строим последовательность: $y_n = y_{n-1}, y_i = y_{i-1}^e \bmod n, i > 1$

Итак, $y_m = y^{e^m} \bmod n$, а так как $(e, \varphi(n)) = 1$, то существует такое m , что $e^m = 1 \bmod \varphi(n)$. Но тогда $y^{(e^m-1)} = 1 \bmod n$, отсюда следует, что $y^{e^m} = y \bmod n$, значит y_{m-1} – решение сравнения $y = x^e \bmod n$.

Другими словами, мы возводим y в степень e до тех пор, пока не встретим $y' = y$. В таком случае предыдущий полученный y будет являться зашифрованным сообщением. При неудачных параметрах алгоритма RSA это серьёзно сократит перебор.

Описание алгоритма MD5

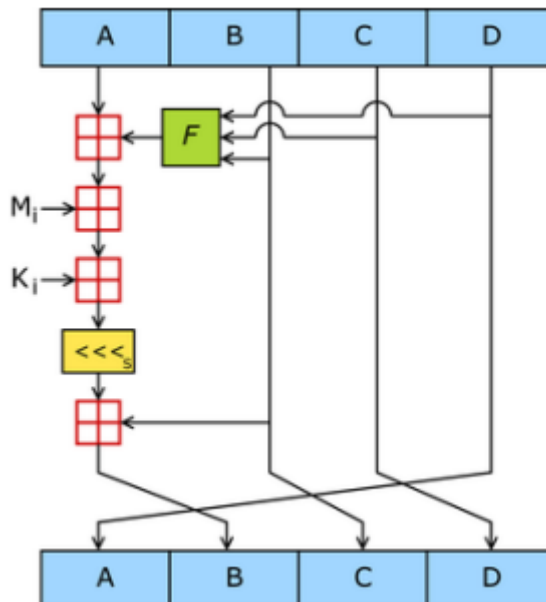


Рис.3 – Схема хеш-функции MD5

На вход алгоритма поступает входной поток данных, хеш которого необходимо найти. Длина сообщения измеряется в битах и может быть любой (в том числе нулевой). Запишем длину сообщения в L . Это число целое и неотрицательное. Кратность каким-либо числам необязательна. После поступления данных идёт процесс подготовки потока к вычислениям.

Шаг 1. *Выравнивание потока*

Сначала к концу потока дописывают единичный бит. Затем добавляют некоторое число нулевых бит такое, чтобы новая длина потока L стала сравнима с 448 по модулю 512 ($L' = 512 \times N + 448$). Выравнивание происходит в любом случае, даже если длина исходного потока уже сравнима с 448.

Шаг 2. *Добавление длины сообщения*

В конец сообщения дописывают 64-битное представление длины данных (количество бит в сообщении) до выравнивания. Сначала записывают младшие 4 байта, затем старшие. Если длина превосходит $2^{64} - 1$, то дописывают только младшие биты (эквивалентно взятию по модулю 2^{64}). После этого длина потока станет кратной 512. Вычисления будут основываться на представлении этого потока данных в виде массива слов по 512 бит.

Шаг 3. *Инициализация буфера*

Для вычислений инициализируются четыре переменные размером по 32 бита, начальные значения которых задаются шестнадцатеричными числами:

$A = 01\ 23\ 45\ 67$
 $B = 89\ AB\ CD\ EF$
 $C = FE\ DC\ BA\ 98$
 $D = 76\ 54\ 32\ 10$

В этих переменных будут храниться результаты промежуточных вычислений. Начальное состояние ABCD называется инициализирующим вектором.

Шаг 4. *Вычисление в цикле*

- 1) Для каждого раунда потребуется своя функция. Вводятся функции от трёх параметров — слов, результатом также будет слово:

$$\begin{aligned}\text{FunF}(X, Y, Z) &= (X \wedge Y) \vee (\neg X \wedge Z) \\ \text{FunG}(X, Y, Z) &= (X \wedge Z) \vee (\neg Z \wedge Y) \\ \text{FunH}(X, Y, Z) &= X \oplus Y \oplus Z\end{aligned}$$

$$\text{FunI} (X , Y , Z) = Y \oplus (\neg Z \vee X)$$

где \oplus , \wedge , \vee , \neg – побитовые логические операции XOR, AND, OR и NOT соответственно.

- 2) Определим таблицу констант $T [1 \dots 64]$ — 64-элементная таблица данных, построенная следующим образом: $T [n] = \text{int}(2^{32} \cdot |\sin n|)$
- 3) Каждый 512-битный блок проходит 4 этапа вычислений по 16 раундов. Для этого блок представляется в виде массива X из 16 слов по 32 бита. Все раунды однотипны и имеют вид: $[abcd \ k \ s \ i]$, определяемый как $a = b + ((a + \text{Fun}(b, c, d) + X[k] + T[i]) \lll s)$, где k — номер 32-битного слова из текущего 512-битного блока сообщения, и $\lll s$ — циклический сдвиг влево на s бит полученного 32-битного аргумента. Число s задается отдельно для каждого раунда.
- 4) Заносим в блок данных элемент n из массива 512-битных блоков. Сохраняются значения A , B , C и D , оставшиеся после операций над предыдущими блоками (или их начальные значения, если блок первый).

$AA = A$

$BB = B$

$CC = C$

$DD = D$

Этап 1

```
/* [abcd k s i] a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s). */
[ABCD 0 7 1][DABC 1 12 2][CDAB 2 17 3][BCDA 3 22 4]
[ABCD 4 7 5][DABC 5 12 6][CDAB 6 17 7][BCDA 7 22 8]
[ABCD 8 7 9][DABC 9 12 10][CDAB 10 17 11][BCDA 11 22 12]
[ABCD 12 7 13][DABC 13 12 14][CDAB 14 17 15][BCDA 15 22 16]
```

Этап 2

```
/* [abcd k s i] a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s). */
[ABCD 1 5 17][DABC 6 9 18][CDAB 11 14 19][BCDA 0 20 20]
[ABCD 5 5 21][DABC 10 9 22][CDAB 15 14 23][BCDA 4 20 24]
[ABCD 9 5 25][DABC 14 9 26][CDAB 3 14 27][BCDA 8 20 28]
[ABCD 13 5 29][DABC 2 9 30][CDAB 7 14 31][BCDA 12 20 32]
```

Этап 3

```
/* [abcd k s i] a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s). */
[ABCD 5 4 33][DABC 8 11 34][CDAB 11 16 35][BCDA 14 23 36]
[ABCD 1 4 37][DABC 4 11 38][CDAB 7 16 39][BCDA 10 23 40]
[ABCD 13 4 41][DABC 0 11 42][CDAB 3 16 43][BCDA 6 23 44]
[ABCD 9 4 45][DABC 12 11 46][CDAB 15 16 47][BCDA 2 23 48]
```

Этап 4

```
/* [abcd k s i] a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s). */  
[ABCD 0 6 49][DABC 7 10 50][CDAB 14 15 51][BCDA 5 21 52]  
[ABCD 12 6 53][DABC 3 10 54][CDAB 10 15 55][BCDA 1 21 56]  
[ABCD 8 6 57][DABC 15 10 58][CDAB 6 15 59][BCDA 13 21 60]  
[ABCD 4 6 61][DABC 11 10 62][CDAB 2 15 63][BCDA 9 21 64]
```

Суммируем с результатом предыдущего цикла:

A = AA + A

B = BB + B

C = CC + C

D = DD + D

После окончания цикла необходимо проверить, есть ли ещё блоки для вычислений. Если да, то переходим к следующему элементу массива ($n + 1$) и повторяем цикл.

Шаг 5. *Результат вычислений*

Результат вычислений находится в буфере ABCD, это и есть хеш. Если выводить побайтово, начиная с младшего байта A и заканчивая старшим байтом D, то мы получим MD5-хеш.

3. Программная реализация:

Процедура цифровой подписи

```
void digital_signature(string s) {  
    string m_hash = md5HashCalculating(s);  
    cout << "HASH: " << m_hash << endl;  
    cpp_int m = hash_to_int(m_hash);  
    //cpp_int m = 16137;  
    cout << "m: " << m << endl;  
    generate_key();  
    cpp_int m_encrypt = encrypt(m, d, n);  
    cout << "m_encrypt: " << m_encrypt << endl;  
    cpp_int m_decrypt = encrypt(m_encrypt, e, n);  
    cout << "m_decrypt: " << m_decrypt << endl;  
}
```

Рис.4 – Функция постановки подписи

Процедура генерации ключей

```
void generate_key() {  
    cpp_int max = cpp_int(5e32);  
    cpp_int min = cpp_int(2e23);  
    cpp_int p = generatePrime(min, max);  
    cpp_int q = generatePrime(min, max);  
    while (q == p) {  
        q = generatePrime(min, max);  
    }  
    cout << p << " --- " << q << endl;  
    n = p * q;  
    cpp_int phi = (p - 1) * (q - 1);  
    e = generatePrime(phi / 2, phi);  
    while (e == p || e == q) {  
        e = generatePrime(phi / 2, phi);  
    }  
  
    cout << "Public key: (" << e << " , " << n << ")" << endl;  
    cpp_int x = 1;  
    cpp_int y = 0;  
    gcd(e, phi, x, y);  
    if (x < 0) {  
        x = x + phi;  
    }  
    d = x;  
    cout << "Private key: (" << d << " , " << n << ")" << endl;  
}
```

Рис.5 – Генерация ключей

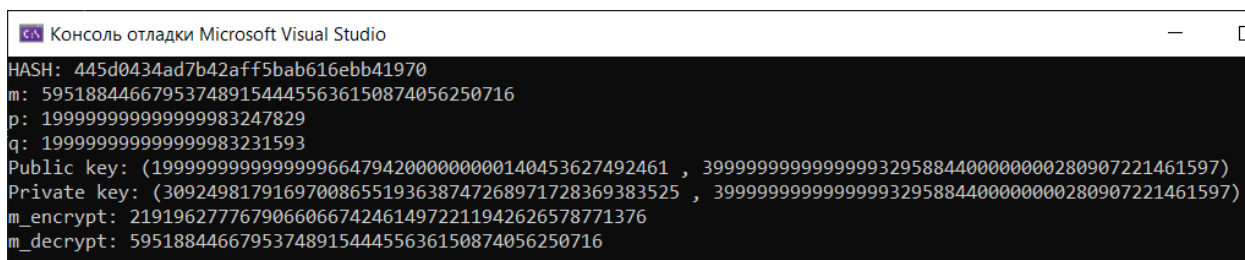
Атака на алгоритм

```
void reEncryptionAttack() {  
    cpp_int prev = 0, cur = 0;  
    e = 397;  
    n = 84517;  
    cpp_int m_encrypt = 8646;  
    m_encrypt = powmod(m_encrypt, e, n);  
    cur = 8646;  
    cout << m_encrypt << endl;  
    while (cur != m_encrypt) {  
        prev = m_encrypt;  
        m_encrypt = powmod(m_encrypt, e, n);  
        cout << m_encrypt << endl;  
    }  
    cout << prev;  
}
```

Рис.6 – Функция атаки на алгоритм

4. Результаты работы программы:

Пример работы алгоритма RSA (на примере цифровой подписи)

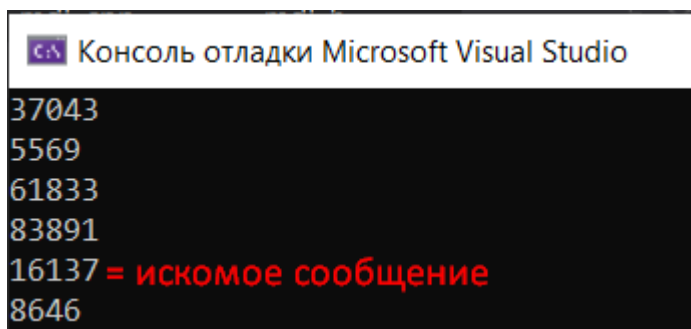


```
Консоль отладки Microsoft Visual Studio
HASH: 445d0434ad7b42aff5bab616ebb41970
m: 5951884466795374891544455636150874056250716
p: 1999999999999999983247829
q: 1999999999999999983231593
Public key: (199999999999999996647942000000000140453627492461 , 399999999999999993295884400000000280907221461597)
Private key: (30924981791697008655193638747268971728369383525 , 399999999999999993295884400000000280907221461597)
m_encrypt: 21919627776790660667424614972211942626578771376
m_decrypt: 5951884466795374891544455636150874056250716
```

Рис.7 – Пример работы алгоритма RSA

Атака повторным шифрованием

$$n = 84517, e = 397, y = 8646, d = 82225$$



```
Консоль отладки Microsoft Visual Studio
37043
5569
61833
83891
16137 = искомое сообщение
8646
```

Рис.8 – Атака повторным шифрованием

Таким образом, при неудачном выборе p и q , атака повторным шифрованием может серьёзно сократить вычисления.

5. Вывод:

Система RSA используется для защиты программного обеспечения и в схемах цифровой подписи. Также она используется в открытой системе шифрования PGP и иных системах шифрования в сочетании с симметричными алгоритмами. Из-за низкой скорости шифрования сообщения обычно шифруют с помощью более производительных симметричных алгоритмов со случайным сеансовым ключом (например, AES, IDEA, Serpent, Twofish), а с помощью RSA шифруют лишь этот ключ, таким образом реализуется гибридная криптосистема. Такой механизм имеет потенциальные уязвимости ввиду необходимости использовать криптографически стойкий генератор псевдослучайных чисел для формирования случайного сеансового ключа симметричного шифрования.

6. Литература:

- 1) А.Л. Чмора "Современная прикладная криптография"
- 2) Б.Я. Рябко "Основы современной криптографии для специалистов в ИТ"
- 3) Баричев, Серов "Основы современной криптографии"