Лабораторная работа №5

Ассиметричная криптография

Цель работы: Научиться реализовывать методы ассиметричной криптографии: генерацию пары ключей (открытый ключ, закрытый ключ), шифрование сообщения с использованием открытого ключа, дешифрование с использованием закрытого ключа.

Теоретическая часть

1. Алгоритм RSA

RSA – аббревиатура от сокращения фамилий 3-х ученых, разработавших эту систему – Рональда Райвеста, Ади Шамира и Леонарда Адельмана. Алгоритм был предложен в 1977 году. Он основывается на том, что нахождение больших простых чисел осуществляется сравнительно легко, но разложение на множители произведения двух таких чисел требует значительных вычислительных затрат.

Простым числом называется число из натурального ряда, большее 1, которое делится без остатка только на 1 и само на себя.

Аксиома. Число простых чисел бесконечно.

Число *x* называется *простым относительно y*, если его нельзя разложить на сомножители, на которые число *y* не делится без остатка. Например, число 4 является простым относительно 15, а число 6 не является простым относительно числа 15.

Числа *x* и *y* называются *взаимно простыми*, если наименьший общий делитель *НОД (x,y) =1*.

Малая теорема Ферма. Если P – простое число, то $x^{p-1} = 1 \pmod{p}$ для любого *x*, простого относительно P.

Следовательно:

$$x^{p-1} = 1 \pmod{p},$$

$$x^{p} \cdot x^{-1} = 1 \pmod{p},$$

$$\frac{x^{p}}{x} = 1 \pmod{p} \quad \left| \times x \right.$$

$$x^{p} = x \pmod{p}$$

Функцией Эйлера (n) называется число положительных целых меньших *n* и простых относительно *n*, на которые *n* не делится без остатка.

N	2	3	4	5	6	7	8	9	10	11	12
(n)	1	2	2	4	2	6	4	6	4	10	4

	1	1	1	1	1	1	1	1	1	1	1
		2	3	2	5	2	3	2	3	2	5
				3		3	5	4	7	3	7
С											
П				4		4	7	5	9	4	11
И						5		7		5	
С						6		8		6	
О										7	
К										8	
										9	
										10	

Теорема 2. Если $n = p \cdot q$, где p и q простые и не равны друг другу, то $(n) = (p-1)\cdot(q-1)$

Например, 3*5 = 15, (15) = 8, (3) = 2, (5) = 4.

Теорема 3. Если $n = p \cdot q$, где p и q простые и не равны друг другу, а x – простое относительно p и q, то $x \cdot (n) = 1 \pmod{p}$.

Из этого следует, что если *e* простое относительно *n*, то легко можно подобрать целое число *d*, такое, что $x \cdot d = 1 (\mathrm{mod}(n))$.

Алгоритм генерации ключей и шифрования/дешифрования

1. Отправитель выбирает два очень больших простых числа *P* и *Q* и вычисляет два произведения *N=P*Q и* M*=(P-1)*(Q-1)*.
2. Затем он выбирает случайное число *D*, взаимно простое с *M*, и вычисляет *E*, удовлетворяющее условию *D*E=1 mod M*.
3. После этого он публикует D и N как свой открытый ключ шифрования, сохраняя E (в паре с *N*) как закрытый ключ.
4. Теперь, чтобы зашифровать данные по известному ключу {*D, N}*, необходимо сделать следующее:
   a. разбить шифруемый текст на блоки, каждый из которых может быть представлен в виде числа *S(i)=0,1, ..., N-1*;
   b. зашифровать текст, рассматриваемый как последовательность чисел *S(i)* по формуле $S'(i)=(S(i)^D) \bmod N$.
5. Чтобы расшифровать эти данные, используют секретный ключ *{E, N}*, необходимо выполнить следующие вычисления:
   $S''(i)=S(i)=(S'(i)^E) \bmod N$.

6. В результате будет получено множество чисел *S''(i)*, которые представляют собой исходный текст.

2. Криптосистема Эль-Гамаля

Данная система является альтернативой алгоритму RSA и при равном значении ключа обеспечивает ту же криптостойкость. Метод Эль-Гамаля основан на проблеме дискретного логарифма. Если возводить число в степень в конечном поле достаточно легко, то восстановить аргумент по значению (то есть найти логарифм) довольно трудно.

Для генерации пары ключей сначала выбирается простое число *p* и два случайных числа, *g* и *x*, оба меньше *p*. Затем вычисляется

$$y = g^k \bmod p.$$

Открытым ключом являются *y*, *g* и *p*. И *g*, и *p* можно сделать общими для группы пользователей. Закрытым ключом является *x*.

Для шифрования сообщения *M* сначала выбирается случайное число *k,* взаимно простое с *p* - 1. Затем вычисляются

$$a = g^k \bmod p,$$

$$b = y^k M \bmod p.$$

Пара (*a*,*b*) является шифротекстом. Получаемый шифротекст в два раза длиннее открытого текста. Для дешифрирования (*a*,*b*) вычисляется

$$M = b/a^x \bmod p.$$

Так как $a^x \equiv g^{kx} \pmod p$ и $b/a^x \equiv y^k M/a^x \equiv g^{xk} M/ g^{kx} = M \pmod p$, то все работает. Или иначе:

$$a^x \bmod p = y^k \bmod p \rightarrow (g^k \bmod p)^x \bmod p = (g^x \bmod p)^k \bmod p.$$

Каждая подпись или шифрование EIGamal требует нового значения k, и это значение должно быть выбрано случайным образом. Если когда-нибудь Злоумышленник раскроет k, он сможет раскрыть закрытый ключ x. Если Злоумышленник когда-нибудь сможет получить два сообщения, подписанные или зашифрованные с помощью одного и того же k, то он сможет раскрыть x, даже не зная значение k.

Алгоритм генерации ключей и шифрования/дешифрования

Открытый ключ:

1. Выбрать простое число p (может быть общим для группы пользователей)
2. Выбрать g <p (может быть общим для группы пользователей)
3. Рассчитать y =g^x mod p.

Закрытый ключ:

1. Выбрать число x < p.

Шифрование:

1. *k* выбирается случайным образом, взаимно простое с *p*–1
2. *a* (шифротекст) =g^k mod *p,*

3. b (шифротекст)= $y^k M \bmod p$.

Дешифрирование:

1. $y = g^x \bmod p$,
2. $a = g^k \bmod p$,
3. $b = M \oplus (y*k \bmod p)$,
4. $M = (ax \bmod p) \oplus b$, *где* $\oplus$ *операция сложения по модулю 2.*

3. Алгоритмы для работы с большими числами

3.1. Алгоритм Рабина-Миллера

Тест Миллера -Рабина — вероятностный полиномиальный тест простоты, позволяющий проверить является ли заданное число простым или нет с определенной долей вероятности.

Алгоритм Рабина Миллера предлагает следующую схему генерации простого числа:

1. Выберите для проверки случайное число p. Вычислите b - число делений p - 1 на 2 (т.е., 2^b - это наибольшая степень числа 2, на которое делится p - 1). Затем вычислите m, такое что $p = 1 + 2^b * m$.
2. Выберите случайное число a, меньшее p.
3. Установите j = 0 и z = a*m mod p.
4. Если z = 1 или если z = p - 1, то p проходит проверку и может быть простым числом.
5. Если j > 0 и z = 1, то p не является простым числом.
6. Установите j = j + 1. Если j < b и $z \neq p - 1$, установите $z = z^2 \bmod p$ и вернитесь на этап (4). Если z = p - 1, то p проходит проверку и может быть простым числом.
7. Если j = b и $z \neq p - 1$, то p не является простым числом.

Гарантируется, что три четверти возможных значений a окажутся свидетелями. Это означает, что составное число проскользнет через t проверок с вероятностью не большей $(1/4)^t$, где t - это число итераций. На самом деле и эти оценки слишком пессимистичны. Для большинства случайных чисел около 99,9% возможных значений a являются свидетелями.

Для повышения эффективности работы алгоритмы генерации простого числа можно использовать следующий общий алгоритм.

1. Сгенерируйте случайное n-битовое число p
2. Установите старший и младший биты равными 1. (Старший бит гарантирует требуемую длину простого числа, а младший бит обеспечивает его нечетность)
3. Убедитесь, что p не делится на небольшие простые числа: 3, 5, 7, 11, и т.д. Во многих реализациях проверяется делимость p на все простые числа, меньшие 256. Наиболее эффективной является проверка на делимость для всех простых чисел, меньших 2000.
4. Выполните тест Rabin-Miller для некоторого случайного a. Если p проходит тест, сгенерируйте другое случайное a и повторите проверку. Выбирайте небольшие значения a для ускорения вычислений. Выполните пять тестов. (Одного может показаться достаточным, но выполните пять.) Если p не проходит одной из проверок, сгенерируйте другое p и попробуйте снова.

Можно не генерировать p случайным образом каждый раз, а последовательно перебирать числа, начиная со случайно выбранного до тех пор, пока не будет найдено простое число. Этап (3) не является обязательным, но это хорошая идея. Проверка, что случайное нечетное p не делится

на 3, 5 и 7 отсекает 54 процента нечетных чисел еще до этапа (4). Проверка делимости на все простые числа, меньшие 100, убирает 76 процентов нечетных чисел, проверка делимости на все простые числа, меньшие 256, убирает 80 процентов нечетных чисел. В общем случае, доля нечетных кандидатов, которые не делятся ни на одно простое число, меньшее n, равна 1.12/ln n. Чем больше проверяемое n, тем больше предварительных вычислений нужно выполнить до теста Rabin-Miller.

Пример алгоритма Рабина-Миллера на языке С (используем функцию pow_on_mod для возведение числа в степень по модулю – рассматривается ниже)

```
bool test_Miller_Rabin(long long m) {

	if(m==2 || m==3)
		return true;
	if(m % 2 == 0 || m == 1){
		return false;
	}
	long long s = 0;
	long long t = m-1;
	long long x=0;

	long long r1 = 2;
	long long r2 = m-2;
	long long a;

	long long r = (log(m) / log(2));

	while(t!=0 && t % 2 == 0){
		s++;
		t/=2;
	}

	for(long long  i = 0; i < r; i++) {
		a = r1 + rand()%(r2-r1);

		x = pow_on_mod(a,t,m);
		if(x==1 || x==m-1) {
			continue;
		}
		for(long long j = 0; j < s-1; j++) {
			x = pow_on_mod(x,2,m);

			if(x == 1)
				return false;
			if(x == m - 1)
					break;
		}

		if(x == m - 1)
			continue;
		return false;
	}

    return true;
}
```

3.2. Тест Лемана

Еще один тест для проверки простоты числа – это тест Лемана. Последовательность действий для проверки числа p:

1. Выберите случайное число a, меньшее p
2. Вычислите $a^{(p-1)/2}$ mod p
3. Если $a^{(p-1)/2}$ mod p ≠1 или -1, то p не является простым
4. Если $a^{(p-1)/2}$ mod p = 1 или -1, то вероятность того, что число p не является простым, не больше 50 процентов.

И снова, вероятность того, что случайное число a будет свидетелем составной природы числа p, не меньше 50 процентов. Повторите эту проверку t раз. Если результат вычислений равен 1 или -1, но не всегда равен 1, то p является простым числом с вероятностью ошибки $(1/2)^t$.

```
private static boolean lehmanTest(int p, int tries) {

        boolean isPrime = true;

        int a = randomGenerator();

        int e = (p - 1 )/2;
        int result = (a^e) % p;

        while (tries!=0)
        {
            if(result % p != 1 && result % p != p - 1)
            {
                a = randomGenerator();
                tries--;
            }
                else
                {
                    isPrime = false;
                }
        }
        return isPrime;
}
```

3.3. Алгоритм Евклида

Два числа называются взаимно простыми, если у них нет общих множителей кроме 1. Иными словами, если наибольший общий делитель a и n равен 1. Это записывается как:

НОД(a,n)=1

Взаимно просты числа 15 и 28, а вот числа 15 и 27 не являются взаимно простыми, 13 и 500 - являются. Простое число взаимно просто со всеми другими числами, кроме чисел, кратных данному простому числу. Одним из способов вычислить наибольший общий делитель двух чисел является алгоритм Эвклида.

На языке C, алгоритм выглядит так:

```
int gcd (int x, int y) {
   int g;
   if (x < 0)
         x = -x;
   if (y < 0)
```

```
            y = -y;
        if (x + y == 0 )
            ERROR ;
        g = y;
        while (x > 0) {
            g = x;
           x = y % x;
         y = g;
        }
        return g;
    }
```

3.4. Расширенный алгоритм Евклида

В общем случае у уравнения $a^{-1} \equiv x \pmod n$ существует единственное решение, если a и n взаимно просты. Если a и n не являются взаимно простыми, то $a^{-1} \equiv x \pmod n$ не имеет решений. Если n является простым числом, то любое число от 1 до n–1 взаимно просто с n и имеет в точности одно обратное значение по модулю n. Обратное значение a по модулю n можно вычислить с помощью расширенного алгоритма Эвклида

```
#pragma argsused
void swap(unsigned long &x, unsigned long &y)
{
 unsigned long t;
 t = x;
 x = y;
 y = t;
}

int even(unsigned long x)
{
 return ((x & 0x01) == 0);
}

int odd(unsigned long x)
{
 return (x & 0x01);
}

void ExtBinEuclid(unsigned long u, unsigned long v, unsigned long &result)
{
  int didswap = 0;
  unsigned long u1, u2, u3;
  unsigned long k, t1, t2, t3;

  if (even(u) && even(v)) return;

  if (u < v)
    {
      didswap = 1;
      swap (u, v);
    }

  for (k = 0 ; even(u) && even(v) ; k++)
    {
printf("%d && %d = %d\n", even(u), even(v), even(u) && even(v));
      u >>= 1;
      v >>= 1;
    }
```

```
  u1 = 1;
  u2 = 0;
  u3 = u;
  t1 = v;
  t2 = u - 1;
  t3 = v;
  do
    {
      do
        {
          if (even(u3))
            {
              if (odd(u1) || odd(u2))
                {
                  u1 += v;
                  u2 +=u;
                }
              u1 >>= 1;
              u2 >>= 1;
              u3 >>= 1;
            }
          if (even(t3) || (u3 < t3))
            {
              swap(u1, t1);
              swap(u2, t2);
              swap(u3, t3);
            }
        } while (even(u3));
      while ((u1 < t1) || (u2 < t2))
        {
          u1 += v;
          u2 += u;
        }
        u1 -= t1;
        u2 -= t2;
        u3 -= t3;

    } while (t3 > 0);
  while ((u1 >= v) && (u2 >= u))
    {
      u1 -= v;
      u2 -= u;
    }
   u1 <<= k;
   u2 <<= k;
   u3 <<= k;
   result = u - u2;
  if (didswap) swap(v, u);
}
```

3.5. Арифметика вычетов и возведение в степень в конечном поле

В системах с открытым ключом активно используется техника возведения в степень в конечном поле или, как ее еще называют, арифметика вычетов. Такой подход позволяет избежать ситуации переполнения разрядной сетки при работе с большими числами.

Арифметика вычетов очень похожа на обычную арифметику: она коммутативна, ассоциативна и дистрибутивна. Кроме того, приведение каждого промежуточного результата по модулю n дает тот же результат, как и выполнение всего вычисления с последующим приведением конечного результата по модулю n.

$$(a + b) \bmod n == ((a \bmod n) + (b \bmod n)) \bmod n$$
$$(a - b) \bmod n == ((a \bmod n) - (b \bmod n)) \bmod n$$
$$(a * b) \bmod n == ((a \bmod n) * (b \bmod n)) \bmod n$$
$$(a * (b+c)) \bmod n == (((a*b) \bmod n) + ((a*c) \bmod n)) \bmod n$$

Вычисление mod n часто используется в криптографии, так как вычисление дискретных логарифмов и квадратных корней mod n может быть нелегкой проблемой. Арифметика вычетов, к тому же, легче реализуется на компьютерах, поскольку она ограничивает диапазон промежуточных значений и результата. Для k-битовых вычетов n, промежуточные результаты любого сложения, вычитание или умножения будут не длиннее, чем 2^k бит.

Поэтому в арифметике вычетов мы можем выполнить возведение в степень без огромных промежуточных результатов. Вычисление степени некоторого числа по модулю другого числа, а x mod n, представляет собой просто последовательность умножений и делений, но существуют приемы, ускоряющие это действие. Один из таких приемов стремится минимизировать количество умножений по модулю, другой - оптимизировать отдельные умножения по модулю. Так как операции дистрибутивны, быстрее выполнить возведение в степень как поток последовательных умножений, каждый раз получая вычеты. Сейчас вы не чувствуете разницы, но она будет заметна при умножении 200-битовых чисел.

Например, если вы хотите вычислить a^8 mod n, не выполняйте наивно семь умножений и одно приведение по модулю:

$$(a * a * a * a * a * a * a * a) \bmod n$$

Вместо этого выполните три меньших умножения и три меньших приведения по модулю:

$$((a^2 \bmod n)^2 \bmod n)^2 \bmod n$$

Точно также,

$$a^{16} \bmod n = (((a^2 \bmod n)^2 \bmod n)^2 \bmod n)^2 \bmod n.$$

Вычисление a^x , где x не является степенью 2, ненамного труднее. Двоичная запись представляет x в виде суммы степеней двойки: 25 – это бинарное 11001, поэтому $25 = 2^4 + 2^3 + 2^0$. Поэтому:

$$a^{25} \bmod n = (a*a^{24}) \bmod n = (a* a^8 *a^{16}) \bmod n =$$
$$(a*((a^2)^2)^2 *(((a^2)^2)^2)^2) \bmod n = (a*(((a*a^2)^2)^2)^2) \bmod n$$

С продуманным сохранением промежуточных результатов вам понадобится только шесть умножений:

$$(((((((a^2 \bmod n)*a)^2 \bmod n)^2 \bmod n)^2 \bmod n)^2 \bmod n)^2 *a) \bmod n$$

Такой прием называется цепочкой сложений, или методом двоичных квадратов и умножения. Он использует простую и очевидную цепочку сложений, в основе которой лежит двоичное представление числа. На языке Си это выглядит следующим образом:

```
unsigned long qe2(unsigned long x, unsigned long y, unsigned long n) {
	unsigned long s, t, u;
	int i;
	s=1; t=x; u=y;
	while (u) {
		if(u&1) s=(s*t)%n;
		u>>1;
		t=(t*t)%n;
	}
	return(s)
}
```

Задание

Реализовать приложение с графическим интерфейсом, позволяющее выполнять следующие действия:

1. Генерировать простые числа от заданной границы вправо
2. Создавать пару: {Открытый ключ, Закрытый ключ}
3. Шифровать текстовые и двоичные файлы Открытым ключом с помощью ассиметричного алгоритма, указанного в варианте задания
4. Дешифровать текстовые и двоичные файлы Закрытым ключом с помощью ассиметричного алгоритма, указанного в варианте задания
5. Сохранять зашифрованные/дешифрованные данные в файл
6. Загружать зашифрованные/дешифрованные данные из файла

Дополнительные требования к приложению

- Программа должна быть оформлена в виде удобной утилиты
- Программа должна обрабатывать файлы любого размера и содержания
- Должна быть предусмотрена возможность просмотра сгенерированных ключей
- Текст программы оформляется прилично (удобочитаемо, с описанием ВСЕХ функций, переменных и критических мест).
- В процессе работы программа ОБЯЗАТЕЛЬНО выдает информацию о состоянии процесса шифрования / дешифрования.
- Интерфейс программы может быть произвольным, но удобным и понятным (разрешается использование библиотек GUI)
- Среда разработки и язык программирования могут быть произвольными.

Примечание: Задание является дифференцированным по сложности. На оценку «Удовлетворительно» при работе с простыми числами допускается использовать готовые алгоритмы и библиотеки, на оценку «хорошо» необходимо реализовать свои функции для работы с простыми числами, при этом длина простых чисел p и q должна быть не менее 32 бит. На оценку «отлично» необходимо реализовать свои функции для работы с простыми числами, при этом длина простых чисел p и q должна быть не менее 128 бит

Требования для сдачи лабораторной работы:

- Демонстрация работы реализованной вами системы.
- АВТОРСТВО
- Теория (ориентирование по алгоритмам и теоретическим аспектам методов тестирования)
- Оформление и представление письменного отчета по лабораторной работе, который содержит:

- Титульный лист
- Задание на лабораторную работу
- Описание используемых алгоритмов шифрования
- Листинг программы

Варианты задания

№	Алгоритм	Метод генерации простых чисел
1.	RSA	Тест Лемана
2.	Эль-Гамаля	Алгоритм Рабина-Миллера
3.	RSA	Алгоритм Рабина-Миллера
4.	Эль-Гамаля	Тест Лемана
5.	RSA	Тест Лемана
6.	Эль-Гамаля	Алгоритм Рабина-Миллера
7.	RSA	Алгоритм Рабина-Миллера
8.	Эль-Гамаля	Тест Лемана
9.	RSA	Тест Лемана
10.	Эль-Гамаля	Алгоритм Рабина-Миллера
11.	RSA	Алгоритм Рабина-Миллера
12.	Эль-Гамаля	Тест Лемана
13.	RSA	Тест Лемана
14.	Эль-Гамаля	Алгоритм Рабина-Миллера
15.	RSA	Алгоритм Рабина-Миллера