**Министерство науки и высшего образования Российской Федерации**
**Федеральное государственное бюджетное образовательное учреждение высшего образования**
**«Московский государственный технический университет имени Н.Э. Баумана**
**(национальный исследовательский университет)»**
**(МГТУ им. Н.Э. Баумана)**

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

# Отчет по лабораторной работе №4

# по курсу "Защита информации"

**Тема** Создание и проверка электронной подписи для документа

**Студент** Золотухин А.В.

**Группа** ИУ7-74Б

**Преподаватели** Чиж И.С.

Москва, 2023 г.

## СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

Цель данной лабораторной работы — реализовать программу создания и проверки электронной подписи для документа с использованием алгоритма RSA и алгоритма хеширования SHA1.

Информацию для разных целей пытались засекречивать с помощью шифрования на протяжении всей истории человечества. Шифр — это множество обратимых преобразований открытого текста, проводимых с целью его защиты от несанкционированного использования. Одним из таких шифров является RSA.

В рамках выполнения лабораторной работы необходимо решить следующие задачи:

— описать алгоритмы RSA и SHA1;

— реализовать алгоритмы RSA и SHA1 для документа.

# 1 Аналитический раздел

## 1.1 Алгоритм шифрования RSA

RSA (Rivest-Shamir-Adleman) — это криптографический алгоритм, используемый для шифрования и подписи данных.

В отличии от симметричных алгоритмов шифрования, имеющих всего один ключ для шифрования и расшифровки информации, в алгоритме RSA используется 2 ключа — открытый (публичный) и закрытый (приватный).

В ассиметричной криптографии и алгоритме RSA, в частности, публичный и приватный ключи являются двумя частями одного целого и неразрывны друг с другом. Для шифрования информации используется открытый ключ, а для её расшифровки приватный.

Рассмотрим процедуру создания публичного и приватного ключей:

1. Выбираем два случайных простых числа $p$ и $q$.

2. Вычисляем их произведение: $N = p * q$.

3. Вычисляем функцию Эйлера: $\varphi(N) = (p - 1) * (q - 1)$

4. Выбираем число $e$ (обычно простое, но необязательно), которое меньше $\varphi(N)$ и является взаимно простым с $\varphi(N)$.

5. Ищем число $d$, обратное числу $e$ по модулю $\varphi(N)$, т.е. остаток от деления $(d * e)$ и $\varphi(N)$ должен быть равен 1.

После произведённых вычислений, у нас будут: $e$ и $n$ — открытый ключ, $d$ и $n$ — закрытый ключ.

Алгоритм RSA широко используется в криптографии для обеспечения конфиденциальности данных и аутентификации. Однако для его безопасной реализации необходимо выбирать достаточно большие ключи, чтобы обеспечить защиту от взлома при помощи методов факторизации.

## 1.2 Алгоритм хеширования SHA1

Алгоритм шифрования SHA-1 (Secure Hash Algorithm 1) является одним из членов семейства криптографических хеш-функций, разработанных для обеспечения безопасной хеширования данных. SHA-1 преобразует входные данные произвольной длины в фиксированный хеш-код длиной 160 бит (20 байт).

Шаги алгоритма SHA-1 включают в себя:

— Инициализацию пяти 32-битных переменных, используемых для сохранения промежуточных результатов.

— Предварительную обработку входных данных, включая добавление бита заполнения и добавление длины сообщения.

— Разделение входных данных на блоки фиксированной длины и последовательное применение операций на каждом блоке, включая логические операции, сдвиги и битовые операции.

— Получение конечного 160-битного хеш-кода.

SHA-1 был широко использован в различных криптографических приложениях, однако с течением времени был выявлен ряд уязвимостей и стал уступать более современным алгоритмам хеширования, таким как SHA-256 или SHA-3, из-за возможности коллизий (ситуации, когда разным входным данным соответствует одинаковый хеш-код). В связи с этим, рекомендуется использовать более сильные алгоритмы хеширования в криптографических приложениях.

### Вывод

В данном разделе были рассмотрены алгоритмы RSA и SHA1.

# 2 Конструкторский раздел

## 2.1 Алгоритм шифрования RSA

Схема алгоритма изображена на рисунке 2.1.

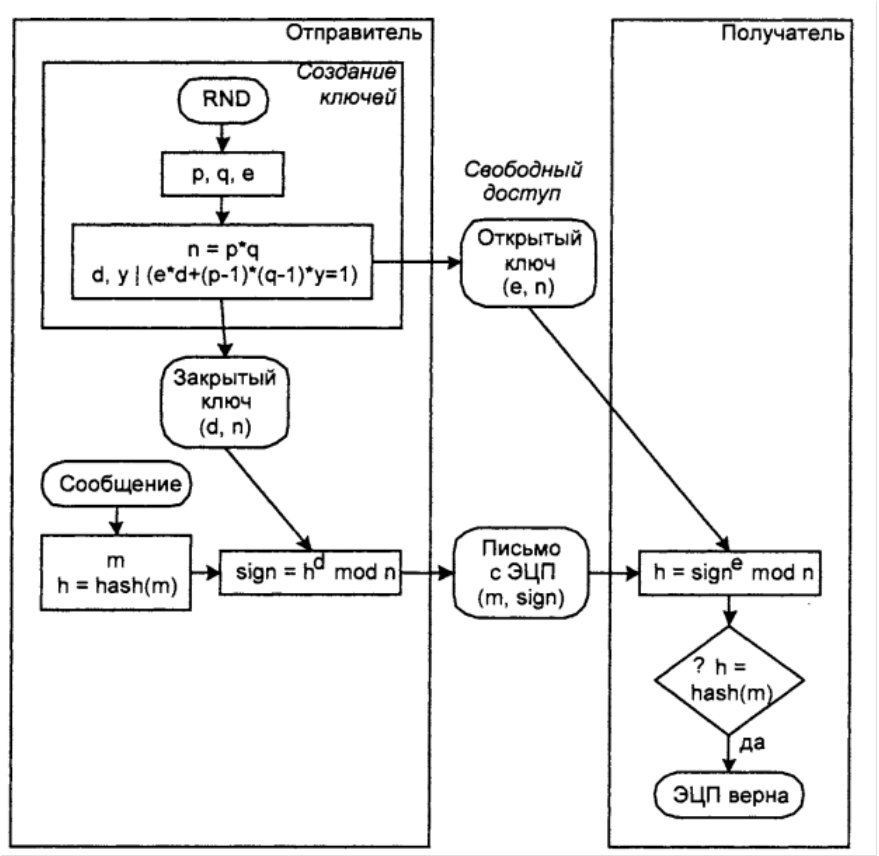


Рисунок 2.1 — Схема алгоритма шифрования RSA

## 2.2 Алгоритм хеширования SHA1

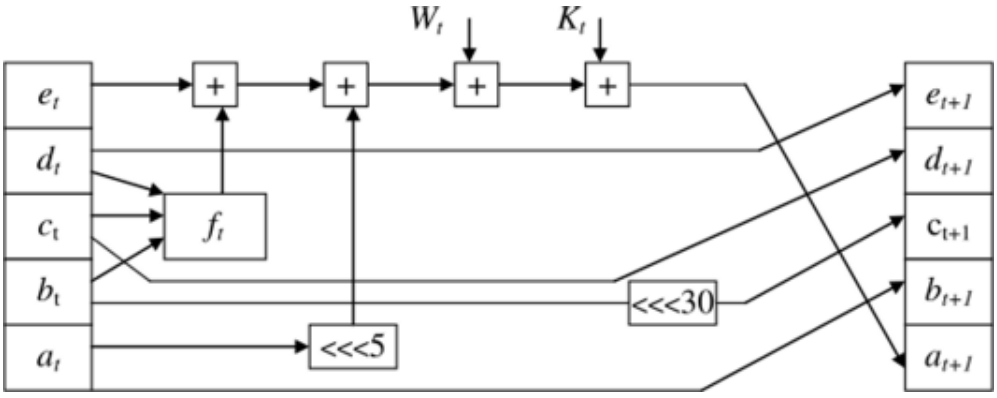


Рисунок 2.2 — Схема алгоритма хеширования SHA1

**Вывод**

В данном разделе были приведены схемы алгоритмов RSA и SHA1.

## 3 Технологический раздел

### 3.1 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы использовался язык программирования C [1], так как он позволяет работать с файлами и массивами. В качестве среды разработки использовалась Visual Studio [2].

### 3.2 Реализация алгоритмов

В листингах 3.1–3.2 представлены реализации алгоритмов RSA и SHA1.

Листинг 3.1 — Функция шифрования

```
1  static void bignum_pow_mod(bignum* result, bignum* base, bignum* expo,
       bignum* mod)
2  {
3      bignum *a = bignum_alloc(), *b = bignum_alloc();
4      bignum *tmp = bignum_alloc();
5
6      bignum_copy(base, a);
7      bignum_copy(expo, b);
8      bignum_fromint(result, 1);
9
10     while(!bignum_iszero(b))
11     {
12         if(b->data[0] & 1)
13         {
14             bignum_imultiply(result, a);
15             bignum_imod(result, mod);
16         }
17         bignum_idivide_2(b);
18         bignum_copy(a, tmp);
19         bignum_imultiply(a, tmp);
20         bignum_imod(a, mod);
21     }
22     bignum_free(a);
23     bignum_free(b);
24     bignum_free(tmp);
25 }
```

Листинг 3.2 — Функция хеширования

```
1  int calculate_sha1(struct sha* sha1, unsigned char* text, uint32_t length)
2  {
3      unsigned int i, j;
4      unsigned char* buffer;
5      uint32_t bits;
```

```
 6       uint32_t temp, k;
 7       uint32_t lb = length * 8;
 8
 9       bits = padded_length_in_bits(length);
10       buffer = (unsigned char*)malloc((bits / 8) + 8);
11       if (buffer == NULL)
12       {
13           printf("\nError allocating memory...");
14           return 1;
15       }
16       memcpy(buffer, text, length);
17       *(buffer + length) = 0x80;
18       for (i = length + 1; i < (bits / 8); i++)
19       *(buffer + i) = 0x00;
20       *(buffer + (bits / 8) + 4 + 0) = (lb >> 24) & 0xFF;
21       *(buffer + (bits / 8) + 4 + 1) = (lb >> 16) & 0xFF;
22       *(buffer + (bits / 8) + 4 + 2) = (lb >> 8) & 0xFF;
23       *(buffer + (bits / 8) + 4 + 3) = (lb >> 0) & 0xFF;
24       sha1->digest[0] = 0x67452301;
25       sha1->digest[1] = 0xEFCDAB89;
26       sha1->digest[2] = 0x98BADCFE;
27       sha1->digest[3] = 0x10325476;
28       sha1->digest[4] = 0xC3D2E1F0;
29       for (i = 0; i < ((bits + 64) / 512); i++)
30       {
31           for (j = 0; j < 80; j++)
32           sha1->w[j] = 0x00;
33           for (j = 0; j < 16; j++)
34           {
35               sha1->w[j] = buffer[j * 4 + 0];
36               sha1->w[j] = sha1->w[j] << 8;
37               sha1->w[j] |= buffer[j * 4 + 1];
38               sha1->w[j] = sha1->w[j] << 8;
39               sha1->w[j] |= buffer[j * 4 + 2];
40               sha1->w[j] = sha1->w[j] << 8;
41               sha1->w[j] |= buffer[j * 4 + 3];
42           }
43           for (j = 16; j < 80; j++)
44           sha1->w[j] = (ROTL(1, (sha1->w[j - 3] ^ sha1->w[j - 8] ^ sha1->w[j
                 - 14] ^ sha1->w[j - 16])));
45           sha1->a = sha1->digest[0];
46           sha1->b = sha1->digest[1];
47           sha1->c = sha1->digest[2];
48           sha1->d = sha1->digest[3];
49           sha1->e = sha1->digest[4];
50           for (j = 0; j < 80; j++)
51           {
```

```
52              if ((j >= 0) && (j < 20))
53              {
54                  sha1->f = ((sha1->b) & (sha1->c)) | ((~(sha1->b)) &
                        (sha1->d));
55                  k = 0x5A827999;
56              }
57              else if ((j >= 20) && (j < 40))
58              {
59                  sha1->f = (sha1->b) ^ (sha1->c) ^ (sha1->d);
60                  k = 0x6ED9EBA1;
61              }
62              else if ((j >= 40) && (j < 60))
63              {
64                  sha1->f = ((sha1->b) & (sha1->c)) | ((sha1->b) & (sha1->d))
                        | ((sha1->c) & (sha1->d));
65                  k = 0x8F1BBCDC;
66              }
67              else if ((j >= 60) && (j < 80))
68              {
69                  sha1->f = (sha1->b) ^ (sha1->c) ^ (sha1->d);
70                  k = 0xCA62C1D6;
71              }
72              temp = ROTL(5, (sha1->a)) + (sha1->f) + (sha1->e) + k +
                    sha1->w[j];
73              sha1->e = (sha1->d);
74              sha1->d = (sha1->c);
75              sha1->c = ROTL(30, (sha1->b));
76              sha1->b = (sha1->a);
77              sha1->a = temp;
78              temp = 0x00;
79          }
80          sha1->digest[0] += sha1->a;
81          sha1->digest[1] += sha1->b;
82          sha1->digest[2] += sha1->c;
83          sha1->digest[3] += sha1->d;
84          sha1->digest[4] += sha1->e;
85
86          buffer = buffer + 64;
87      }
88      return 0;
89  }
```

### 3.3 Тестирование реализации алгоритма

Было проведено тестирование на следующих входных данных:

— пустой файл;

— текстовый файл;

— файл формата jpg;

— архив формата zip.

**Для пустого файла:**

Generated digest = 156048787866750836371052389361754228214758992386

Encrypted SHA = 176338054747182782702536097907593539025900695606382214856988428006496363222282605794483616231511818699319034210730249190497793166181820421273123757605696717030651934714044414957236542357355407952218471077927979353051991088821577338569592513073924902245634460785461973450641955434587915590487749 88

Decrypted Hash value = 156048787866750836371052389361754228214758992386

**Для тестового файла:**

Generated digest = 814791834312585864639464326153728085570323449260

Encrypted SHA = 421570812774064700130738501290455547914727460910589217716107234805835086170562220821845851596072042066134630659498440170300943046074973320070711053256196598488836738495604539471052256166033965794147838789437108680101714749567869059700474703674024680137657875079935314070724158174300219014058532 77

Decrypted Hash value = 814791834312585864639464326153728085570323449260

**Для файла формата jpg:**

Generated digest = 140902034080479019990595221960460975206787978420 0

Encrypted SHA = 191032064883149011936027216760268191432287911921288685012914925477497973086705427847598678406563310651198711257459093958054454447741305203252219947853525710090752612120343753877559752490777051098022217342828374220983531254311306035101249878913166019686601354148487294865947863329746918686707187 31

Decrypted Hash value = 140902034080479019990595221960460975206787978420 0

**Для файла формата zip:**

Generated digest = 473167284371037228757425872343191003776458840519

Encrypted SHA = 513579172498203763282726096907604655931937679980854524702901793365158166060728321979115667959984425233852273577588928705404371437349727508750971420668757953741118428997925602487163445962751059606487135718388813445490737265872033574366986597129946288773850787714763848886540811122028532477725999 12

Decrypted Hash value = 473167284371037228757425872343191003776458840519

Для каждого из файлов был сформирован хэш. Хэш после дешифрования совпадает с хешом, поданным на вход функции дешифратору.

Все тесты пройдены успешно.

### Вывод

В данном разделе были перечислены средства разработки, с помощью которых был реализованы алгоритмы RSA и SHA1, приведена реализация алгоритмов.

## ЗАКЛЮЧЕНИЕ

В результате выполнения данной лабораторной работы была достигнута цель работы: реализована программа создания и проверки электронной подписи для документа с использованием алгоритма RSA и алгоритма хеширования SHA1.

Были решены все задачи — описаны и реализованы алгоритмы RSA и SHA1.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Документация языка C++ [Электронный ресурс]. — Режим доступа: `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf` (дата обращения: 13.11.2022).

2. Visual Studio Code [Электронный ресурс]. — Режим доступа: `https://code.visualstudio.com/docs` (дата обращения: 20.09.2022).