Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

«Исследование алгоритмов генерации

и верификации ЭЦП»

Студент: Белицкий В.Д.

ФИТ 3 курс 5 группа

Преподаватель:

Савельева Маргарита Геннадьевна

Минск 2023

## 1. Описание приложения

Приложение позволяет выполнить следующие задачи:

* генерация ЭЦП на основе алгоритмов RSA, Эль-Гамаля и Шнора;
* оценка времени выполнения указанных процедур при реальных параметрах;
* Приложение использует язык программирования C#.

## 2. Методика выполнения поставленных задач

## 2.1. Алгоритм RSA

Создание ЭЦП (Электронной Цифровой Подписи) на основе RSA включает следующие шаги:

1. Генерация ключей:

* Сгенерируйте два случайных простых числа *p* и *q*.
* Вычислите модуль n как произведение *p* и *q*: *n* = *p* × *q*.
* Вычислите значение функции Эйлера φ(*n*) = (*p* - 1) × (*q* - 1).
* Выберите открытую экспоненту e, которая является взаимно простой с φ(*n*) и меньше φ(*n*).
* Вычислите закрытую экспоненту *d*, которая является мультипликативно обратной к e по модулю φ(*n*): *d* ≡ (mod φ(*n*)).
* Открытый ключ: (*n*, e)
* Закрытый ключ: (*n*, *d*)

1. Подписывание сообщения:

* Преобразуйте сообщение в хэш-значение с использованием хэш-функции, например, SHA-256.
* Зашифруйте хэш-значение с помощью закрытого ключа RSA: signature = (mod *n*).
* Полученное значение signature представляет собой ЭЦП для сообщения.

1. Проверка подписи:

* Получите открытый ключ RSA (*n*, *e*) отправителя.
* Расшифруйте ЭЦП с использованием открытого ключа RSA: decrypted\_hash = ×(mod *n*).
* Вычислите хэш-значение исходного сообщения с использованием той же хэш-функции.
* Сравните полученное хэш-значение с расшифрованным хэш-значением. Если они совпадают, подпись считается верной.

Ограничения:

При генерации ключей необходимо выбирать достаточно большие простые числа p и q для обеспечения безопасности RSA.

Размер модуля n должен быть достаточно большим, например, 2048 бит или более, для обеспечения защиты от атак по факторизации.

Выбор открытой экспоненты e должен быть осуществлен с учетом безопасности и стандартных рекомендаций, например, использование простых чисел или значения 65537.

Хэш-функция, используемая для преобразования сообщения в хэш-значение, должна быть стойкой к коллизиям и рекомендованной к использованию, например, SHA-256.

Реализация генерации ЭЦП представлена на листинге 2.1.

public List<string> Encode(string hash, int e, int n)

{

List<string> result = new List<string>();

foreach (char c in hash)

{

int index = Array.IndexOf(characters, c);

BigInteger bi = BigInteger.ModPow(new BigInteger(index), e, n);

result.Add(bi.ToString());

}

return result;

Рисунок 2.1 – Генерация ЭЦП RSA

Реализация генерации хеша на стороне получателя для дальнейшего сравнения с полученным хешем представлена на рисунке 2.2.

public string Decode(List<string> input, int d, int n)

{

try

{

string result = "";

foreach (string item in input)

{

BigInteger bi = BigInteger.ModPow(new BigInteger(Convert.ToDouble(item)), d, n);

int index = Convert.ToInt32(bi.ToString());

result += characters[index].ToString();

}

return result;

}

catch (Exception)

{

return "";

}

}

Рисунок 2.2 – Генерация хэша для проверки ЭЦП

Результат работы приложения представлен на рисунке 2.3.

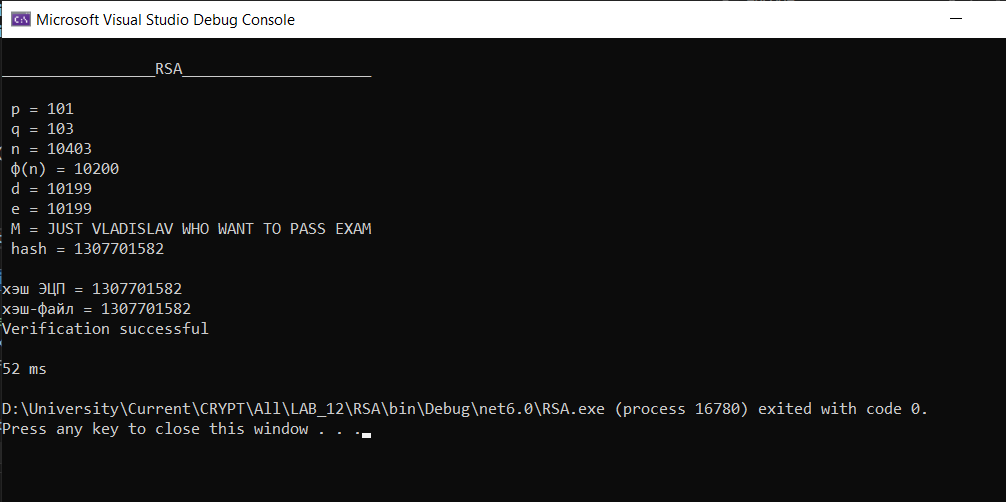


Рисунок 2.3 – Результат работы алгоритма RSA

## 2.2. Алгоритм Эль-Гамаля

Алгоритм ЭЦП (Электронной Цифровой Подписи) на основе схемы Эль-Гамаля включает следующие шаги:

1. Генерация ключей:

* Выберите простое число *p*.
* Выберите первообразный корень g по модулю *p*.
* Выберите случайное число x, такое что 1 <= *x* <*p* -1.
* Вычислите *y* = mod *p*.
* Открытый ключ: (*p*, g, *y*)
* Закрытый ключ: *x*

1. Подписывание сообщения:

* Преобразуйте сообщение в хэш-значение с использованием хэш-функции, например, SHA-256.
* Выберите случайное *k*, такое что 1 <= *k* < *p* -1 и НОД(*k*, *p* -1) = 1.
* Вычислите *r* = mod *p*.
* Вычислите *s* = (hash + *x* ×*r*) × mod (*p* -1).
* Полученная подпись: (*r*, *s*)

1. Проверка подписи:

* Получите открытый ключ Эль-Гамаля (*p*, *g*, *y*) отправителя.
* Вычислите значение хэш-значения сообщения.
* Вычислите *w* = mod (*p* -1).
* Вычислите *u*1 = (hash × *w*) mod (*p* -1) и *u*2 = (*r* × *w*) mod (*p* -1).
* Вычислите *v* = ( × mod *p*) mod *p*.
* Если *v* равно *r*, то подпись считается верной.

Ограничения:

Выбор простого числа *p* должен быть осуществлен с учетом безопасности и рекомендаций, например, использование достаточно больших простых чисел.

Размер числа *p* должен быть достаточно большим, чтобы обеспечить безопасность схемы.

Выбор случайных чисел *x* и *k* должен осуществляться с помощью надежного генератора случайных чисел.

Хэш-функция, используемая для преобразования сообщения в хэш-значение, должна быть стойкой к коллизиям и рекомендованной к использованию, например, SHA-256.

Важно сохранять закрытый ключ в тайне, так как он используется для создания подписи и его компрометация может привести к подделке подписей.

Весь код представлен на листинге 2.4.

static void Main(string[] args)

{

int p = 2137; // простое

int g = 2127; // первый корень < p

int x = 1116; // < p

int y = (int)BigInteger.ModPow(g, x, p);

int k = 7; // взаимно простое с p-1

int a = (int)BigInteger.ModPow(g, k, p);

Console.WriteLine($"p={p}\ng={g}\nx={x}\ny={y}\nk={k}\na={a}\n");

int H = 2119;

int m = p - 1;

int k\_1 = 0;

for (int i = 0; i < 10000; i++)

{

if (((k \* i) % m) == 1)

{

k\_1 = i;

break;

}

}

var b = (k\_1 \* (H - (x \* a) % m) % m) % m;

Console.WriteLine($"H={H}\nk\_1={k\_1}\nb={b}\nS = {a},{b}\n");

Console.WriteLine("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Верификация\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

var ya = BigInteger.ModPow(y, a, p);

var ab = BigInteger.ModPow(a, b, p);

var pr1 = BigInteger.ModPow(ya \* ab, 1, p);

var pr2 = BigInteger.ModPow(g, H, p);

Console.WriteLine("Проводим верфикацию");

if (pr1 == pr2)

{

Console.WriteLine($"{pr1} = {pr2}\nВерификация успешна");

}

else Console.WriteLine($"\nВерификация не пройдена");

Console.WriteLine("\nДля примера провалим: \n");

pr2 += pr2 + 1;

if (pr1 == pr2)

{

Console.WriteLine($"{pr1} = {pr2}\nВерификация успешна");

}

else Console.WriteLine($"{pr1} != {pr2}\nВерификация не пройдена");

Console.ReadKey();

}

Рисунок 2.4 – Методы для работы алгоритма Эль-Гамаля

Результат ЭЦП Эль-Гамаля представлен на рисунке 2.5.

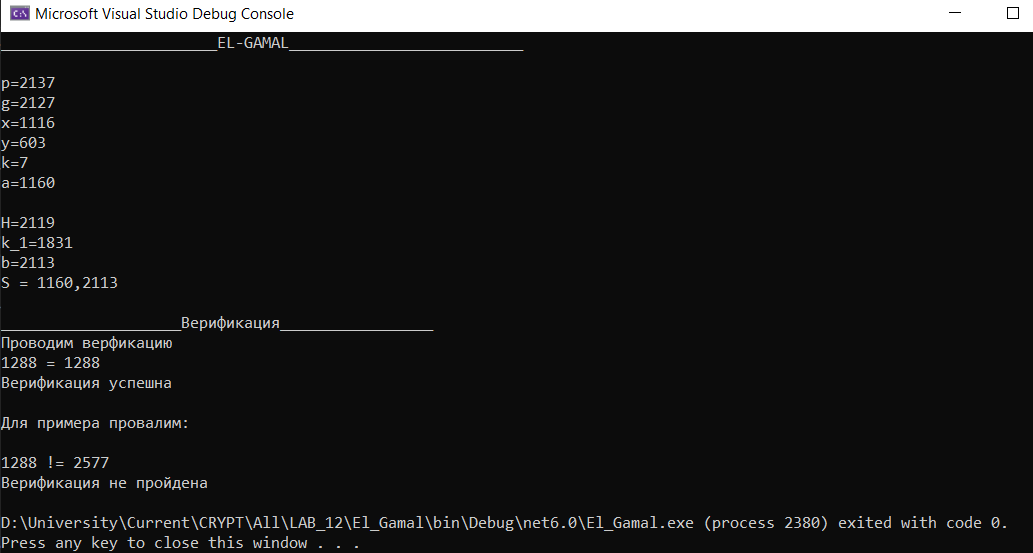


Рисунок 2.5 – Результат работы ЭЦП Эль-Гамаля

## 2.3. Алгоритм Шнорра

Алгоритм Шнорра для ЭЦП (Электронной Цифровой Подписи) является одним из популярных алгоритмов подписывания сообщений. Он включает следующие шаги:

1. Генерация ключей:

* Выберите простое число p большое и безопасное.
* Выберите целое число *q*, которое является простым делителем числа *p*-1.
* Выберите случайное целое число *g*, такое что mod *p* = 1.
* Выберите случайное целое число *x*, такое что 1 <= *x* < *q*.
* Вычислите *y* = mod *p*.
* Открытый ключ: (*p*, *q*, *g*, *y*)
* Закрытый ключ: *x*

1. Подписывание сообщения:

* Преобразуйте сообщение в хэш-значение с использованием хэш-функции, например, SHA-256.
* Выберите случайное целое число *k*, такое что 1 <= *k* < *q*.
* Вычислите *r* = ( mod *p*) mod *q*.
* Вычислите *e* = хэш-значение сообщения.
* Вычислите *s* = (*k* – *x* ×*r*) × mod *q*.
* Полученная подпись: (*r*, *s*)

1. Проверка подписи:

* Получите открытый ключ Шнорра (*p*, *q*, *g*, *y*) отправителя.
* Вычислите значение хэш-значения сообщения.
* Вычислите *v* = × mod *p* mod *q*.
* Если *v* равно *r*, то подпись считается верной.

Полный код реализации алгоритма Шнора представлен на листинге 2.6

static void Main()

{

Stopwatch stopwatch = Stopwatch.StartNew();

Console.WriteLine("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_SHNOR\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

BigInteger p = 2267;

BigInteger q = 103;

string text = File.ReadAllText(".\\original.txt");

BigInteger g = 354;

BigInteger m = 967;

int x = 30;

Console.WriteLine("p = "+ p + "\nq = " + q + "\ng = " + g + "\nm = " + m + "\nx = "+ x + "\n");

BigInteger y = BigInteger.ModPow(m, x, p);

BigInteger a = BigInteger.ModPow(g, 13, p);

using (var md5 = MD5.Create())

{

var hashBytes = md5.ComputeHash(System.Text.Encoding.ASCII.GetBytes(text + a));

BigInteger hash = new BigInteger(hashBytes);

hash = BigInteger.Abs(hash);

File.WriteAllText(".\\Hash.txt", hash.ToString());

Console.WriteLine("Записали хэш в Hash.txt");

BigInteger b = (13 + x \* hash) % q;

BigInteger dov = BigInteger.ModPow(g, b, p);

BigInteger X = (dov \* BigInteger.ModPow(y, hash, p)) % p;

var hashBytes2 = md5.ComputeHash(System.Text.Encoding.ASCII.GetBytes(text + X));

BigInteger hash2 = new BigInteger(hashBytes2);

hash2 = BigInteger.Abs(hash2);

bool f = hash == hash2;

Console.WriteLine("Исходный файл вернул: "+f);

string text2 = File.ReadAllText(".\\False.txt");

var hashBytes3 = md5.ComputeHash(System.Text.Encoding.ASCII.GetBytes(text2 + X));

BigInteger hash3 = new BigInteger(hashBytes3);

hash3 = BigInteger.Abs(hash3);

bool f2 = hash == hash3;

Console.WriteLine("Модифцированный файл вернул: " + f2);

}

stopwatch.Stop();

Console.WriteLine($"{stopwatch.ElapsedMilliseconds} ms");

Console.ReadLine();

Листинг 2.6 – Реализация алгоритма Шнора

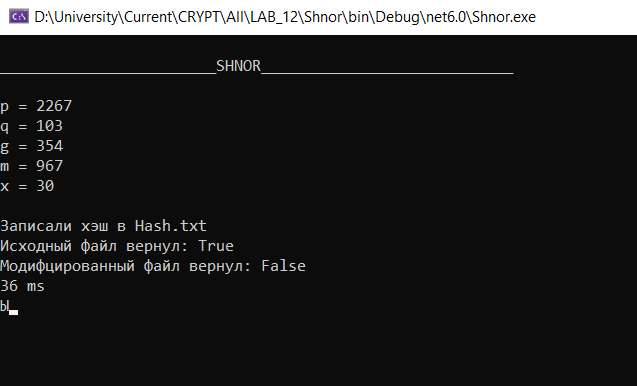


Рисунок 2.7 – Результат работы алгоритма Шнорра

Также на рисунке 2.8 представлен график оценки времени представленных процедур создания ЭЦП.

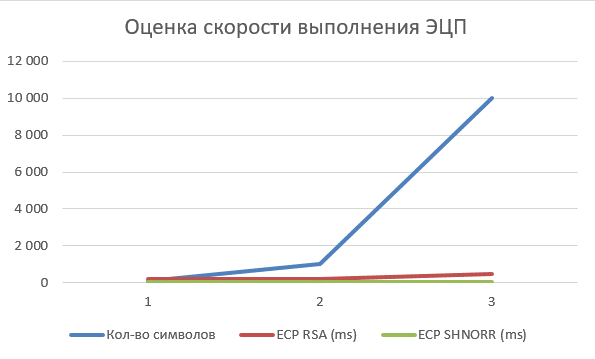


Рисунок 2.8 – Оценочный график выполнения всех процедур создания ЭЦП

## Вывод

В ходе лабораторной работы были изучены алгоритмы генерации и верификации электронной цифровой подписи, а также приобретены практические навыки их реализации. Приложение реализовывает генерацию и верификацию ЭЦП на основе алгоритмов RSA, Эль-Гамаля и Шнорра

Также была оценена скорость генерации и верификации ЭЦП.