Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ  
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информационных технологий

[Кафедра информационных](https://www.belstu.by/fakultety/fit/vm) систем и технологий

Специальность 1-40 01 01 Программное обеспечение информационных технологий

**Отчёт по лабораторной работе №12**

по дисциплине Информационная безопасность

Тема: Исследование алгоритмов генерации и верификации электронной цифровой подписи

Исполнитель:

Студент 3 курса группы 4

Станчик Максим Андреевич

Руководитель:

Ассистент Савельева М.Г.

Минск, 2025

**Лабораторная работа №12**

**Цель:** изучение алгоритмов генерации и верификации электронной цифровой подписи и приобретение практических навыков их реализации.

**Задачи:**

1. Закрепить теоретические знания по алгебраическому описанию и алгоритмам реализации операций генерации и верификации электронной цифровой подписи (ЭЦП).

2. Получить навыки практической реализации методов генерации и верификации ЭЦП на основе хеширования подписываемых сообщений и алгоритмов RSA, Эль-Гамаля и Шнорра, а также DSA.

3. Разработать приложение для реализации заданных алгоритмов генерации и верификации ЭЦП.

4. Оценить скорость генерации и верификации ЭЦП.

5. Результаты выполнения лабораторной работы оформить в виде описания разработанного приложения, методики выполнения экспериментов с использованием приложения и результатов эксперимента

# Теоретические сведения

Электронная цифровая подпись – контрольная характеристика сообщения, которая вырабатывается с использованием личного ключа, проверяется с использованием открытого ключа, служит для контроля целостности и подлинности сообщения и обеспечивает невозможность отказа от авторства.

Таким образом, ЭЦП выполняет те же функции, что и собственноручная (поставленная «от руки») подпись:

• аутентифицирование лица, подписавшего сообщение;

• контроль целостности подписанного сообщения;

• защита сообщения от подделок;

• доказательство авторства лица, подписавшего сообщение, если это лицо отрицает свое авторство.

Важнейшие отличительные особенности ЭЦП:

• ЭЦП представляет собой бинарную последовательность (в отличие от графического образа, каковым является подпись от руки);

• указанная бинарная последовательность зависит от содержания подписываемого сообщения.

Электронная цифровая подпись – бинарная (или в ином виде) последовательность символов, являющаяся реквизитом электронного документа, зависящая от содержания этого документа и предназначенная для подтверждения целостности и подлинности электронного документа.

ЭЦП на основе RSA. Здесь можно рассматривать две ситуации:

• сообщение *Мo* подписывается и передается в открытом (незашифрованном) виде;

• сообщение *Мo* подписывается и передается в зашифрованном виде.

При этом подпись S вычисляется на основе соотношения:

*S* ≡ *(H(Mo))dо* mod*no*.

Передаваемое сообщение *М'* = *Мo*||*S*. Соответственно, операция расшифрования на приемной стороне (получатель анализирует *Мп*||*S*) производится с известной модификацией ключей:

*H(Mo)* ≡ *(S) ео* mod*no*.

Далее вычисляется *Н(Mп)*. Если *H(Mo)* = *H(Mп)*, подпись верифицирована.

Если подписываемое сообщение *М(М')* также должно передаваться в зашифрованном виде, то обычно *М'* шифруется на стороне отправителя стандартным образом: с помощью открытого ключа получателя (*еп* и *nп*), который перед основным процессом верификации подписи расшифровывает послание своим тайным ключом: *dп* и *nп*. Далее осуществляются вычисления и анализ, как и в первом случае.

Основное отличие в применении расчетов в алгоритме Эль-Гамаля состоит в том, что результатом зашифрования является только одна пара чисел, а не пара для каждого блока исходного сообщения. Причем в рассматриваемом случае таким сообщением является хеш подписываемого документа: *Н(Mо)*.

Алгоритм ЭЦП К. Шнорра (К. Schnorr) является вариантом алгоритма ЭЦП Эль-Гамаля. Одной из особенностей ЭЦП Эль-Гамаля является то, что число *p* должно быть очень большим, чтобы сделать действительно трудной проблему дискретного логарифма. Рекомендуемая длина *p* должна составлять по крайней мере 1024 бита. Чтобы уменьшить размер подписи, Шнорр предложил новую схему, но с уменьшенным размером подписи.

Ключевая информация: *p* – простое число в диапазоне от 512 до 1024 битов; *q* –160-битное простое число, делитель (*p* – 1); любое число *g* (*g* ≠ 1) такое, что

*gq* ≡ 1mod *p*.

Числа *p*, *g*, *q* являются открытыми и могут применяться группой пользователей. Выбирается число *х* < *q* (*х* является тайным ключом) и вычисляется последний элемент открытого ключа:

*y* ≡ *g–х* mod *p*.

Секретный ключ имеет длину не менее 160 битов.

# Ход работы

Было необходимо разработать приложение, выполняющее генерацию и верификацию ЭЦП с помощью алгоритмов RSA, Эль-Гамаля и Шнорра.

Алгоритм RSA основан на сложности факторизации больших чисел. Основная идея заключается в том, что, имея *n* и *e*, можно зашифровать сообщение, но для расшифровки необходимо знать *d*, который зависит от секретных *p* и *q*. Поэтому были определены поля используются для хранения значений, необходимых для генерации ключей и выполнения операций шифрования и подписи. Они вычисляются в конструкторе класса RSA, что позволяет создать как публичный, так и закрытый ключ. Код полей приведен в листинге 2.1.

private BigInteger p;

private final BigInteger q;

private final BigInteger n;

private final BigInteger fi;

private final BigInteger e;

private final BigInteger d;

Листинг 2.1 – Код полей в алгоритме RSA

Метод CreateDigitalSignature реализует основную функцию алгоритма, создание цифровой подписи. Подпись позволяет проверить целостность и подлинность сообщения. Используя закрытый ключ *d*, метод создает подпись, которую можно проверить с использованием соответствующего открытого ключа *e*. Код метода приведен в листинге 2.2.

public BigInteger createDigitalSignature(String text) {

try {

MessageDigest digest = MessageDigest.getInstance("SHA-256");

byte[] hash = digest.digest(text.getBytes("UTF-8"));

BigInteger hashBigInt = new BigInteger(1, hash);

return hashBigInt.modPow(d, n);}

catch (NoSuchAlgorithmException e) {

e.printStackTrace();

return null;}

catch (java.io.UnsupportedEncodingException e) {

e.printStackTrace();

return null;}};

Листинг 2.2 – Код метода для создания цифровой подписи

Метод используется для получения публичного ключа. Публичный ключ используется для шифрования сообщений и проверки цифровых подписей. Он может быть распространен среди пользователей, которые хотят отправить зашифрованные сообщения владельцу закрытого ключа. Код метода приведен в листинге 2.3.

public BigInteger[] getPublicKey() { return new BigInteger[]{this.e, this.n}; };

Листинг 2.3 – Код метода для получения публичного ключа

Метод verifyDigitalSignature реализует основную функцию алгоритма RSA — проверку цифровой подписи. Он использует публичный ключ для проверки подлинности сообщения, обеспечивая целостность и достоверность данных. Метод принимает в качестве параметров сообщение, и цифровую подпись, которую нужно проверить. Код метода приведен в листинге 2.4.

public boolean verifyDigitalSignature(String text, BigInteger digitalSign) {

try {

BigInteger signBytes = digitalSign.modPow(e, n);

MessageDigest digest = MessageDigest.getInstance("SHA-256");

byte[] receivedHash = digest.digest(text.getBytes("UTF-8"));

BigInteger receivedHashBigInt = new BigInteger(1, receivedHash);

return receivedHashBigInt.equals(signBytes);

}

catch (NoSuchAlgorithmException e) {

e.printStackTrace();

return false;

}

catch (java.io.UnsupportedEncodingException e) {

e.printStackTrace();

return false;

}

};

Листинг 2.4 – Код метода для проверки цифровой подписи

Конструктор RSA инициализирует все компоненты, необходимые для работы алгоритма RSA, обеспечивая генерацию ключей и подготовку к шифрованию и подписыванию сообщений. Код конструктора приведен в листинге 2.5.

public RSA() {

this.p = generatePrimeNumber(512);

this.q = generatePrimeNumber(512);

this.n = this.p.multiply(this.q);

this.fi = this.p.subtract(BigInteger.ONE).multiply(this.q.subtract(BigInteger.ONE));

this.e = generateCoprimeNumber(this.fi);

this.d = e.modInverse(fi);};

Листинг 2.5 – Код конструктора для инициализации компонентов для работа алгоритма RSA

Метод verifyDigitalSignature принимает в качестве параметров сообщение, для которого проверяется подпись, и массив, содержащий подпись, состоящую из двух частей. Этот метод реализует основную функцию алгоритма ElGamal — проверку цифровой подписи. Он использует публичный ключ (*y*, *g*, *p*) для проверки подлинности сообщения, обеспечивая целостность и достоверность данных. Позволяет проверить, что сообщение было подписано соответствующим закрытым ключом. Метод verifyDigitalSignature класса ElGamal принимает на вход полученный от отправителя текст, электронную подпись (массив из двух параметров: *а* и *b*) и осуществляет верификацию подписи с помощью алгоритма Эль-Гамаля. Для верификации подписи вычисляется хеш полученного сообщения *Н(Мп)* = *h*. Далее нужно убедиться, что выполняется равенство *yaab* ≡ *gh*mod *p*. Код метода приведен в листинге 2.6.

public boolean verifyDigitalSignature(String message, BigInteger[] digitalSignature) {

try {

MessageDigest digest = MessageDigest.getInstance("SHA-256");

byte[] hash = digest.digest(message.getBytes("UTF-8"));

BigInteger hashBigInt = new BigInteger(1, hash);

BigInteger leftPart = g.modPow(hashBigInt, p);

BigInteger rightPart = y.modPow(digitalSignature[0], p)

.multiply(digitalSignature[0].modPow(digitalSignature[1], p)).mod(p);

return leftPart.equals(rightPart);

}

catch (NoSuchAlgorithmException e) {

e.printStackTrace();

return false;

}

catch (UnsupportedEncodingException e) {

throw new RuntimeException(e);

}

};

Листинг 2.6 – Код метода для проверки подлинности сообщения

Метод createDigitalSignature принимает строку, для которой создается цифровая подпись, и возвращает массив с двумя элементами, представляющими подпись. Он реализует основную функцию алгоритма ElGamal — создание цифровой подписи. Подпись, состоящая из двух частей, позволяет проверить подлинность сообщения. Подпись создается с использованием закрытого ключа (*x*), что обеспечивает безопасность. Метод createDigitalSignature класса ElGamal принимает на вход оригинальный текст и подписывает его с помощью алгоритма Эль-Гамаля. Подписью являются два числа *а* и *b* (*S* = {*a*, *b*}). Число а вычисляется по формуле: *a* ≡ *gk* mod *p*. При вычислении параметра b, решается уравнение *Н(Mо)* ≡ (*x*×*a* + *k×b*) mod (*p* – 1). Код метода представлен в листинге 2.7.

public BigInteger[] createDigitalSignature(String message) {

BigInteger[] digitalSignI = new BigInteger[2];

SecureRandom random = new SecureRandom();

try {

MessageDigest digest = MessageDigest.getInstance("SHA-256");

byte[] hashBytes = digest.digest(message.getBytes("UTF-8"));

BigInteger hash = new BigInteger(1, hashBytes);

do {

BigInteger k;

do {

k = new BigInteger(p.bitLength() - 1, random).mod(p.subtract(BigInteger.TWO)).add(BigInteger.TWO);

} while (!isCoprime(k, p.subtract(BigInteger.ONE)));

digitalSignI[0] = g.modPow(k, p);

BigInteger temp = hash.subtract(x.multiply(digitalSignI[0])).mod(p.subtract(BigInteger.ONE));

temp = temp.multiply(k.modInverse(p.subtract(BigInteger.ONE))).mod(p.subtract(BigInteger.ONE));

if (temp.compareTo(BigInteger.ZERO) < 0) {

temp = p.subtract(BigInteger.ONE).subtract(temp.abs());

}

digitalSignI[1] = temp;

} while (digitalSignI[1].equals(BigInteger.ZERO));

} catch (NoSuchAlgorithmException e) {

e.printStackTrace();

} catch (UnsupportedEncodingException e) {

throw new RuntimeException(e);

}

return digitalSignI;

}

Листинг 2.7 – Код метода для создания цифровой подписи

Метод getPublicKey предоставляет доступ к публичному ключу для использования в криптографических операциях. Используется для шифрования сообщений и проверки цифровых подписей. Код метода представлен в листинге 2.8.

public BigInteger[] getPublicKey() { return new BigInteger[]{this.p, this.g, this.y}; }

Листинг 2.8 – Код метода для предоставления доступа к публичному ключу

Поля *p*, *g*, *x* и *y*, а также конструктор класса ElGamal формируют основу для работы алгоритма. Они обеспечивают необходимые параметры для криптографических операций, позволяя пользователю безопасно шифровать сообщения и подписывать их. Код приведен в листинге 2.9.

private final BigInteger p;

private final BigInteger g;

private final BigInteger x;

private final BigInteger y;

public ElGamal() {

SecureRandom random = new SecureRandom();

this.p = generatePrimeNumber(512);

this.g = generateCoprimeNumber(this.p);

this.x = BigInteger.valueOf(2).add(new BigInteger(1, random).mod(this.p.subtract(BigInteger.ONE).subtract(BigInteger.TWO)));

this.y = this.g.modPow(this.x, this.p);

}

Листинг 2.9 – Код конструктора и полей класса ElGamal

Класс Schnorr реализует основные элементы алгоритма цифровой подписи Шнорра. Поля и конструктор обеспечивают генерацию необходимых ключей и параметров для безопасной работы с цифровыми подписями. Код представлен в листинге 2.10.

private final BigInteger p;

private final BigInteger q;

private final BigInteger g;

private BigInteger x;

private final BigInteger y;

public Schnorr() {

SecureRandom random = new SecureRandom();

this.p = BigInteger.valueOf(48731);

this.q = BigInteger.valueOf(443);

this.g = BigInteger.valueOf(11444);

do {

this.x = BigInteger.valueOf(random.nextInt(q.intValue() - 1) + 1);

} while (this.x.compareTo(BigInteger.ONE) < 0 || this.x.compareTo(this.q) >= 0);

this.y = this.g.modPow(this.x, this.p);

}

Листинг 2.10 – Код конструктора и полей класса Schnoor

Метод createDigitalSignature является важной частью механизма цифровой подписи в алгоритме Шнорра. Он позволяет пользователям подписывать сообщения, обеспечивая их целостность и аутентичность. Создает цифровую подпись. Подпись состоит из двух частей, которые позволяют подтвердить подлинность сообщения. Метод createDigitalSignature класса Schorr принимает на вход оригинальный текст и подписывает его с помощью алгоритма Шнорра. Ключевой информацией в алгоритме Шнорра являются числа *p*, *g*, *q*, *y*. *p* – простое число в диапазоне от 512 до 1024 битов; *q* –160-битное простое число, делитель (*p* – 1); любое число *g* (*g* ≠ 1) такое, что *gq* ≡ 1 mod p, число *y* ≡ *g*–*х* mod *p*. Для подписи сообщения Мо выбирается случайное число *k* (1 < *k* < *q*) и вычисляет параметр *а*: *а* ≡ *gk*mod*p*. Далее вычисляется хеш от канкатенации сообщения *Мо* и числа *а*: *h* = *H*(*Mo*||*a*). Далее вычисляется значение *b*: *b* ≡ (*k* + *x*\**h*) mod *q*. Получателю отправляются *М'* = *Мо*||*S*; *S* = {*h*, *b*}.

Код метода представлен в листинге 2.11.

public BigInteger[] createDigitalSignature(String message) {

SecureRandom random = new SecureRandom();

BigInteger k;

do {

k = BigInteger.valueOf(random.nextInt(q.intValue() - 2) + 2);

} while (k.compareTo(BigInteger.ONE) <= 0 || k.compareTo(q) >= 0);

BigInteger a = g.modPow(k, p);

message += a.toString();

try {

MessageDigest digest = MessageDigest.getInstance("SHA-256");

byte[] hash = digest.digest(message.getBytes("UTF-8"));

BigInteger hashBigInt = new BigInteger(1, hash);

BigInteger r = hashBigInt;

BigInteger s = k.add(x.multiply(hashBigInt)).mod(q);

return new BigInteger[]{r, s};

}

catch (NoSuchAlgorithmException | java.io.UnsupportedEncodingException e) {

e.printStackTrace();

return null;

}

}

Листинг 2.11 – Код метода createDigitalSignature

Метод getPublicKey() возвращает массив, содержащий четыре значения: *p*, *q*, *g* и *y*. Эти значения составляют публичный ключ в алгоритме Шнорра. Публичный ключ используется для проверки цифровых подписей и шифрования сообщений. Он может быть распространен среди пользователей, которые хотят отправить зашифрованные сообщения владельцу закрытого ключа. Код метода представлен в листинге 2.12.

public BigInteger[] getPublicKey() {

return new BigInteger[]{this.p, this.q, this.g, this.y};

}

Листинг 2.12 – Код метода для получения публичного ключа

Метод verifyDigitalSignature класса Schorr принимает на вход полученный от отправителя текст, электронную подпись и осуществляет верификацию подписи с помощью алгоритма Шнорра. Для проверки подписи вычисляется *Х* ≡ *gbyh* (mod *p*). Затем он проверяет выполнение равенства: *h* = *Н(Mп||Х)*. Подпись достоверна, если равенство выполняется. Код метода представлен в листинге 2.13.

public boolean verifyDigitalSignature(String message, BigInteger[] digitalSignature) {

try {

BigInteger r = digitalSignature[0];

BigInteger s = digitalSignature[1];

BigInteger a = g.modPow(s, p).multiply(y.modPow(q.subtract(r), p)).mod(p);

String combined = message + a.toString();

MessageDigest digest = MessageDigest.getInstance("SHA-256");

byte[] receivedHash = digest.digest(combined.getBytes("UTF-8"));

BigInteger receivedHashBigInt = new BigInteger(1, receivedHash);

return r.equals(receivedHashBigInt);

}

catch (NoSuchAlgorithmException | java.io.UnsupportedEncodingException e) {

e.printStackTrace();

return false;

}

}

Листинг 2.13 – Код метода для верификации подписи

Метод generatePrimeNumber генерирует случайное простое число. В качестве параметров принимает длину в битах для генерируемого числа. Это определяет размер числа. Метод возвращает простое число, которое имеет заданную длину в битах. Код метода представлен в листинге 2.14.

public static BigInteger generatePrimeNumber(int bitLength) {

BigInteger primeCandidate;

do {

primeCandidate = new BigInteger(bitLength, random);

} while (!primeCandidate.isProbablePrime(100));

return primeCandidate;

}

Листинг 2.14 – Код метода для генерации случайного простого числа

Метод generateCoprimeNumber генерирует случайное число, которое является взаимно простым с заданным числом *fi*. В качестве параметров принимает число, с которым нужно найти взаимно простое число. Метод возвращает случайное взаимно простое число с *fi*. Код метода представлен в листинге 2.15.

public static BigInteger generateCoprimeNumber(BigInteger fi) {

BigInteger min = fi.add(BigInteger.ONE);

BigInteger max = fi.multiply(BigInteger.TWO);

BigInteger coprime;

do {

coprime = new BigInteger(max.bitLength(), random).mod(max.subtract(min)).add(min);

} while (!isCoprime(fi, coprime));

return coprime;

}

Листинг 2.15 – Код метода для генерации случайного простого числа

Метод isCoprime проверяет, являются ли два числа взаимно простыми. В качестве параметров принимает первое число и второе число. Возвращает true, если числа *a* и *b* взаимно простые, и false в противном случае. Код метода представлен в листинге 2.16.

public static boolean isCoprime(BigInteger a, BigInteger b) {

return a.gcd(b).equals(BigInteger.ONE);

}

Листинг 2.16 – Код метода для проверки, являются ли два числа взаимно простыми

Результат работы приложения с исходным текстом «Maxim Stanchik Andreevich» представлен на рисунке 2.1.

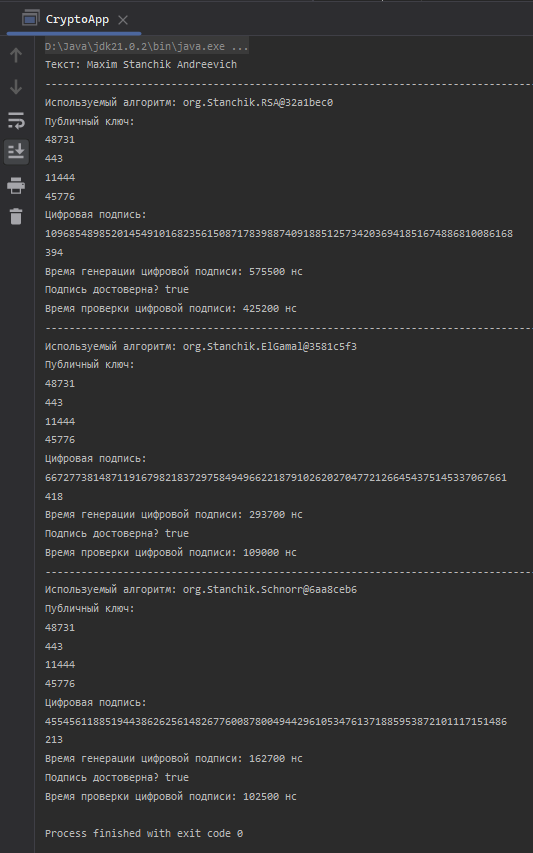


Рисунок 2.1 – Результат работы приложения

# Время генерации и верификации ЭЦП

Сравним время, за которое генерируется и верифицируется электронная цифровая подпись по алгоритмам RSA, Эль-Гамаля и Шнорра. Графики сравнения представлены на рисунках 3.1, 3.2, 3.2.

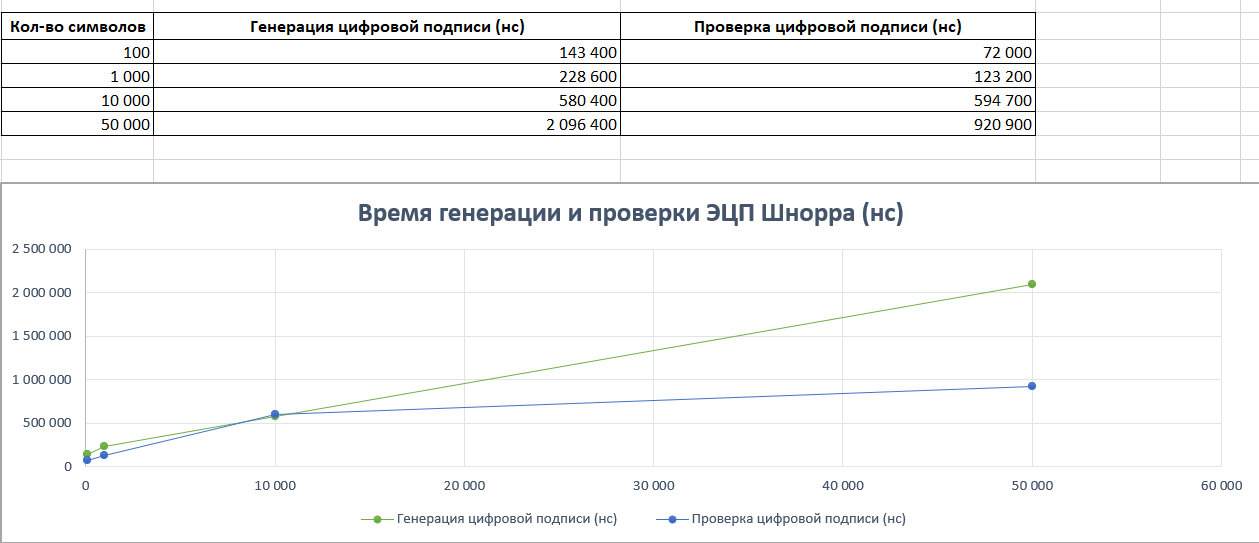


Рисунок 3.1 – График сравнения зависимости времени генерации и проверки ЭЦП Шнорра (нс) от количества символов в тексте

Генерация ЭЦП быстрее, чем у RSA и в некоторых случаях быстрее, чем у Эль-Гамаля. Проверка ЭЦП — самая быстрая из всех трёх алгоритмов. Хорошо масштабируется с увеличением длины сообщения. Алгоритм Шнорра наиболее эффективен по скорости проверки и в целом показал стабильные и низкие затраты времени при увеличении длины текста.

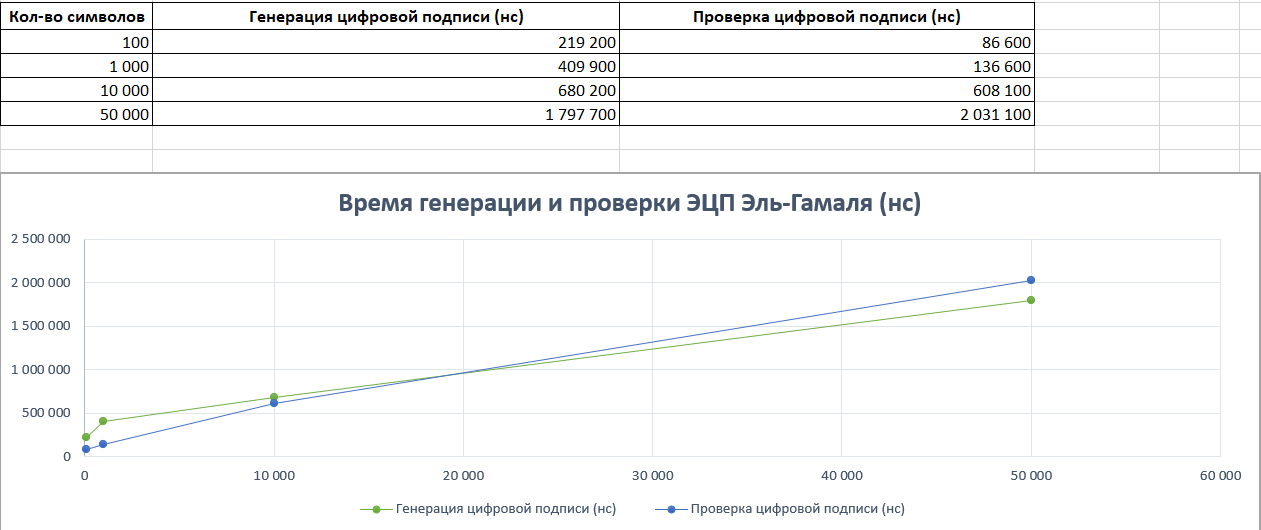


Рисунок 3.2 – График сравнения зависимости времени генерации и проверки ЭЦП Эль-Гамаля (нс) от количества символов в тексте

Генерация и проверка ЭЦП по времени примерно одинаковы.Работает быстрее RSA при больших объёмах данных.По сравнению со Шнорром — чуть медленнее при генерации, но быстрее RSA и в генерации, и в проверке.Вывод: Эль-Гамаль показывает хороший баланс между скоростью генерации и проверки ЭЦП, особенно на средних и больших объёмах текста.

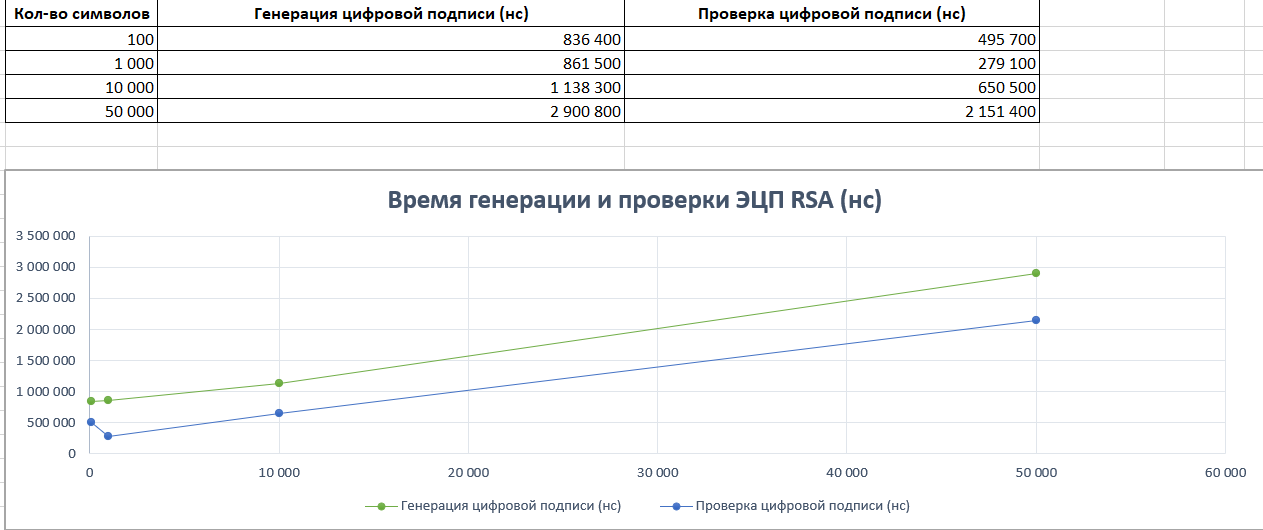


Рисунок 3.3 – График сравнения зависимости времени генерации и проверки ЭЦП RSA (нс) от количества символов в тексте

Самый медленный при генерации ЭЦП. Проверка быстрее генерации, но всё ещё уступает Шнорру и Эль-Гамалю.При увеличении текста время резко возрастает. RSA менее эффективен по времени по сравнению с другими алгоритмами и может быть не лучшим выбором при работе с большими объёмами данных, если ключевым фактором является производительность.

**Вывод:** в ходе выполнения лабораторной работы был изучен принцип генерации и верификации ЭЦП на основе алгоритмов RSA, Эль-Гамаля и Шнорра. Также было разработано приложение, выполняющее генерацию и верификацию ЭЦП на основе данных алгоритмов. Была оценена скорость генерации и верификации подписи. Наиболее производительным является алгоритм Шнорра, особенно если приоритет скорость проверки подписи. Эль-Гамаль является хороший компромисс между скоростью генерации и проверки. RSA стоит использовать, если важна совместимость или стойкость проверена временем, но он проигрывает по производительности.