ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Московский институт электроники и математики им. А.Н. Тихонова

Чудников Александр Александрович

Разбор экзаминационных вопросов

по дисциплине «Криптографические Методы Защиты Информации»

Студент

_____ Чудников А. А.
(подпись) *(И.О. Фамилия)*

Рецензент

_____ Чухно Р. Ш.
(подпись) *(И.О. Фамилия)*

Москва, 2025

Contents

1 Введение

Целью данной курсовой работы является исследование, практическая реализация и сравнительный анализ классических асимметричных криптосистем (RSA, Эль-Гамаля) и протокола обмена ключами Диффи-Хеллмана на языке C++.

Курсовая работа разделена на 3 части, каждая из которых включает в себя работу с каждым из вышеописанных протоколов:

- **Теоретическая часть**: Описание математических основ, лежащих в основе рассматриваемых криптосистем.
- **Практическая часть**: Программная реализация алгоритмов на языке C++.
- **Анализ проделанной работы**: Тестирование и оценка реализованных алгоритмов.

Программная реализация протоколов на языке C++ располагается на Github-е по ссылке:

2 Теория

2.1 Схема шифрования RSA

RSA (Rivest–Shamir–Adleman) — это асимметричная криптографическая система, основанная на вычислительной сложности задачи факторизации больших целых чисел. Предложенная в 1977 году Роном Ривестом, Ади Шамиром и Леонардом Адлеманом, она стала одной из первых практически применимых систем с открытым ключом и на сегодняшний день является мировым стандартом. Интересно, что аналогичный алгоритм был разработан ранее, в 1973 году, британским инженером Клиффордом Коксом (Clifford Cocks), но его работа была засекречена.

Принцип работы RSA заключается в использовании двух ключей: открытого, который можно свободно распространять, и закрытого (секретного), который должен храниться в тайне.

2.1.1 Алгоритм генерации ключевой пары

Для создания открытого и закрытого ключей выполняются следующие шаги:

1. **Выбор простых чисел.** Выбираются два различных очень больших простых числа p и q.
2. **Вычисление модуля.** Вычисляется их произведение, которое называется модулем:

$$m = p \cdot q.$$

3. **Вычисление функции Эйлера.** Вычисляется значение функции Эйлера от числа m:

$$\varphi(m) = (p-1)(q-1).$$

   Именно это значение, а не сам модуль m, определяет криптографическую стойкость ключа.

4. **Выбор открытой экспоненты.** Выбирается целое число e (открытая экспонента), удовлетворяющее двум условиям:
$$1 < e < \varphi(m) \quad \text{и} \quad \text{НОД}(e, \varphi(m)) = 1.$$
Часто в качестве e выбирают небольшие простые числа, например 65537 ($2^{16}+1$), для ускорения процесса шифрования.

5. **Вычисление секретной экспоненты.** Вычисляется число d (секретная экспонента), которое является мультипликативно обратным к e по модулю $\varphi(m)$:
$$ed \equiv 1 \pmod{\varphi(m)}.$$
Число d находится с помощью расширенного алгоритма Евклида.

6. **Формирование ключей.**
   - **Открытый ключ** — это пара чисел (m, e).
   - **Закрытый ключ** — это пара чисел (m, d).

   Простые числа p и q после генерации ключей должны быть уничтожены.

2.1.2 Алгоритм шифрования

Чтобы зашифровать сообщение s, представленное в виде целого числа ($1 < s < m$), отправитель (абонент А) должен выполнить следующую операцию, используя открытый ключ получателя (абонента Б):
$$c \equiv s^e \pmod{m}.$$
Полученное число c является шифртекстом и передаётся по открытому каналу связи.

2.1.3 Алгоритм расшифрования

Для расшифрования шифртекста c получатель (абонент Б) использует свой секретный ключ (m, d):
$$s \equiv c^d \pmod{m}.$$
Корректность расшифрования следует из теоремы Эйлера. Поскольку $ed \equiv 1 \pmod{\varphi(m)}$, то $ed = k\varphi(m) + 1$ для некоторого целого k. Тогда:
$$c^d \equiv (s^e)^d = s^{ed} = s^{k\varphi(m)+1} \equiv (s^{\varphi(m)})^k \cdot s^1 \equiv 1^k \cdot s \equiv s \pmod{m}.$$
Таким образом, получатель восстанавливает исходное сообщение s.

2.1.4 Криптографическая стойкость

Стойкость криптосистемы RSA основывается на вычислительной сложности задачи факторизации больших целых чисел. Зная только открытый ключ (m, e), злоумышленник для нахождения секретного ключа d должен сначала вычислить значение $\varphi(m) = (p-1)(q-1)$. Для этого ему необходимо разложить модуль m на простые сомножители p и q.

Более того, можно доказать, что знание секретного ключа d эквивалентно возможности факторизовать модуль m. Это основано на вероятностном алгоритме, который использует тот факт, что $ed - 1$ является кратным $\varphi(m)$. Представим $ed - 1$ в виде $2^n \cdot t$, где t — нечётное число. Для случайным образом выбранного числа a ($1 < a < m$) рассматривается последовательность $a^t, a^{2t}, \ldots, a^{2^n t} \pmod{m}$. Последний элемент этой последовательности равен 1.

Если в этой последовательности удаётся найти элемент v, такой что $v \not\equiv \pm 1 \pmod{m}$, но $v^2 \equiv 1 \pmod{m}$, то найден нетривиальный квадратный корень из единицы. В этом случае делители p и q могут быть найдены как НОД$(v-1, m)$ и НОД$(v+1, m)$. Доказывается, что для случайно выбранного a вероятность успеха такой атаки составляет не менее $\frac{1}{2}$. Несколько попыток с разными значениями a почти гарантированно приводят к факторизации модуля.

Таким образом, безопасность RSA неразрывно связана со сложностью задачи факторизации, и раскрытие секретного ключа d полностью компрометирует систему.

2.2 Схема шифрования Эль-Гамаля

Схема шифрования Эль-Гамаля, предложенная Тахиром Эль-Гамалем в 1984 году, является асимметричной криптосистемой, основанной на сложности решения задачи дискретного логарифмирования в конечных полях. В отличие от RSA, это вероятностный алгоритм: одно и то же открытое сообщение может быть зашифровано в множество различных шифртекстов, что повышает его стойкость к криптоанализу.

Рассмотрим реализацию схемы в мультипликативной группе конечного поля $\mathbb{F}_p^*$.

2.2.1 Генерация публичных параметров

Перед генерацией ключей для пользователей необходимо создать общий набор параметров, которые будут известны всем участникам системы:

1. Выбирается большое простое число p.
2. Выбирается целое число g (называемое генератором или порождающим элементом), которое является порождающим элементом циклической подгруппы порядка q в группе $\mathbb{F}_p^*$.

Параметры (p, q, g) являются открытыми и могут использоваться всеми участниками.

2.2.2 Алгоритм генерации ключевой пары

Каждый пользователь (получатель сообщения) генерирует свою пару ключей:

1. Выбирается случайное целое число d (секретный ключ), удовлетворяющее условию $1 < d < q$.
2. Вычисляется открытый ключ e по формуле:

$$e \equiv g^d \pmod{p}.$$

Таким образом:

- **Открытый ключ** — это кортеж (e, p, q, g).
- **Закрытый ключ** — это число d.

2.2.3 Алгоритм шифрования

Для шифрования сообщения s (представленного в виде числа, $1 < s < p$), отправитель использует открытый ключ получателя:

1. Выбирается случайное сессионное (эфемерное) число k, такое что $1 < k < q$.
2. Вычисляется первая компонента шифртекста:
$$r \equiv g^k \pmod{p}.$$
3. Вычисляется вторая компонента шифртекста:
$$c \equiv s \cdot e^k \pmod{p}.$$

Шифртекстом является пара чисел (r, c).

2.2.4 Алгоритм расшифрования

Получив шифртекст (r, c), получатель использует свой секретный ключ d для восстановления исходного сообщения s:
$$s \equiv c \cdot r^{-d} \pmod{p}.$$
Здесь r^{-d} — это мультипликативное обратное к r^d по модулю p.

Корректность расшифрования обеспечивается следующими преобразованиями:
$$c \cdot r^{-d} \equiv (s \cdot e^k) \cdot (g^k)^{-d} \equiv s \cdot (g^d)^k \cdot g^{-kd} \equiv s \cdot g^{dk} \cdot g^{-dk} \equiv s \cdot g^0 \equiv s \pmod{p}.$$

2.2.5 Криптостойкость и особенности

Безопасность схемы Эль-Гамаля целиком основывается на вычислительной сложности **задачи дискретного логарифмирования (DLP)**. Зная открытые параметры (p, q, g) и открытый ключ e, злоумышленник для нахождения секретного ключа d должен решить уравнение $e \equiv g^d \pmod{p}$. Для достаточно больших p и q эта задача считается неразрешимой за приемлемое время.

Схема имеет две важные особенности:

- **Вероятностный характер.** Из-за использования случайного числа k при каждом шифровании, одно и то же сообщение s преобразуется в разные шифртексты. Это свойство защищает от атак, основанных на частотном анализе.
- **Удвоение размера.** Шифртекст (r, c) состоит из двух чисел, поэтому его размер вдвое превышает размер исходного сообщения s. Это является основным недостатком схемы, ограничивающим её применение для шифрования больших объёмов данных.

2.3 Протокол обмена ключами Диффи–Хеллмана

Протокол Диффи–Хеллмана (Diffie–Hellman, DH), разработанный Уитфилдом Диффи и Мартином Хеллманом в 1976 году, стал первым в истории практическим методом для установления общего секретного ключа по незащищённому каналу связи. В отличие от RSA и Эль-Гамаля, DH не является схемой шифрования, а предназначен исключительно для выработки общего секрета, который затем может быть использован в симметричных алгоритмах шифрования.

2.3.1 Генерация публичных параметров

Как и в схеме Эль-Гамаля, участники должны предварительно согласовать общие параметры:

1. Большое простое число p.
2. Порождающий элемент (генератор) g циклической группы по модулю p.

Эти параметры (p, g) являются открытыми и не требуют защиты.

2.3.2 Алгоритм выработки общего секрета

Процесс выработки ключа между двумя участниками, А и Б, происходит следующим образом:

1. **Сторона А** выбирает случайное секретное число x (где $1 < x < p - 1$), вычисляет $K_A = g^x \pmod p$ и отправляет результат K_A стороне Б.
2. **Сторона Б** выбирает своё случайное секретное число y (где $1 < y < p - 1$), вычисляет $K_B = g^y \pmod p$ и отправляет результат K_B стороне А.
3. **Сторона А**, получив K_B, вычисляет общий секретный ключ k по формуле:
$$k \equiv (K_B)^x \equiv (g^y)^x \equiv g^{yx} \pmod p.$$
4. **Сторона Б**, получив K_A, вычисляет тот же самый ключ:
$$k \equiv (K_A)^y \equiv (g^x)^y \equiv g^{xy} \pmod p.$$

В результате оба участника получают одинаковый секретный ключ k, не передавая в открытом виде свои секретные числа x и y.

2.3.3 Криптостойкость

Безопасность протокола Диффи–Хеллмана основана на сложности решения **задачи дискретного логарифмирования (DLP)**. Злоумышленник, перехватывающий сообщения в канале, имеет доступ к числам $p, g, K_A = g^x$ и $K_B = g^y$. Чтобы вычислить общий секрет $k = g^{xy}$, ему необходимо найти либо секретное число x из K_A, либо y из K_B. Эта задача, как и в случае с Эль-Гамалем, считается вычислительно неразрешимой для достаточно больших p.

2.3.4 Уязвимость: атака «человек посередине» (Man-in-the-Middle)

Базовый протокол Диффи–Хеллмана уязвим к атаке «человек посередине» (Man-in-the-Middle, MitM), поскольку он не аутентифицирует участников обмена. Злоумышленник С может вклиниться в канал связи между А и Б.

Атака проходит следующим образом:

1. Участник А отправляет $g^x \pmod p$ в сторону Б. Злоумышленник С перехватывает это сообщение.
2. Злоумышленник С генерирует собственное секретное число z_1, вычисляет $g^{z_1} \pmod p$ и отправляет его участнику Б, выдавая себя за А.

3. Участник Б отправляет $g^y \pmod p$ в сторону А. С также перехватывает это сообщение.
4. Злоумышленник С генерирует второе секретное число z_2, вычисляет $g^{z_2} \pmod p$ и отправляет его участнику А, выдавая себя за Б.

В результате складывается следующая ситуация:

- Участник А устанавливает общий ключ со злоумышленником С: $k_{AC} = (g^{z_2})^x = g^{xz_2} \pmod p$.
- Участник Б также устанавливает общий ключ со злоумышленником С: $k_{BC} = (g^{z_1})^y = g^{yz_1} \pmod p$.

При этом А и Б не догадываются, что выработали ключи не друг с другом, а с нарушителем. Злоумышленник С, зная оба ключа, может перехватывать, читать и изменять все сообщения, которыми обмениваются А и Б, сохраняя при этом иллюзию защищённого канала. Для защиты от этой атаки требуются дополнительные механизмы аутентификации, например, цифровые подписи.

3 Реализация

3.1 Используемые библиотеки

В процессе реализации были использованы следующий сторонние библиотеки:

- **Qt (6.9.1)**: Использовался чисто в качестве графической обертки для более интерактивного взаимодействия с протоколами.
- **Boost (multiprecision, random)**: Для работы с большими числами и генерацией псевдо случайных рандомных чисел.
- **spdlog**: Для более удобного логирования событий и упрощения отладки.
- **fmt**: В качестве внешней зависимости spdlog.

3.2 Используемые средства сборки

В качестве компилятора был выбран **clang 20.1.6** в связи с тем что сборка при помощи последнего быстрее на 5-10 процентов.

В качестве системы сборки был выбран **CMake 4.0.3**.

3.3 Программная реализация RSA

3.3.1 Структура класса и зависимости

Реализация инкапсулирована в класс ‘RSA‘, который наследуется от абстрактного базового класса ‘Protocol‘. Это позволяет в будущем легко интегрировать другие криптографические протоколы в единую систему. В заголовочном файле ‘rsa.hpp‘ определена структура класса и его основные компоненты.

nvim

```
#ifndef RSA_HPP
#define RSA_HPP

#include "protocol.hpp"

#include <boost/multiprecision/cpp_int.hpp>
#include <boost/random/mersenne_twister.hpp>

using BigInt = boost::multiprecision::cpp_int;

namespace CRYPTO
{
class RSA final : public Protocol
{
public:
  explicit RSA(unsigned int key_bits = 2048);

  void init() override;

  QString encrypt(const QString& plaintext) override;
  QString decrypt(const QString& ciphertext) override;

private:
  void generateKeys();
  BigInt generatePrime(unsigned int bits, boost::random::mt19937& rng);

  // Параметры RSA
  unsigned int m_key_bits;
  BigInt     m_m; // Модуль m = p * q (часть открытого ключа)
  BigInt     m_e; // Открытая экспонента e (часть открытого ключа)
  BigInt     m_d; // Закрытая экспонента d (закрытый ключ)

  // Простые множители сохраняются для потенциального использования, но
  ↪  являются частью закрытого ключа
  BigInt m_p, m_q;
  BigInt m_phi; // Функция Эйлера: phi(n) = (p-1)*(q-1)
};
} // namespace CRYPTO

#endif // RSA_HPP
```

3.3.2 Генерация ключей

Процесс генерации ключевой пары является наиболее важным и сложным этапом. Он реализован в методе ‘generateKeys()‘ и соответствует классическому алгоритму.

1. **Генерация простых чисел p и q.** Для этого используется вспомогательный метод ‘generatePrime()‘, который генерирует случайное число заданной битовой длины и проверяет его на простоту с помощью вероятностного теста Миллера–Рабина. В реализации используется 25 итераций теста, что обеспечивает чрезвычайно высокую вероятность того, что сгенерированное

число действительно является простым. Генерация продолжается до тех пор, пока не будут найдены два различных простых числа.

2. **Вычисление модуля m и функции Эйлера $\varphi(m)$.** Модуль вычисляется как $m = p \cdot q$, а значение функции Эйлера — как $\varphi(m) = (p-1)(q-1)$.

3. **Выбор открытой экспоненты e.** В качестве открытой экспоненты выбрано стандартное значение $e = 65537$. Это число является простым и имеет всего два единичных бита в двоичном представлении ($2^{16}+1$), что позволяет значительно ускорить операцию возведения в степень при шифровании. Производится проверка, что НОД$(e, \varphi(m)) = 1$.

4. **Вычисление секретной экспоненты d.** Секретная экспонента d вычисляется как мультипликативное обратное к e по модулю $\varphi(m)$, то есть $d \equiv e^{-1} \pmod{\varphi(m)}$. Для этого используется расширенный алгоритм Евклида, реализованный в отдельной функции ‘inverse()‘.

nvim

```
BigInt RSA::generatePrime(unsigned int bits, boost::random::mt19937& rng)
{
  // Задаем диапазон для генерации числа с нужным количеством бит
  BigInt                      lower_bound = BigInt(1) << (bits - 1);
  BigInt                      upper_bound = (BigInt(1) << bits) - 1;
  boost::random::uniform_int_distribution<BigInt> dist(lower_bound,
  ↪   upper_bound);

  BigInt candidate;
  while (true)
  {
    candidate = dist(rng);
    // Убедимся, что число нечетное
    if (candidate % 2 == 0)
    {
      candidate++;
    }
    // Проверяем на простоту с помощью теста Миллера-Рабина
    // 25 итераций дают очень высокую вероятность того, что число простое
    if (boost::multiprecision::miller_rabin_test(candidate, 25))
    {
      return candidate;
    }
  }
}
```

nvim

```
void RSA::generateKeys()
{
    // 1. Инициализация генератора случайных чисел
    boost::random::mt19937
    ↪   rng(std::chrono::high_resolution_clock::now().time_since_epoch().count());

    // 2. Генерация двух различных простых чисел p и q
    unsigned int prime_bits = m_key_bits / 2;
    do
    {
        m_p = generatePrime(prime_bits, rng);
        m_q = generatePrime(prime_bits, rng);
    } while (m_p == m_q);

    // 3. Вычисление модуля m и функции Эйлера phi(m)
    m_m   = m_p * m_q;
    m_phi = (m_p - 1) * (m_q - 1);

    // 4. Выбор открытой экспоненты e
    m_e = 65537;
    if (boost::multiprecision::gcd(m_e, m_phi) != 1)
    {
        // В маловероятном случае, если 65537 не подходит,
        // генерируем ключи заново
        generateKeys();
        return;
    }

    // 5. Вычисление закрытой экспоненты d
    m_d = inverse(m_e, m_phi);
}
```

3.3.3 Шифрование и расшифрование

Процессы шифрования и расшифрования требуют преобразования текстовых данных в числовое представление и обратно.

Шифрование

Метод ‘encrypt()‘ выполняет следующие шаги:

1. Входная строка ‘QString‘ преобразуется в массив байтов ‘QByteArray‘ в кодировке UTF-8.
2. Байтовый массив представляется в виде строки шестнадцатеричных символов. Это необходимо для однозначного преобразования двоичных данных в число.
3. Шестнадцатеричная строка преобразуется в большое целое число ‘BigInt‘.
4. Производится проверка, что полученное число меньше модуля m.
5. Выполняется основная операция шифрования: $c = s^e \pmod m$ с помощью функции ‘boost::multiprecision::powm‘, оптимизированной для модульного возведения в степень.

6. Полученный шифртекст c преобразуется обратно в шестнадцатеричную строку для передачи.

nvim

```
QString RSA::encrypt(const QString& plaintext)
{
    // 1. Преобразуем открытый текст в байты
    QByteArray bytes = plaintext.toUtf8();

    // 2. Преобразуем байты в шестнадцатеричную строку.
    QString hex_plaintext = bytes.toHex();

    // 3. Преобразуем шестнадцатеричную строку в BigInt.
    BigInt message("0x" + hex_plaintext.toStdString());

    // 4. Проверяем, что сообщение меньше модуля n
    if (message >= m_m)
    {
        throw std::runtime_error("Message is too large for the current key
        ↪  size.");
    }

    // 5. Шифруем: c = m^e mod n
    BigInt ciphertext = boost::multiprecision::powm(message, m_e, m_m);

    // 6. Преобразуем зашифрованное число в строку в шестнадцатеричном формате
    return QString::fromStdString(ciphertext.str(0, std::ios_base::hex));
}
```

Расшифрование

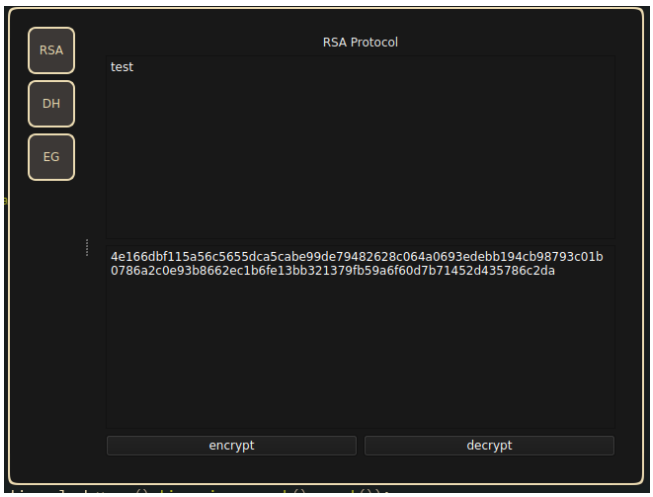
Метод ‘decrypt()‘ выполняет обратную последовательность действий:

1. Входной шифртекст в виде шестнадцатеричной строки преобразуется в большое целое число ‘BigInt‘.
2. Выполняется операция расшифрования: $s = c^d \pmod{m}$ с помощью функции ‘powm‘.
3. Расшифрованное число преобразуется обратно в шестнадцатеричную строку.
4. На этом этапе выполняется важная коррекция: если полученная строка имеет нечетную длину, в начало добавляется ведущий ноль. Это необходимо, так как при преобразовании числа в строку ‘0x0F‘ может превратиться в ‘”f”‘ вместо ‘”0f”‘, что приведет к ошибке на следующем шаге.
5. Скорректированная шестнадцатеричная строка преобразуется в ‘QByteArray‘, а затем в ‘QString‘ в кодировке UTF-8, восстанавливая исходное сообщение.

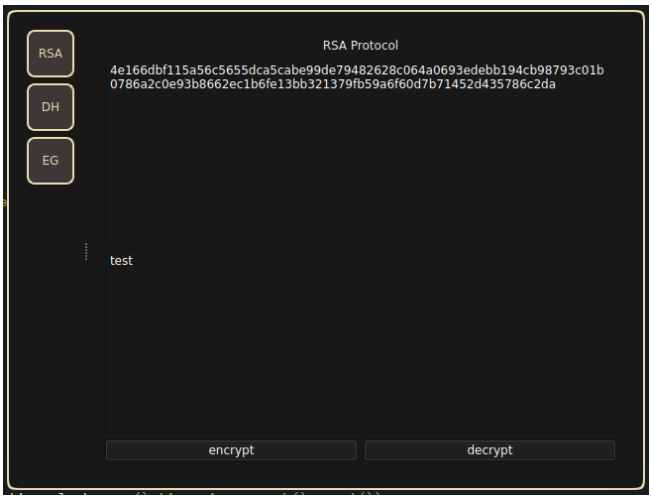
nvim

```
QString RSA::decrypt(const QString& ciphertext)
{
    // 1. Преобразуем шифротекст в BigInt
    BigInt encrypted_message("0x" + ciphertext.toStdString());

    // 2. Расшифровываем: m = c^d mod n
    BigInt decrypted_message = boost::multiprecision::powm(encrypted_message,
    ↪  m_d, m_m);

    // 3. Преобразуем расшифрованное число обратно в шестнадцатеричную строку
    std::string hex_str = decrypted_message.str(0, std::ios_base::hex);

    // 4. Восстанавливаем возможный утерянный ведущий ноль
    if (hex_str.length() % 2 != 0)
    {
        hex_str.insert(0, "0");
    }

    // 5. Преобразуем шестнадцатеричную строку обратно в QByteArray
    QByteArray bytes =
    ↪  QByteArray::fromHex(QByteArray::fromStdString(hex_str));

    // 6. Создаем QString из байтов в кодировке UTF-8
    return QString::fromUtf8(bytes);
}
```

3.4 Пример работы RSA



(a) Encryption



(b) Decryption

3.5 Программная реализация схемы Эль-Гамаля

3.5.1 Структура класса и зависимости

Реализация инкапсулирована в класс ‘ElGamal‘, который, как и ‘RSA‘, наследуется от абстрактного базового класса ‘Protocol‘. Это обеспечивает унифицированный интерфейс для работы с различными криптосистемами. Структура класса и его ключевые параметры определены в заголовочном файле ‘elgamal.hpp‘.

nvim

```cpp
#ifndef ELGAMAL_HPP
#define ELGAMAL_HPP

#include "protocol.hpp"

#include <boost/multiprecision/cpp_int.hpp>
#include <boost/random/mersenne_twister.hpp>

using BigInt = boost::multiprecision::cpp_int;

namespace CRYPTO
{
class ElGamal final : public Protocol
{
public:
  explicit ElGamal(unsigned int key_bits = 2048);

  void init() override;

  QString encrypt(const QString& plaintext) override;
  QString decrypt(const QString& ciphertext) override;

private:
  void generateParameters();
  void generateKeys();
  BigInt generatePrime(unsigned int bits, boost::random::mt19937& rng);

  unsigned int m_key_bits; // Длина ключа в битах

  // Публичные параметры, общие для всех
  BigInt m_p; // Большое простое число (модуль)
  BigInt m_g; // Порождающий элемент (генератор)

  // Ключи
  BigInt m_d; // Закрытый ключ (секретное число, 1 < d < p-1)
  BigInt m_e; // Открытый ключ (e = g^d mod p)
};

} // namespace CRYPTO

#endif // ELGAMAL_HPP
```

3.5.2 Генерация параметров и ключей

В отличие от RSA, в схеме Эль-Гамаля процесс инициализации разделен на два логических этапа: генерация общих криптографических параметров и создание ключевой пары на их основе. Эти этапы выполняются в методе ‘init()‘, который вызывает ‘generateParameters()‘ и ‘generateKeys()‘.

1. **Генерация простого модуля p и генератора g.** Сначала генерируется большое простое число p заданной битовой длины с помощью метода ‘generatePrime()‘, использующего тест Миллера–Рабина (25 итераций) для проверки простоты. Затем выбирается генератор g циклической группы по модулю p. Для упрощения реализации в качестве g выбрано значение 2. В криптографически стойких системах выбор генератора является более сложной задачей, но для демонстрационных целей это приемлемо.

2. **Выбор секретного ключа d.** Секретный ключ d выбирается как случайное целое число из диапазона $1 < d < p - 1$.

3. **Вычисление открытого ключа e.** Открытый ключ e вычисляется на основе g и d по формуле $e = g^d \pmod{p}$. Для этого используется функция модульного возведения в степень ‘boost::multiprecision::powm‘.

nvim

```
void ElGamal::generateParameters()
{
  boost::random::mt19937
    ↪  rng(std::chrono::high_resolution_clock::now().time_since_epoch().count());

  // 1. Генерируем большое простое число p нужной битовой длины.
  m_p = generatePrime(m_key_bits, rng);

  // 2. Выбираем генератор g.
  // Для простоты реализации часто выбирают небольшое число.
  m_g = 2;
}
```

nvim

```
void ElGamal::generateKeys()
{
  boost::random::mt19937
    ↪  rng(std::chrono::high_resolution_clock::now().time_since_epoch().count());

  // 1. Выбираем случайный секретный ключ d
  // в диапазоне 1 < d < p-1. Используем p-1, так как размер подгруппы
  // может быть p-1 (если p - безопасное простое).
  boost::random::uniform_int_distribution<BigInt> dist(2, m_p - 2);
  m_d = dist(rng);

  // 2. Вычисляем открытый ключ e: e = g^d mod p.
  m_e = boost::multiprecision::powm(m_g, m_d, m_p);
}
```

3.5.3 Шифрование и расшифрование

Особенностью схемы Эль-Гамаля является то, что шифртекст состоит из двух частей, а сам процесс шифрования является вероятностным.

Шифрование

Метод ‘encrypt()‘ выполняет следующие шаги:

1. Входная строка ‘QString‘ преобразуется в большое целое число s через промежуточное представление в виде шестнадцатеричной строки, аналогично реализации в RSA.
2. Проверяется, что полученное число s меньше модуля p.
3. **Генерация сессионного ключа k.** Для каждого акта шифрования генерируется новое случайное число k (эфермерный ключ) в диапазоне $1 < k < p - 1$. Использование уникального k для каждого сообщения обеспечивает вероятностный характер шифрования.
4. **Вычисление первой компоненты шифртекста r.** Первая часть вычисляется как $r = g^k \pmod{p}$.
5. **Вычисление второй компоненты шифртекста c.** Вторая часть вычисляется по формуле $c = s \cdot e^k \pmod{p}$.
6. Компоненты r и c преобразуются в шестнадцатеричные строки и объединяются через пробел для формирования итогового шифртекста.

nvim

```
QString ElGamal::encrypt(const QString& plaintext)
{
  if (m_p == 0 || m_g == 0 || m_e == 0)
  {
    throw std::runtime_error("ElGamal is not initialized. Call init() before
    ↪  encryption.");
  }

  // 1. Преобразуем открытый текст в BigInt s (через hex, как в RSA).
  QByteArray bytes     = plaintext.toUtf8();
  QString    hex_plaintext = bytes.toHex();
  BigInt     s("0x" + hex_plaintext.toStdString());

  if (s >= m_p)
  {
    throw std::runtime_error("Message is too large for the current key
    ↪  size.");
  }
```

```
nvim
  // 2. Выбираем случайное сессионное (эфемерное) число k: 1 < k < p-1.
  boost::random::mt19937                    rng(std::chrono::high_resolution_clock::now().time_si
  boost::random::uniform_int_distribution<BigInt> dist(2, m_p - 2);
  BigInt                          k = dist(rng);

  // 3. Вычисляем первую компоненту шифртекста: r = g^k mod p.
  BigInt r = boost::multiprecision::powm(m_g, k, m_p);

  // 4. Вычисляем вторую компоненту шифртекста: c = s * e^k mod p.
  BigInt e_k = boost::multiprecision::powm(m_e, k, m_p);
  BigInt c   = (s * e_k) % m_p;

  // 5. Формируем шифртекст: пара (r, c) в виде hex-строк, разделенных
  ↪  пробелом.
  QString r_hex = QString::fromStdString(r.str(0, std::ios_base::hex));
  QString c_hex = QString::fromStdString(c.str(0, std::ios_base::hex));

  return r_hex + " " + c_hex;
}
```

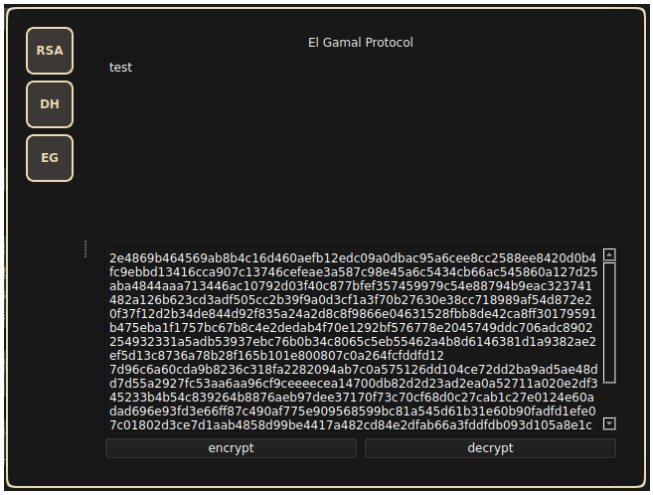
Расшифрование

Метод ‘decrypt()‘ выполняет обратную последовательность действий для восстановления исходного сообщения:

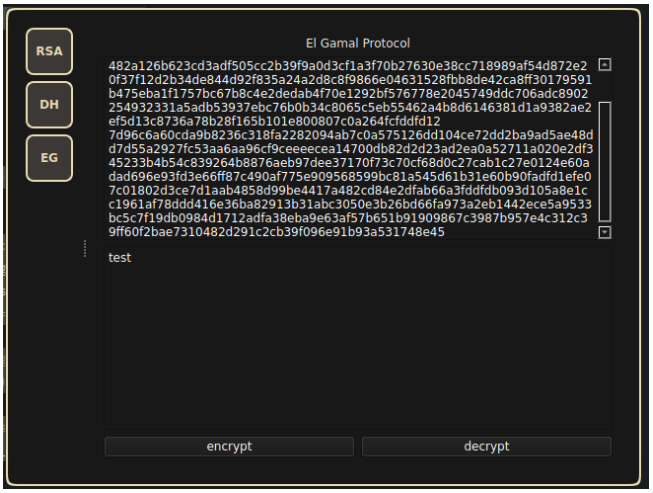
1. Входной шифртекст разделяется по пробелу на две шестнадцатеричные строки, представляющие компоненты r и c.
2. Обе строки преобразуются в большие целые числа ‘BigInt‘.
3. **Выполняется операция расшифрования.** Исходное сообщение s восстанавливается по формуле $s = c \cdot (r^d)^{-1} \pmod{p}$. Практически это реализуется в несколько шагов:
   - Вычисляется значение $r^d \pmod{p}$.
   - Для полученного значения находится мультипликативное обратное по модулю p с помощью вспомогательной функции ‘inverse()‘, реализующей расширенный алгоритм Евклида.
   - Компонента c умножается на найденное обратное значение по модулю p.
4. Расшифрованное число преобразуется обратно в шестнадцатеричную строку. Как и в RSA, выполняется коррекция: если строка имеет нечетную длину, в начало добавляется ведущий ноль для правильного последующего преобразования.
5. Скорректированная шестнадцатеричная строка преобразуется в ‘QByteArray‘, а затем в ‘QString‘ в кодировке UTF-8.

nvim

```
QString ElGamal::decrypt(const QString& ciphertext)
{
  if (m_p == 0 || m_d == 0)
  {
    throw std::runtime_error("ElGamal is not initialized or private key is
    ↪  missing.");
  }

  // 1. Разделяем шифртекст на две hex-строки (r и c).
  QStringList parts = ciphertext.split(' ');
  if (parts.size() != 2)
  {
    throw std::runtime_error("Invalid ciphertext format for ElGamal. Expected
    ↪  'r_hex c_hex'.");
  }

  // 2. Преобразуем hex-строки r и c обратно в BigInt.
  BigInt r("0x" + parts[0].toStdString());
  BigInt c("0x" + parts[1].toStdString());

  // 3. Расшифровываем: s = c * (r^d)^-1 mod p.
  //    Это эквивалентно s = c * r^(p-1-d) mod p, но вычисление через
  //    обратный элемент более прямолинейно.

  // Вычисляем r^d mod p
  BigInt r_d = boost::multiprecision::powm(r, m_d, m_p);

  // Находим мультипликативное обратное для r^d по модулю p
  BigInt r_d_inv = inverse(r_d, m_p);

  // Находим исходное сообщение s
  BigInt decrypted_message = (c * r_d_inv) % m_p;

  // 4. Преобразуем расшифрованное число обратно в QString (аналогично
  ↪  RSA::decrypt).
  std::string hex_str = decrypted_message.str(0, std::ios_base::hex);
  if (hex_str.length() % 2 != 0)
  {
    hex_str.insert(0, "0"); // Восстанавливаем потерянный ведущий ноль
  }

  QByteArray bytes = QByteArray::fromHex(QByteArray::fromStdString(hex_str));
  return QString::fromUtf8(bytes);
}
```



(a) Encryption



(b) Decryption

3.6 Пример работы схемы Эль-Гамаля

3.7 Программная реализация Диффи-Хеллмана

3.7.1 Структура класса и зависимости

Реализация протокола инкапсулирована в класс ‘DiffieHellman‘, который, аналогично ‘RSA‘, наследуется от абстрактного базового класса ‘Protocol‘. Это обеспечивает унифицированный интерфейс для всех криптографических протоколов в системе. Для работы с большими числами, необходимыми для криптографической стойкости, используется библиотека ‘Boost.Multiprecision‘ с типом ‘BigInt‘. Интеграция с пользовательским интерфейсом Qt обеспечивается за счет использования типа ‘QString‘ для текстовых данных.

nvim

```
#ifndef DIFFIE_HELLMAN_HPP
#define DIFFIE_HELLMAN_HPP

#include "protocol.hpp"

#include <boost/multiprecision/cpp_int.hpp>
#include <boost/random/mersenne_twister.hpp>

using BigInt = boost::multiprecision::cpp_int;

namespace CRYPTO
{
class DiffieHellman final : public Protocol
{
public:
  explicit DiffieHellman(unsigned int prime_bits = 512);
  void init() override;
```

nvim

```
    QString encrypt(const QString& plaintext) override;
    QString decrypt(const QString& ciphertext) override;

    void generateSharedSecret();

    BigInt getPrime();
    BigInt getPublicKey() const;

    void setOtherPartyPublicKey(const BigInt& key);
    void setOtherPartyPrime(const BigInt& prime);

private:
    void generateParameters();
    void generateKeys();

    BigInt generatePrime(unsigned int bits, boost::random::mt19937& rng);

private:
    // --- Параметры и ключи Диффи-Хеллмана ---
    unsigned int m_prime_bits;

    // Публичные параметры, общие для обеих сторон
    BigInt m_p; // Простое число-модуль `p`
    BigInt m_g; // Первообразный корень (генератор) `g`

    // Ключи этого участника
    BigInt m_private_key; // Закрытый ключ (секретное число `a` или `b`)
    BigInt m_public_key;  // Открытый ключ (A = g^a mod p или B = g^b mod p)

    // Данные, полученные от другого участника
    BigInt m_other_party_public_key; // Открытый ключ другого участника (A или
    ↪  B)

    // Финальный результат
    BigInt m_shared_secret; // Общий секретный ключ `s = (B^a) mod p = (A^b) mod
    ↪  p`
};

} // namespace CRYPTO

#endif // DIFFIE_HELLMAN_HPP
```

3.7.2 Инициализация и генерация ключей

В отличие от RSA, где одна сторона генерирует пару ключей, в протоколе DH каждый участник (назовем их Алиса и Боб) выполняет свой собственный процесс генерации. Этот процесс реализован в методе ‘init()‘, который вызывает внутренний метод ‘generateParametersAndKeys()‘.

1. **Согласование публичных параметров.** На первом этапе обе стороны должны согласовать два числа: большое простое число p (модуль) и целое число g (генератор или первообразный корень по модулю p). В данной реализации каждый участник генерирует свое простое число p с помощью вспомогательного метода ‘generatePrime()‘, который использует тест Миллера-

Рабина для проверки на простоту. Для симуляции предполагается, что участники "договариваются" об использовании одного из этих наборов параметров (на практике параметры p и g часто стандартизированы и общеизвестны). В качестве генератора g для простоты используется константное значение 5.

2. **Генерация секретного ключа.** Каждый участник генерирует свой собственный секретный ключ — случайное целое число. Алиса генерирует число a, а Боб — b. Эти числа должны находиться в диапазоне $[2, p-2]$ и храниться в строжайшем секрете.

3. **Вычисление открытого ключа.** Используя публичные параметры и свой секретный ключ, каждый участник вычисляет свой открытый ключ.

   - Алиса вычисляет: $A = g^a \pmod{p}$
   - Боб вычисляет: $B = g^b \pmod{p}$

   Для выполнения модульного возведения в степень используется оптимизированная функция 'boost::multiprecision::powm'.

nvim

```
void DiffieHellman::generateParameters()
{
  boost::random::mt19937 rng(std::random_device {}());

  // Шаг 1: Генерация большого простого числа p
  m_p = generatePrime(m_prime_bits, rng);

  // Шаг 2: Выбор генератора g
  m_g = 5;
}

void DiffieHellman::generateKeys()
{
  boost::random::mt19937 rng(std::random_device {}());

  // Шаг 3: Генерация закрытого ключа `a`
  boost::multiprecision::uniform_int_distribution<BigInt> dist(2, m_p - 2);
  m_private_key = dist(rng);

  // Шаг 4: Вычисление открытого ключа A = g^a mod p
  m_public_key = boost::multiprecision::powm(m_g, m_private_key, m_p);
}
```

3.7.3 Обмен ключами и вычисление общего секрета

После генерации ключей участники обмениваются своими открытыми ключами (A и B) по публичному каналу. Процесс обмена и последующего вычисления общего секрета управляется классом 'DiffieHellmanExchanger'.

1. **Обмен.** Алиса отправляет Бобу свой открытый ключ A, а Боб отправляет Алисе свой ключ B. В программной реализации это симулируется вызовом метода 'setOtherPartyPublicKey()'.

2. **Вычисление общего секрета.** Получив открытый ключ другой стороны, каждый участник может вычислить общий секретный ключ s. Важнейшим свойством протокола является то, что обе стороны получают одинаковый результат, выполняя вычисления независимо друг от друга.

   - Алиса вычисляет: $s = B^a \pmod{p} = (g^b)^a \pmod{p}$
   - Боб вычисляет: $s = A^b \pmod{p} = (g^a)^b \pmod{p}$

   В результате $s = g^{ab} \pmod{p}$ становится их общим секретом. Злоумышленник, перехвативший p, g, A, B, не может легко вычислить s, так как для этого ему потребуется решить задачу дискретного логарифмирования (найти a из A или b из B).

nvim

```
void DiffieHellman::generateSharedSecret()
{
  if (m_other_party_public_key == 0)
  {
    throw std::runtime_error("Открытый ключ другого участника не
    ↪  установлен.");
  }

  // Вычисление общего секрета: s = (B^a) mod p
  m_shared_secret = boost::multiprecision::powm(m_other_party_public_key,
   ↪  m_private_key, m_p);
}
```

3.7.4 Использование общего секрета для шифрования

Протокол Диффи-Хеллмана предназначен исключительно для обмена ключами и не является протоколом шифрования. Однако для демонстрации успешного установления общего секрета были реализованы методы ‘encrypt()‘ и ‘decrypt()‘, которые используют полученный ключ s для симметричного шифрования.

1. **Преобразование данных.** Входная строка ‘QString‘ преобразуется в массив байтов ‘QByteArray‘ в кодировке UTF-8, а затем в большое целое число ‘BigInt‘.

2. **Операция шифрования/расшифрования.** В качестве симметричного шифра используется простая операция побитового исключающего ”ИЛИ” (XOR) между числовым представлением сообщения и общим секретным ключом s.

   - Шифрование: ‘ciphertext_int = plaintext_int XOR s‘
   - Расшифрование: ‘plaintext_int = ciphertext_int XOR s‘

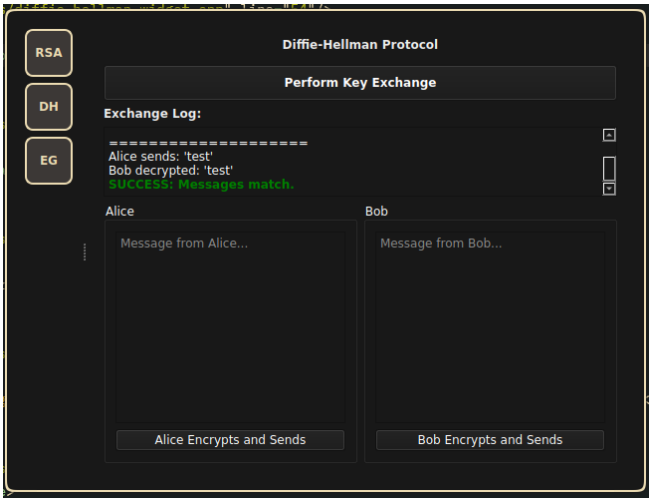
   Так как операция XOR является обратимой, один и тот же метод фактически выполняет и шифрование, и расшифрование.

3. **Формат передачи.** Для передачи зашифрованные данные преобразуются в шестнадцатеричную строку. При расшифровании выполняется обратное преобразование из шестнадцатеричной строки в ‘BigInt‘.

3.8 Пример работы протокола Диффи-Хеллмана

На рисунке ниже продемонстрирован интерфейс виджета, симулирующего работу протокола. После нажатия на кнопку ”Выполнить обмен ключами (DH)” в логе отображается процесс установления общего секрета. Затем Алиса и Боб могут обмениваться зашифрованными сообщениями, подтверждая, что они обладают одним и тем же ключом.

(a) Key exchange



(b) Message exchange

4 Тестирование и анализ

В данном разделе представлен анализ производительности криптографических алгоритмов RSA, ElGamal и протокола обмена ключами Diffie-Hellman. Тестирование проводилось для размеров ключей от 8 до 4096 бит с шагом, равным удвоению предыдущего значения.

Для наглядности все графики построены в логарифмическом масштабе по обеим осям (log-log). Ось X (размер ключа) имеет основание логарифма 2, что позволяет равномерно распределить тестовые точки. Ось Y (время выполнения) использует натуральный логарифм для эффективного отображения значений в широком диапазоне — от долей миллисекунды до десятков секунд.

4.1 Генерация и обмен ключами

На первом этапе сравнивается время, необходимое для создания ключевого материала. Для RSA и ElGamal это процесс генерации пары ключей, а для Diffie-Hellman — полный цикл обмена, в результате которого обе стороны получают общий секрет.

Анализ графика: График наглядно демонстрирует экспоненциальную зависимость времени генерации от размера ключа для всех алгоритмов.

- **Diffie-Hellman** показывает интересное поведение: для малых ключей (до 512 бит) он является одним из самых быстрых, но его производительность резко деградирует при увеличении ключа, делая его самым медленным на размерах 2048 и 4096 бит.

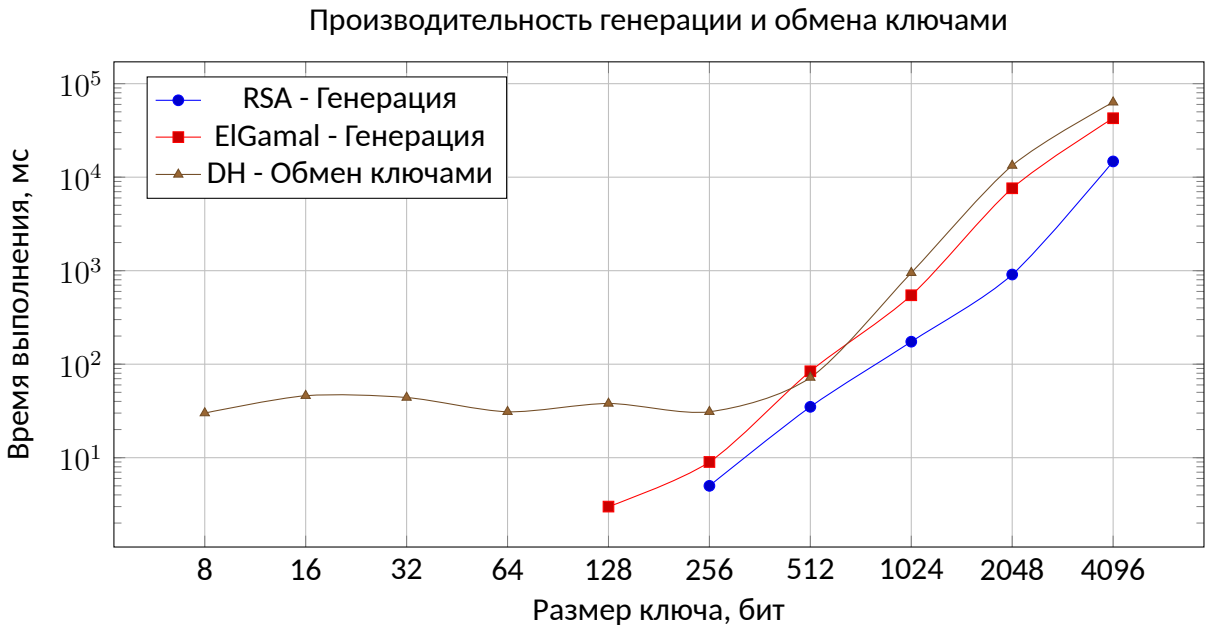


Figure 1: Сравнение времени генерации ключей и полного цикла обмена DH.

- **RSA** демонстрирует стабильный и предсказуемый рост. Он немного быстрее ElGamal.
- **ElGamal** является самым затратным по времени на этапе генерации ключей среди асимметричных шифров, что особенно заметно на больших размерах ключа.

4.2 Шифрование

На этом графике сравнивается производительность операций шифрования для RSA и ElGamal. Diffie-Hellman здесь не представлен, так как он является протоколом обмена ключами, а не шифрования.

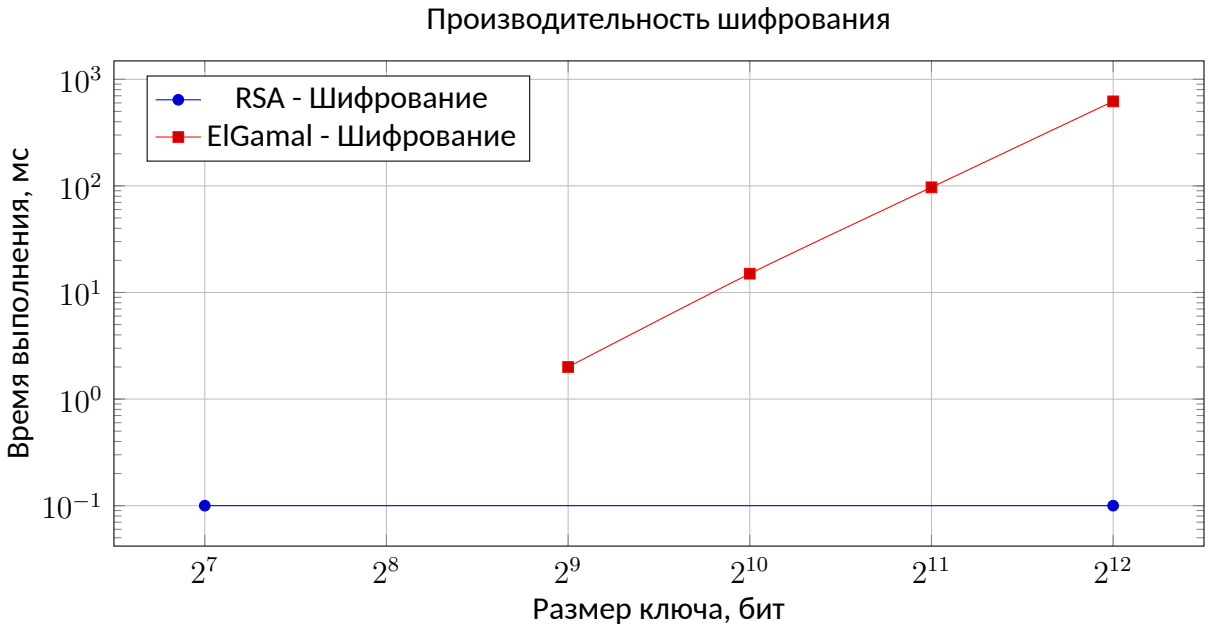


Figure 2: Сравнение времени шифрования.

Анализ графика: Здесь наблюдается колоссальная разница в производительности.

- **RSA** показывает практически мгновенное шифрование (в логах 0 мс). Это объясняется использованием фиксированной, небольшой публичной экспоненты ($e = 65537$), что делает операцию возведения в степень чрезвычайно быстрой. На графике это представлено как плоская линия на минимальном уровне.
- **ElGamal**, напротив, требует выполнения двух сложных модульных возведений в степень для каждой операции шифрования. Это делает его значительно медленнее RSA, и его производительность также экспоненциально зависит от размера ключа.

4.3 Расшифрование

Последний график сравнивает время, необходимое для расшифрования сообщения.

Анализ графика: В отличие от шифрования, производительность расшифрования для RSA и ElGamal оказывается **практически идентичной**.

- Это связано с тем, что в обоих случаях операция расшифрования является наиболее вычислительно сложной и требует модульного возведения в степень с использованием большой секретной экспоненты (секретного ключа d в RSA и x в ElGamal).
- Небольшие расхождения в значениях можно отнести на счет погрешностей измерения и особенностей конкретных сгенерированных ключей.

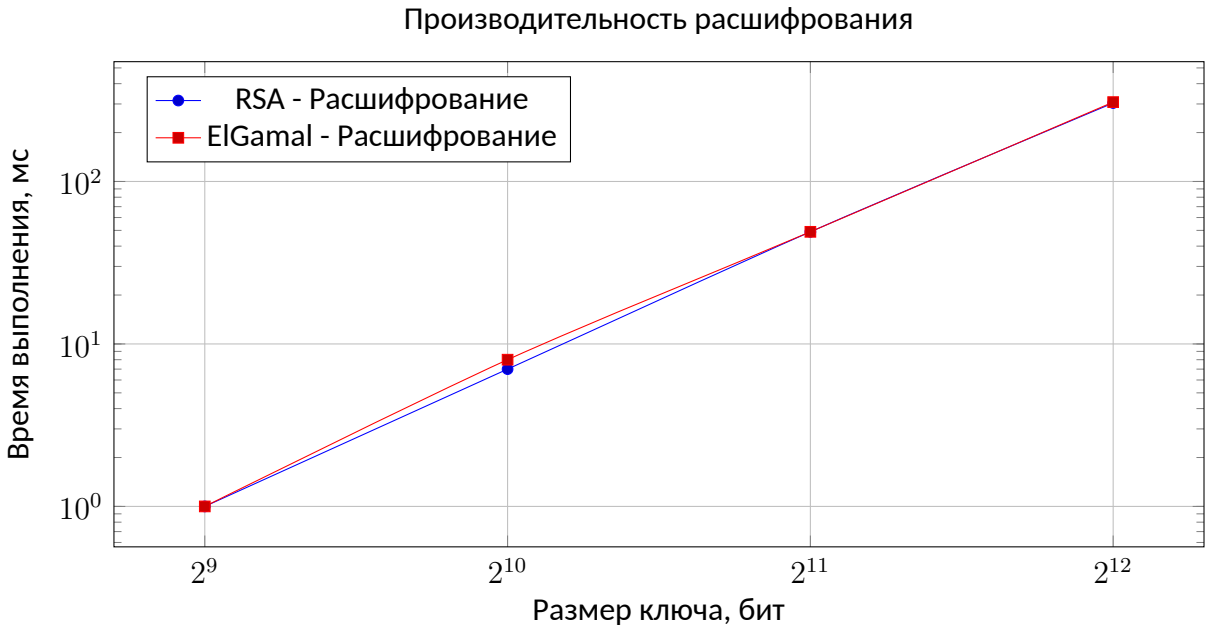


Figure 3: Сравнение времени расшифрования.

- Это подтверждает теоретические выкладки о том, что основная вычислительная нагрузка в асимметричных системах, как правило, ложится на владельца секретного ключа.

5 Заключение

В рамках данной курсовой работы была успешно выполнена задача по исследованию, практической реализации и сравнительному анализу трех фундаментальных криптографических примитивов: асимметричных шифросистем RSA и Эль-Гамаля, а также протокола обмена ключами Диффи-Хеллмана.

Работа была структурирована в три логических этапа:

- **Теоретическое исследование**, в ходе которого были рассмотрены математические основы, алгоритмы генерации ключей, шифрования, расшифрования, а также криптографическая стойкость и уязвимости каждой системы.
- **Программная реализация** на языке C++ с использованием объектно-ориентированного подхода. Создание общего базового класса `Protocol` позволило обеспечить унифицированный интерфейс и продемонстрировать модульность и расширяемость системы. Для работы с большими числами была использована библиотека Boost.Multiprecision, а для создания интерактивного демонстрационного приложения — фреймворк Qt.
- **Тестирование и анализ**, ставшие кульминацией работы. Было проведено комплексное тестирование производительности реализованных алгоритмов для размеров ключей в диапазоне от 8 до 4096 бит.

По результатам тестирования были сделаны следующие ключевые выводы:

- **Генерация ключей:** RSA демонстрирует наилучшую производительность и предсказуемый рост времени. Эль-Гамаль является значительно более затратным на этом этапе, в то время

как полный цикл обмена ключами Диффи-Хеллмана, будучи быстрым на малых ключах, показывает наихудшую масштабируемость.

- **Шифрование:** RSA обладает подавляющим преимуществом в скорости шифрования благодаря использованию малой публичной экспоненты ($e = 65537$). Шифрование в схеме Эль-Гамаля является вычислительно сложной операцией, что делает его на несколько порядков медленнее.
- **Расшифрование:** Производительность операций расшифрования для RSA и Эль-Гамаля оказалась практически идентичной. Это подтверждает теоретическое положение о том, что основная вычислительная нагрузка в обеих системах ложится на владельца секретного ключа, так как требует модульного возведения в степень с использованием большой секретной экспоненты.

Программная реализация протоколов на языке C++, использованная в данной работе, располагается в открытом доступе на GitHub по ссылке:

[https://github.com/Alexander-Chudnikov-Git/qt-crypto-algorithms.git]