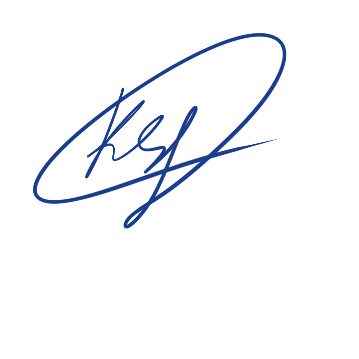
**Федеральное государственное бюджетное образовательное учреждение высшего образования "Белгородский государственный технологический университет им. В.Г. Шухова"**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем.

**Алгоритмы шифрования**

Выполнил:

Студент группы КБ-211

Коренев Д.Н.

Принял:

Твердохлеб В.В.

Оглавление

[Введение 3](#_Toc162305761)

[1. Теоритические сведения 4](#_Toc162305762)

[1.1. Алгоритм RSA 4](#_Toc162305763)

[1.2. Алгоритм Эль-Гамаля 6](#_Toc162305764)

[1.2.1. Подпись Эль-Гамаля 7](#_Toc162305765)

[1.3. Сеть Фейстеля 10](#_Toc162305766)

[1.4. Алгоритм DES 10](#_Toc162305767)

[2. Практическая часть 14](#_Toc162305768)

[2.1. Реализация алгоритма RSA 14](#_Toc162305769)

[2.2. Реализация алгоритма Эль-Гамаля 17](#_Toc162305770)

[2.2.1. Реализация алгоритма подписи Эль-Гамаля 20](#_Toc162305771)

[2.3. Реализация алгоритма DES 23](#_Toc162305772)

[2.4. Алгоритм перемешивания 36](#_Toc162305773)

[Приложения 38](#_Toc162305774)

Введение

В эпоху цифровизации, когда данные становятся ключевым активом, криптография выступает важнейшим инструментом защиты конфиденциальности, целостности и аутентификации информации. Эта курсовая работа посвящена анализу криптографических методов, с акцентом на изучение ключевых алгоритмов шифрования, которые обеспечивают защиту данных.

Мы подробно рассмотрим принципы работы симметричных и асимметричных шифров, их практическое применение и потенциальные уязвимости. В рамках практической части работы будет предпринята попытка создания собственного симметричного алгоритма шифрования. Этот экспериментальный проект позволит нам не только глубже понять механизмы шифрования, но и оценить трудности, с которыми сталкиваются разработчики при создании новых защищенных систем.

Целью данной работы является не только освоение теоретических основ криптографии, но и развитие практических навыков в сфере проектирования и анализа криптографических алгоритмов. Это даст возможность лучше понять важность криптографии в обеспечении безопасности современных информационных систем и необходимость продолжения исследований в этом направлении.

1. Теоритические сведения

1.1. Алгоритм RSA

Алгоритм RSA (Rivest-Shamir-Adleman) — это криптографический алгоритм с открытым ключом, который был разработан в 1977 году Рональдом Ривестом, Ади Шамиром и Леонардом Адлеманом. Он основан на математической проблеме факторизации больших чисел и является одним из первых и наиболее широко используемых методов шифрования с открытым ключом.

Генерация ключей:

1. Выбираются два больших простых числа и .
2. Вычисляется их произведение , которое называется модулем.
3. Вычисляется значение функции Эйлера: .
4. Выбирается целое число , которое меньше и взаимно простое с .
5. Вычисляется число , которое является мультипликативно обратным к по модулю , .

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Автоматически созданное описание

Рисунок 1. Схема создания ключей RSA.

Шифрование:

1. Открытый ключ состоит из пары чисел .
2. Чтобы зашифровать сообщение , вычисляется , где — это зашифрованный текст.

Изображение выглядит как текст, снимок экрана, Шрифт, белый

Автоматически созданное описание

Рисунок 2. Схема шифрования RSA.

Расшифрование:

1. Секретный ключ состоит из пары чисел .
2. Чтобы расшифровать зашифрованный текст , вычисляется , где — это исходное сообщение.

Изображение выглядит как текст, Шрифт, снимок экрана, белый

Автоматически созданное описание

Рисунок 3. Схема расшифрования RSA.

Применение алгоритма RSA:

Алгоритм RSA используется во многих областях, включая безопасную передачу данных, цифровые подписи, аутентификацию и другие криптографические приложения. Он является основой для таких стандартов, как PGP, S/MIME, TLS/SSL и многих других.

Безопасность алгоритма RSA:

Безопасность алгоритма RSA основывается на сложности задачи факторизации больших чисел. Пока что нет эффективного алгоритма, который мог бы быстро факторизовать большие числа, что делает RSA достаточно безопасным для современных приложений. Однако с развитием квантовых компьютеров и алгоритмов квантовой факторизации, таких как алгоритм Шора, безопасность RSA может быть под угрозой в будущем.

Алгоритм RSA остается одним из самых важных достижений в области криптографии и продолжает играть ключевую роль в обеспечении безопасности цифровой информации.

1.2. Алгоритм Эль-Гамаля

Алгоритм Эль-Гамаля — это криптосистема с открытым ключом, основанная на трудности вычисления дискретных логарифмов в конечном поле. Этот алгоритм был предложен Тахером Эль-Гамалем в 1985 году и включает в себя как алгоритм шифрования, так и алгоритм цифровой подписи.

Генерация ключей:

* 1. Генерируется случайное простое число .
  2. Выбирается целое число — первообразный корень по модулю .
  3. Выбирается случайное целое число , такое что .
  4. Вычисляется .
  5. Открытый ключ состоит из тройки чисел , а закрытый ключ — из числа .

Изображение выглядит как текст, снимок экрана, Шрифт, Графика

Автоматически созданное описание

Рисунок 4. Схема генерации ключей для алгоритма Эль-Гамаля.

Шифрование:

* 1. Сообщение должно быть меньше числа .
  2. Выбирается сессионный ключ — случайное целое число , такое что .
  3. Вычисляются числа и .
  4. Пара чисел является шифротекстом.

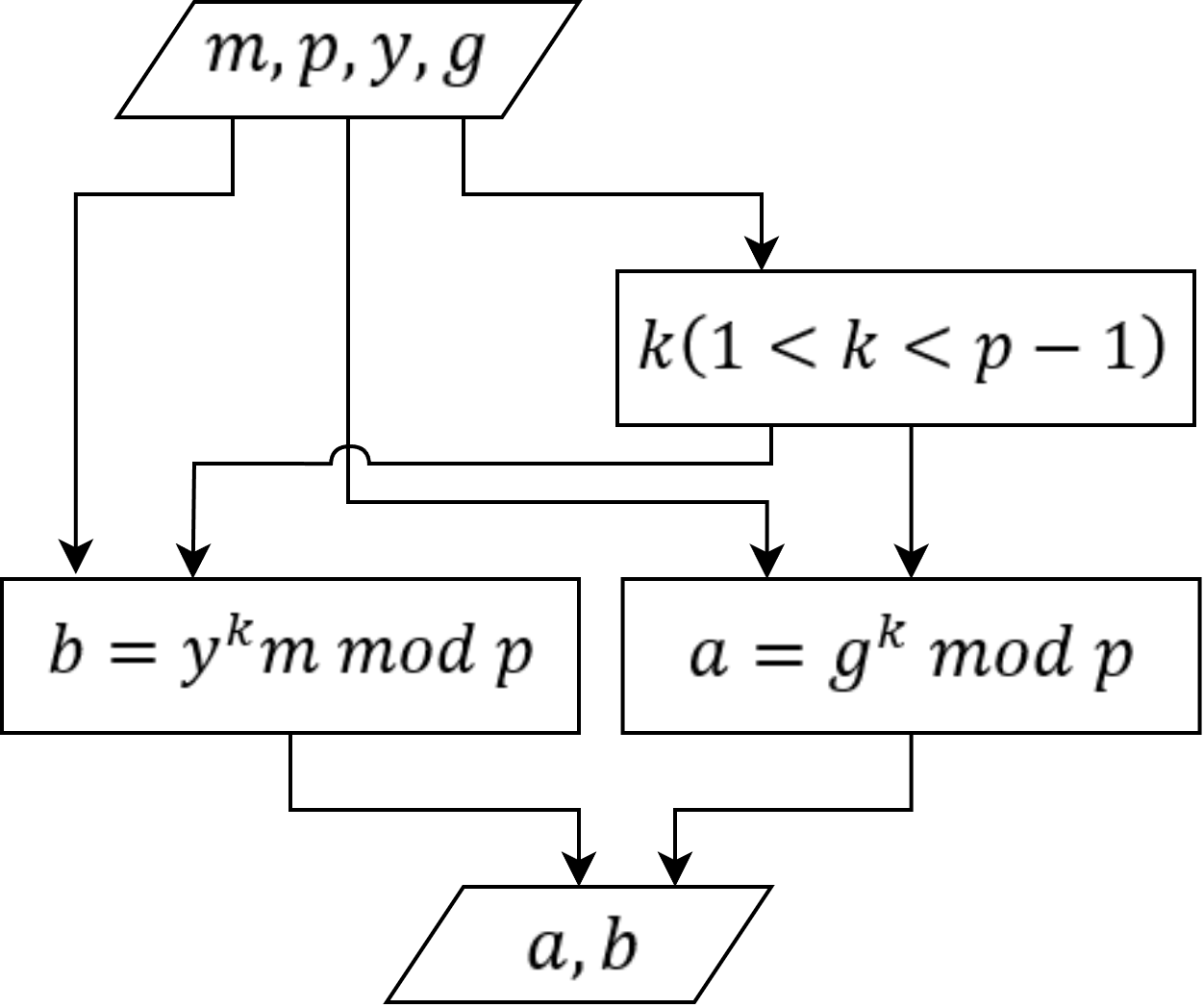


Рисунок 5. Схема шифрования алгоритмом Эль-Гамаля.

Расшифрование:

Зная закрытый ключ , исходное сообщение можно вычислить из шифротекста по формуле .

*Изображение выглядит как текст, Шрифт, снимок экрана, белый

Автоматически созданное описание*

Рисунок 6. Схема дешифрования алгоритмом эль-Гамаля.

Особенности и безопасность:

* Безопасность алгоритма основана на сложности задачи вычисления дискретных логарифмов.
* Алгоритм Эль-Гамаля лежит в основе стандартов электронной цифровой подписи в США (DSA) и России (ГОСТ Р 34.10-94).

Этот алгоритм является важным инструментом в криптографии и используется для обеспечения безопасности в электронных коммуникациях.

1.2.1. Подпись Эль-Гамаля

Алгоритм подписи Эль-Гамаля — это криптографический алгоритм, который использует асимметричное шифрование для создания цифровых подписей.

Генерация ключей:

1. Генерируется случайное простое число .

2. Выбирается целое число — первообразный корень по модулю .

3. Выбирается случайное целое число , такое что .

4. Вычисляется .

5. Открытый ключ состоит из тройки чисел , а закрытый ключ — из числа .

Изображение выглядит как текст, снимок экрана, Шрифт, Графика

Автоматически созданное описание

Рисунок 7. Схема генерации ключей для подписи алгориомом Эль-Гамаля.

Создание подписи:

1. Выбирается случайное число , взаимно простое с .

2. Вычисляется .

3. Вычисляется , где — это хеш-значение сообщения , а — мультипликативно обратное по модулю .

4. Подпись сообщения состоит из пары чисел .

Изображение выглядит как текст, снимок экрана, Шрифт, Графика

Автоматически созданное описание

Рисунок 8. Схема подписи алгоритмом Эль-Гамаля.

Проверка подписи:

1. Вычисляется .

2. Вычисляется .

3. Вычисляется .

4. Если , то подпись считается действительной.

Изображение выглядит как текст, снимок экрана, Шрифт, Графика

Автоматически созданное описание

Рисунок 9. Схема проверки подписи алгоритмом Эль-Гамаля.

Безопасность алгоритма:

Безопасность алгоритма подписи Эль-Гамаля зависит от сложности задачи вычисления дискретных логарифмов. Пока что нет эффективного способа решения этой задачи для больших чисел, что делает алгоритм достаточно надежным для использования в системах электронной подписи.

Алгоритм подписи Эль-Гамаля является важным инструментом в области криптографии и используется для подтверждения подлинности сообщений и документов в электронном виде.

1.3. Сеть Фейстеля

Сеть Фейстеля — это метод построения блочных шифров, который был разработан Хорстом Фейстелем. Основная идея сети Фейстеля заключается в разделении входного блока данных на две равные части и выполнении серии итераций, в ходе которых эти части обрабатываются и комбинируются с использованием подключей, производных от основного ключа шифрования¹.

Принцип работы сети Фейстеля:

1. Входные данные разделяются на две половины: левую и правую .

2. Итерации (раунды): на каждом шаге выполняются следующие действия:

- Выбирается подключ для текущего раунда.

- Применяется функция к правой половине и подключу , результат обозначается как .

- Результат функции складывается по модулю 2 (XOR) с левой половиной , получается новая правая половина .

- Правая половина становится новой левой половиной .

3. Выходные данные получаются путём объединения последних половин и после раундов и выполнения конечной перестановки.

Сеть Фейстеля позволяет легко реализовать шифрование и расшифрование, так как процесс расшифрования является просто обратным процессом шифрования с использованием тех же подключей в обратном порядке.

1.4. Алгоритм DES

Алгоритм DES (Data Encryption Standard) — это стандарт симметричного шифрования, который был разработан компанией IBM и утверждён правительством США в 1977 году. DES использует блоки данных размером 64 бита и ключ шифрования длиной 56 бит. Основой алгоритма является сеть Фейстеля, которая выполняет 16 раундов сложных преобразований.

Принцип работы алгоритма DES:

1. Начальная перестановка: Входной 64-битный блок данных подвергается начальной перестановке (IP), которая распределяет биты по определённой схеме.

2. Раунды шифрования: Следует 16 раундов, в каждом из которых используется подключ, полученный из основного ключа шифрования. В каждом раунде происходят следующие операции:

* Расширение: 32-битная половина блока расширяется до 48 бит.
* Смешивание с ключом: Расширенный блок смешивается с подключом с помощью операции XOR.
* S-блоки: 48-битный блок проходит через 8 S-блоков, которые заменяют 6-битные входные данные на 4-битные выходные.
* P-перестановка: 32-битный блок, полученный после S-блоков, подвергается ещё одной перестановке.
* Сложение по модулю 2: Результат P-перестановки складывается по модулю 2 с другой половиной входного блока.

3. Конечная перестановка: После 16 раундов выполняется конечная перестановка (IP-1), которая является обратной к начальной перестановке.

Изображение выглядит как диаграмма, текст, План, Технический чертеж

Автоматически созданное описание

Рисунок 10. Схема шифрования алгоритма DES.

Изображение выглядит как текст, диаграмма, чек, План

Автоматически созданное описание

Рисунок 11. Схема расшифрования алгоритма DES.

Криптостойкость алгоритма DES:

DES был стандартом шифрования на протяжении многих лет, но из-за ограниченной длины ключа в 56 бит, он стал уязвим для атак методом полного перебора. В связи с этим был разработан алгоритм Triple DES (3DES), который увеличивает стойкость за счёт троекратного применения DES с разными ключами.

DES сыграл важную роль в истории криптографии и заложил основы для разработки более современных и безопасных алгоритмов шифрования.

2. Практическая часть

2.1. Реализация алгоритма RSA

Исходный код реализации:

C++

/// @brief Check if the number is prime

/// @param x number

/// @return true if the number is prime, false otherwise

bool is\_prime(uint\_t x)

{

    if (x < 2)

    {

        return false;

    }

    for (uint\_t i = 2; i \* i <= x; i++)

    {

        if (x % i == 0)

        {

            return false;

        }

    }

    return true;

}

/// @brief Fast modular exponentiation

/// @tparam T type of the base

/// @param x base

/// @param pow power

/// @param mod modulo

/// @return x^pow % mod

template <typename T>

T pow\_mod(T x, uint\_t pow, uint\_t mod)

{

    T res = 1;

    for (uint\_t i = 0; i < pow; i++)

    {

        res = (res \* x) % mod;

    }

    return res;

}

/// @brief RSA modulo

/// @param p first prime number

/// @param q second prime number

/// @return RSA modulo

uint\_t rsa\_N(uint\_t p, uint\_t q)

{

    MASSERT(is\_prime(p), "p must be prime");

    MASSERT(is\_prime(q), "q must be prime");

    return p \* q;

}

/// @brief RSA t parameter

/// @param p first prime number

/// @param q second prime number

/// @return RSA t parameter (p - 1) \* (q - 1)

uint\_t rsa\_t(uint\_t p, uint\_t q)

{

    MASSERT(is\_prime(p), "p must be prime");

    MASSERT(is\_prime(q), "q must be prime");

    return (p - 1) \* (q - 1);

}

/// @brief Generate RSA encryption key

/// @param t RSA t parameter

/// @return RSA encryption key

uint\_t rsa\_public\_key(uint\_t t)

{

    std::vector<uint\_t> lst;

    for (uint\_t i = 2; i < t - 1; i++)

    {

        if (is\_prime(i) && t % i != 0)

        {

            lst.push\_back(i);

        }

    }

    return lst[rand() % lst.size()];

}

/// @brief Generate RSA decryption key

/// @param cif\_key RSA encryption key

/// @param t RSA t parameter

/// @return RSA decryption key

uint\_t rsa\_private\_key(uint\_t cif\_key, uint\_t t)

{

    for (uint\_t i = 1; i < t; i++)

    {

        if ((cif\_key \* i) % t == 1)

        {

            return i;

        }

    }

    return -1;

}

/// @brief RSA encrypt

/// @tparam T type of the data

/// @param x data

/// @param key RSA encryption key

/// @param N modulo

/// @return encrypted data

template <typename T>

uint\_t rsa\_encrypt(T x, uint\_t key, uint\_t N)

{

    return pow\_mod(x, key, N);

}

/// @brief RSA decrypt

/// @param x encrypted data

/// @param key RSA decryption key

/// @param N modulo

/// @return decrypted data

uint\_t rsa\_decrypt(uint\_t x, uint\_t key, uint\_t N)

{

    return pow\_mod(x, key, N);

}

Пример работы программы:

Изображение выглядит как текст, снимок экрана, дисплей, программное обеспечение

Автоматически созданное описание

Рисунок 12. Создание ключей RSA.

Изображение выглядит как текст, снимок экрана, программное обеспечение, мультимедиа

Автоматически созданное описание

Рисунок 13. Шифрование алгоритмом RSA.

Изображение выглядит как электроника, текст, снимок экрана, программное обеспечение

Автоматически созданное описание

Рисунок 14. Расшифрование алгоритмом RSA.

2.2. Реализация алгоритма Эль-Гамаля

Исходный код реализации:

С++

/// @brief Check if the number is prime

/// @param x number

/// @return true if the number is prime, false otherwise

bool is\_prime(uint\_t x)

{

    if (x < 2)

    {

        return false;

    }

    for (uint\_t i = 2; i \* i <= x; i++)

    {

        if (x % i == 0)

        {

            return false;

        }

    }

    return true;

}

/// @brief Fast modular exponentiation

/// @tparam T type of the base

/// @param x base

/// @param pow power

/// @param mod modulo

/// @return x^pow % mod

template <typename T>

T pow\_mod(T x, uint\_t pow, uint\_t mod)

{

    T res = 1;

    for (uint\_t i = 0; i < pow; i++)

    {

        res = (res \* x) % mod;

    }

    return res;

}

/// @brief Check if the number is a primitive root modulo p

/// @param g number

/// @param p modulo

/// @return true if g is a primitive root modulo p, false otherwise

bool is\_primitive\_root\_mod(uint\_t g, uint\_t p)

{

    for (uint\_t j = 1; j < p - 1; j++)

    {

        if (pow\_mod(g, j, p) == 1)

        {

            return false;

        }

    }

    return true;

}

/// @brief Get the primitive root modulo p

/// @param p modulo

/// @return primitive root modulo p

uint\_t primitive\_root\_mod(uint\_t p)

{

    for (uint\_t i = 2; i < p; i++)

    {

        if (is\_primitive\_root\_mod(i, p))

        {

            return i;

        }

    }

    return -1;

}

/// @brief Generate ElGamal session key

/// @param p modulo

/// @return ElGamal session key

uint\_t \_\_elg\_session\_key\_x(uint\_t p)

{

    return rand() % (p - 1) + 1;

}

/// @brief ElGamal y parameter

/// @param g primitive root modulo p

/// @param x ElGamal session key

/// @param p modulo

/// @return ElGamal y parameter

uint\_t \_\_elg\_y(uint\_t g, uint\_t x, uint\_t p)

{

    return pow\_mod(g, x, p);

}

/// @brief Make ElGamal private key

/// @param key\_x pointer to the result

/// @param p modulo

void elg\_private\_key(uint\_t \*key\_x, uint\_t p)

{

    \*key\_x = \_\_elg\_session\_key\_x(p);

}

/// @brief Make ElGamal public key

/// @param key\_y ElGamal y parameter, pointer to the result

/// @param key\_g generator, pointer to the result

/// @param x ElGamal session key

/// @param p modulo

void elg\_public\_key(uint\_t \*key\_y, uint\_t \*key\_g, uint\_t x, uint\_t p)

{

    \*key\_g = primitive\_root\_mod(p);

    \*key\_y = \_\_elg\_y(\*key\_g, x, p);

}

/// @brief ElGamal encrypt

/// @param a pointer to the result

/// @param b pointer to the result

/// @param m data

/// @param key\_y ElGamal y parameter

/// @param key\_g generator

/// @param p modulo

void elg\_encrypt(uint\_t \*a, uint\_t \*b, uint\_t m, uint\_t key\_y, uint\_t key\_g, uint\_t p)

{

    uint\_t k = \_\_elg\_session\_key\_x(p);

    \*a = pow\_mod(key\_g, k, p);

    \*b = (m \* pow\_mod(key\_y, k, p)) % p;

}

/// @brief ElGamal decrypt

/// @param a

/// @param b

/// @param key\_x ElGamal session key

/// @param p modulo

/// @return decrypted data

uint\_t elg\_decrypt(uint\_t a, uint\_t b, uint\_t key\_x, uint\_t p)

{

    return (b \* pow\_mod(a, p - 1 - key\_x, p)) % p;

}

Пример работы программы:

Изображение выглядит как текст, снимок экрана, программное обеспечение, мультимедиа

Автоматически созданное описание

Рисунок 15. Генерация ключей для алгоритма Эль-Гамаля.

Изображение выглядит как электроника, текст, снимок экрана, программное обеспечение

Автоматически созданное описание

Рисунок 16. Шифрование алгоритмом Эль-Гамаля.

Изображение выглядит как электроника, текст, снимок экрана, программное обеспечение

Автоматически созданное описание

Рисунок 17. Расшифрование алгоритмом Эль-Гамаля.

2.2.1. Реализация алгоритма подписи Эль-Гамаля

Исходный код реализации:

С++

/// @brief Coprime number to the given one

/// @param num number

/// @return coprime number

uint\_t coprime(uint\_t num)

{

    for (uint\_t i = 2; i < num; i++)

    {

        if (num % i != 0)

        {

            return i;

        }

    }

    return -1;

}

/// @brief Get the multiplicative inverse of a number

/// @param k number

/// @param p modulo

/// @return multiplicative inverse of k modulo p

uint\_t multiplicative\_inverse(uint\_t k, uint\_t p)

{

    for (uint\_t i = 1; i < p; i++)

    {

        if ((k \* i) % p == 1)

        {

            return i;

        }

    }

    return -1;

}

/// @brief ElGamal signature k parameter

/// @param p modulo

/// @return ElGamal signature k parameter

uint\_t \_\_elgsig\_k(uint\_t p)

{

    return coprime(p - 1);

}

/// @brief ElGamal signature a parameter

/// @param g generator

/// @param k ElGamal signature k parameter

/// @param p modulo

/// @return ElGamal signature a parameter

uint\_t \_\_elgsig\_a(uint\_t g, uint\_t k, uint\_t p)

{

    return pow\_mod(g, k, p);

}

/// @brief ElGamal signature reverse k parameter

/// @param k ElGamal signature k parameter

/// @param p modulo

/// @return ElGamal signature reverse k parameter

uint\_t \_\_elgsig\_reverse\_k(uint\_t k, uint\_t p)

{

    return multiplicative\_inverse(k, p - 1);

}

/// @brief ElGamal signature b parameter

/// @param m data

/// @param k ElGamal signature k parameter

/// @param x ElGamal session key

/// @param a ElGamal signature a parameter

/// @param p modulo

/// @return ElGamal signature b parameter

uint\_t \_\_elgsig\_b(uint\_t m, uint\_t k, uint\_t x, uint\_t a, uint\_t p)

{

    uint\_t mmod = (\_\_elgsig\_reverse\_k(k, p) \* (m - x \* a)) % (p - 1);

    // 'C' peculiarity about mod operation:

    return mmod >= 0 ? mmod : mmod + p - 1;

}

/// @brief ElGamal signature

/// @param a pointer to the result

/// @param b pointer to the result

/// @param key\_x ElGamal session key

/// @param key\_g generator

/// @param p modulo

/// @param m data

void elgsig\_sign(uint\_t \*a, uint\_t \*b,

                 uint\_t key\_x, uint\_t key\_g,

                 uint\_t p, uint\_t m)

{

    uint\_t k = \_\_elgsig\_k(p);

    \*a = \_\_elgsig\_a(key\_g, k, p);

    \*b = \_\_elgsig\_b(m, k, key\_x, \*a, p);

}

/// @brief ElGamal signature check

/// @param key\_y ElGamal y parameter

/// @param key\_g generator

/// @param a ElGamal signature a parameter

/// @param b ElGamal signature b parameter

/// @param p modulo

/// @param m data

/// @return true if the signature is valid, false otherwise

bool elgsig\_check(uint\_t key\_y, uint\_t key\_g,

                  uint\_t a, uint\_t b, uint\_t p, uint\_t m)

{

    return (pow\_mod(key\_y, a, p) \* pow\_mod(a, b, p)) % p == pow\_mod(

                                                                key\_g, m, p);

}

Пример работы программы:

Изображение выглядит как текст, снимок экрана, программное обеспечение, мультимедиа

Автоматически созданное описание

Рисунок 18. Генерация ключей для подписи с использование алгоритма Эль-Гамаля.

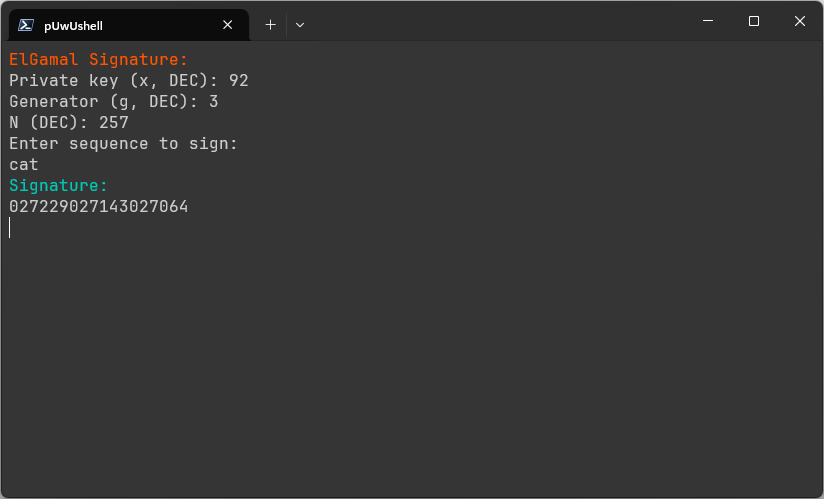


Рисунок 19. Создание подписи алгоритмом Эль-Гамаля.

Изображение выглядит как электроника, текст, снимок экрана, программное обеспечение

Автоматически созданное описание

Рисунок 20. Проверка подписи алгоритмом Эль-Гамаля.

2.3. Реализация алгоритма DES

Исходный код реализации:

С++

#define DES\_ENCRYPTION\_MODE 1

#define DES\_DECRYPTION\_MODE 0

typedef struct des\_key\_set

{

    byte\_t k[8];

    byte\_t c[4];

    byte\_t d[4];

} des\_key\_set;

byte\_t \_\_des\_initial\_key\_permutaion[] = {0x39, 0x31, 0x29, 0x21,

                                         0x19, 0x11, 0x09, 0x01,

                                         0x3A, 0x32, 0x2A, 0x22,

                                         0x1A, 0x12, 0x0A, 0x02,

                                         0x3B, 0x33, 0x2B, 0x23,

                                         0x1B, 0x13, 0x0B, 0x03,

                                         0x3C, 0x34, 0x2C, 0x24,

                                         0x3F, 0x37, 0x2F, 0x27,

                                         0x1F, 0x17, 0x0F, 0x07,

                                         0x3E, 0x36, 0x2E, 0x26,

                                         0x1E, 0x16, 0x0E, 0x06,

                                         0x3D, 0x35, 0x2D, 0x25,

                                         0x1D, 0x15, 0x0D, 0x05,

                                         0x1C, 0x14, 0x0C, 0x04};

// Initial permutation (IP)

byte\_t \_\_des\_initial\_message\_permutation[] = {0x3A, 0x32, 0x2A, 0x22,

                                              0x1A, 0x12, 0x0A, 0x02,

                                              0x3C, 0x34, 0x2C, 0x24,

                                              0x1C, 0x14, 0x0C, 0x04,

                                              0x3E, 0x36, 0x2E, 0x26,

                                              0x1E, 0x16, 0x0E, 0x06,

                                              0x40, 0x38, 0x30, 0x28,

                                              0x20, 0x18, 0x10, 0x08,

                                              0x39, 0x31, 0x29, 0x21,

                                              0x19, 0x11, 0x09, 0x01,

                                              0x3B, 0x33, 0x2B, 0x23,

                                              0x1B, 0x13, 0x0B, 0x03,

                                              0x3D, 0x35, 0x2D, 0x25,

                                              0x1D, 0x15, 0x0D, 0x05,

                                              0x3F, 0x37, 0x2F, 0x27,

                                              0x1F, 0x17, 0x0F, 0x07};

// 17

int \_\_des\_key\_shift\_sizes[] = {-1,

                               1, 1, 2, 2, 2, 2, 2, 2,

                               1, 2, 2, 2, 2, 2, 2, 1};

// Subkey permutation

byte\_t \_\_des\_sub\_key\_permutation[] = {0x0E, 0x11, 0x0B, 0x18, 0x01, 0x05,

                                      0x03, 0x1C, 0x0F, 0x06, 0x15, 0x0A,

                                      0x17, 0x13, 0x0C, 0x04, 0x1A, 0x08,

                                      0x10, 0x07, 0x1B, 0x14, 0x0D, 0x02,

                                      0x29, 0x34, 0x1F, 0x25, 0x2F, 0x37,

                                      0x1E, 0x28, 0x33, 0x2D, 0x21, 0x30,

                                      0x2C, 0x31, 0x27, 0x38, 0x22, 0x35,

                                      0x2E, 0x2A, 0x32, 0x24, 0x1D, 0x20};

// Expansion table (E)

byte\_t \_\_des\_message\_expansion[] = {0x20, 0x01, 0x02, 0x03, 0x04, 0x05,

                                    0x04, 0x05, 0x06, 0x07, 0x08, 0x09,

                                    0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D,

                                    0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11,

                                    0x10, 0x11, 0x12, 0x13, 0x14, 0x15,

                                    0x14, 0x15, 0x16, 0x17, 0x18, 0x19,

                                    0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D,

                                    0x1C, 0x1D, 0x1E, 0x1F, 0x20, 0x01};

// S\_i transformation tables

byte\_t \_\_des\_S1[] = {0x0E, 0x04, 0x0D, 0x01, 0x02, 0x0F, 0x0B, 0x08,

                     0x03, 0x0A, 0x06, 0x0C, 0x05, 0x09, 0x00, 0x07,

                     0x00, 0x0F, 0x07, 0x04, 0x0E, 0x02, 0x0D, 0x01,

                     0x0A, 0x06, 0x0C, 0x0B, 0x09, 0x05, 0x03, 0x08,

                     0x04, 0x01, 0x0E, 0x08, 0x0D, 0x06, 0x02, 0x0B,

                     0x0F, 0x0C, 0x09, 0x07, 0x03, 0x0A, 0x05, 0x00,

                     0x0F, 0x0C, 0x08, 0x02, 0x04, 0x09, 0x01, 0x07,

                     0x05, 0x0B, 0x03, 0x0E, 0x0A, 0x00, 0x06, 0x0D};

byte\_t \_\_des\_S2[] = {0x0F, 0x01, 0x08, 0x0E, 0x06, 0x0B, 0x03, 0x04,

                     0x09, 0x07, 0x02, 0x0D, 0x0C, 0x00, 0x05, 0x0A,

                     0x03, 0x0D, 0x04, 0x07, 0x0F, 0x02, 0x08, 0x0E,

                     0x0C, 0x00, 0x01, 0x0A, 0x06, 0x09, 0x0B, 0x05,

                     0x00, 0x0E, 0x07, 0x0B, 0x0A, 0x04, 0x0D, 0x01,

                     0x05, 0x08, 0x0C, 0x06, 0x09, 0x03, 0x02, 0x0F,

                     0x0D, 0x08, 0x0A, 0x01, 0x03, 0x0F, 0x04, 0x02,

                     0x0B, 0x06, 0x07, 0x0C, 0x00, 0x05, 0x0E, 0x09};

byte\_t \_\_des\_S3[] = {0x0A, 0x00, 0x09, 0x0E, 0x06, 0x03, 0x0F, 0x05,

                     0x01, 0x0D, 0x0C, 0x07, 0x0B, 0x04, 0x02, 0x08,

                     0x0D, 0x07, 0x00, 0x09, 0x03, 0x04, 0x06, 0x0A,

                     0x02, 0x08, 0x05, 0x0E, 0x0C, 0x0B, 0x0F, 0x01,

                     0x0D, 0x06, 0x04, 0x09, 0x08, 0x0F, 0x03, 0x00,

                     0x0B, 0x01, 0x02, 0x0C, 0x05, 0x0A, 0x0E, 0x07,

                     0x01, 0x0A, 0x0D, 0x00, 0x06, 0x09, 0x08, 0x07,

                     0x04, 0x0F, 0x0E, 0x03, 0x0B, 0x05, 0x02, 0x0C};

byte\_t \_\_des\_S4[] = {0x07, 0x0D, 0x0E, 0x03, 0x00, 0x06, 0x09, 0x0A,

                     0x01, 0x02, 0x08, 0x05, 0x0B, 0x0C, 0x04, 0x0F,

                     0x0D, 0x08, 0x0B, 0x05, 0x06, 0x0F, 0x00, 0x03,

                     0x04, 0x07, 0x02, 0x0C, 0x01, 0x0A, 0x0E, 0x09,

                     0x0A, 0x06, 0x09, 0x00, 0x0C, 0x0B, 0x07, 0x0D,

                     0x0F, 0x01, 0x03, 0x0E, 0x05, 0x02, 0x08, 0x04,

                     0x03, 0x0F, 0x00, 0x06, 0x0A, 0x01, 0x0D, 0x08,

                     0x09, 0x04, 0x05, 0x0B, 0x0C, 0x07, 0x02, 0x0E};

byte\_t \_\_des\_S5[] = {0x02, 0x0C, 0x04, 0x01, 0x07, 0x0A, 0x0B, 0x06,

                     0x08, 0x05, 0x03, 0x0F, 0x0D, 0x00, 0x0E, 0x09,

                     0x0E, 0x0B, 0x02, 0x0C, 0x04, 0x07, 0x0D, 0x01,

                     0x05, 0x00, 0x0F, 0x0A, 0x03, 0x09, 0x08, 0x06,

                     0x04, 0x02, 0x01, 0x0B, 0x0A, 0x0D, 0x07, 0x08,

                     0x0F, 0x09, 0x0C, 0x05, 0x06, 0x03, 0x00, 0x0E,

                     0x0B, 0x08, 0x0C, 0x07, 0x01, 0x0E, 0x02, 0x0D,

                     0x06, 0x0F, 0x00, 0x09, 0x0A, 0x04, 0x05, 0x03};

byte\_t \_\_des\_S6[] = {0x0C, 0x01, 0x0A, 0x0F, 0x09, 0x02, 0x06, 0x08,

                     0x00, 0x0D, 0x03, 0x04, 0x0E, 0x07, 0x05, 0x0B,

                     0x0A, 0x0F, 0x04, 0x02, 0x07, 0x0C, 0x09, 0x05,

                     0x06, 0x01, 0x0D, 0x0E, 0x00, 0x0B, 0x03, 0x08,

                     0x09, 0x0E, 0x0F, 0x05, 0x02, 0x08, 0x0C, 0x03,

                     0x07, 0x00, 0x04, 0x0A, 0x01, 0x0D, 0x0B, 0x06,

                     0x04, 0x03, 0x02, 0x0C, 0x09, 0x05, 0x0F, 0x0A,

                     0x0B, 0x0E, 0x01, 0x07, 0x06, 0x00, 0x08, 0x0D};

byte\_t \_\_des\_S7[] = {0x04, 0x0B, 0x02, 0x0E, 0x0F, 0x00, 0x08, 0x0D,

                     0x03, 0x0C, 0x09, 0x07, 0x05, 0x0A, 0x06, 0x01,

                     0x0D, 0x00, 0x0B, 0x07, 0x04, 0x09, 0x01, 0x0A,

                     0x0E, 0x03, 0x05, 0x0C, 0x02, 0x0F, 0x08, 0x06,

                     0x01, 0x04, 0x0B, 0x0D, 0x0C, 0x03, 0x07, 0x0E,

                     0x0A, 0x0F, 0x06, 0x08, 0x00, 0x05, 0x09, 0x02,

                     0x06, 0x0B, 0x0D, 0x08, 0x01, 0x04, 0x0A, 0x07,

                     0x09, 0x05, 0x00, 0x0F, 0x0E, 0x02, 0x03, 0x0C};

byte\_t \_\_des\_S8[] = {0x0D, 0x02, 0x08, 0x04, 0x06, 0x0F, 0x0B, 0x01,

                     0x0A, 0x09, 0x03, 0x0E, 0x05, 0x00, 0x0C, 0x07,

                     0x01, 0x0F, 0x0D, 0x08, 0x0A, 0x03, 0x07, 0x04,

                     0x0C, 0x05, 0x06, 0x0B, 0x00, 0x0E, 0x09, 0x02,

                     0x07, 0x0B, 0x04, 0x01, 0x09, 0x0C, 0x0E, 0x02,

                     0x00, 0x06, 0x0A, 0x0D, 0x0F, 0x03, 0x05, 0x08,

                     0x02, 0x01, 0x0E, 0x07, 0x04, 0x0A, 0x08, 0x0D,

                     0x0F, 0x0C, 0x09, 0x00, 0x03, 0x05, 0x06, 0x0B};

// Permutation table (P)

byte\_t \_\_des\_right\_sub\_msg\_permut[] = {0x10, 0x07, 0x14, 0x15,

                                       0x1D, 0x0C, 0x1C, 0x11,

                                       0x01, 0x0F, 0x17, 0x1A,

                                       0x05, 0x12, 0x1F, 0x0A,

                                       0x02, 0x08, 0x18, 0x0E,

                                       0x20, 0x1B, 0x03, 0x09,

                                       0x13, 0x0D, 0x1E, 0x06,

                                       0x16, 0x0B, 0x04, 0x19};

// Final permutation (IP^-1)

byte\_t \_\_des\_final\_msg\_permut[] = {0x28, 0x08, 0x30, 0x10,

                                   0x38, 0x18, 0x40, 0x20,

                                   0x27, 0x07, 0x2F, 0x0F,

                                   0x37, 0x17, 0x3F, 0x1F,

                                   0x26, 0x06, 0x2E, 0x0E,

                                   0x36, 0x16, 0x3E, 0x1E,

                                   0x25, 0x05, 0x2D, 0x0D,

                                   0x35, 0x15, 0x3D, 0x1D,

                                   0x24, 0x04, 0x2C, 0x0C,

                                   0x34, 0x14, 0x3C, 0x1C,

                                   0x23, 0x03, 0x2B, 0x0B,

                                   0x33, 0x13, 0x3B, 0x1B,

                                   0x22, 0x02, 0x2A, 0x0A,

                                   0x32, 0x12, 0x3A, 0x1A,

                                   0x21, 0x01, 0x29, 0x09,

                                   0x31, 0x11, 0x39, 0x19};

/// @brief Check if the key is weak in terms of DES

/// @param key

/// @return true if the key is weak, false otherwise

bool \_\_des\_is\_key\_weak(byte\_t \*key)

{

    byte\_t weak\_key1[] = {0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01};

    byte\_t weak\_key2[] = {0xFE, 0xFE, 0xFE, 0xFE, 0xFE, 0xFE, 0xFE, 0xFE};

    byte\_t weak\_key3[] = {0xE0, 0xE0, 0xE0, 0xE0, 0xF1, 0xF1, 0xF1, 0xF1};

    byte\_t weak\_key4[] = {0x1F, 0x1F, 0x1F, 0x1F, 0x0E, 0x0E, 0x0E, 0x0E};

    return memcmp(key, weak\_key1, 8) == 0 ||

           memcmp(key, weak\_key2, 8) == 0 ||

           memcmp(key, weak\_key3, 8) == 0 ||

           memcmp(key, weak\_key4, 8) == 0;

}

/// @brief Check if the key is semi-weak in terms of DES

/// @param key

/// @return true if the key is semi-weak, false otherwise

bool \_\_des\_is\_key\_semi\_weak(byte\_t \*key)

{

    byte\_t s\_weak\_key\_01[] = {0x01, 0xFE, 0x01, 0xFE, 0x01, 0xFE, 0x01, 0xFE};

    byte\_t s\_weak\_key\_02[] = {0xFE, 0x01, 0xFE, 0x01, 0xFE, 0x01, 0xFE, 0x01};

    byte\_t s\_weak\_key\_03[] = {0x1F, 0xE0, 0x1F, 0xE0, 0x0E, 0xF1, 0x0E, 0xF1};

    byte\_t s\_weak\_key\_04[] = {0xE0, 0x1F, 0xE0, 0x1F, 0xF1, 0x0E, 0xF1, 0x0E};

    byte\_t s\_weak\_key\_05[] = {0x01, 0xE0, 0x01, 0xE0, 0x01, 0xF1, 0x01, 0xF1};

    byte\_t s\_weak\_key\_06[] = {0xE0, 0x01, 0xE0, 0x01, 0xF1, 0x01, 0xF1, 0x01};

    byte\_t s\_weak\_key\_07[] = {0x1F, 0xFE, 0x1F, 0xFE, 0x0E, 0xFE, 0x0E, 0xFE};

    byte\_t s\_weak\_key\_08[] = {0xFE, 0x1F, 0xFE, 0x1F, 0xFE, 0x0E, 0xFE, 0x0E};

    byte\_t s\_weak\_key\_09[] = {0x01, 0x1F, 0x01, 0x1F, 0x01, 0x0E, 0x01, 0x0E};

    byte\_t s\_weak\_key\_10[] = {0x1F, 0x01, 0x1F, 0x01, 0x0E, 0x01, 0x0E, 0x01};

    byte\_t s\_weak\_key\_11[] = {0xFE, 0xE0, 0xFE, 0xE0, 0xFE, 0xF1, 0xFE, 0xF1};

    byte\_t s\_weak\_key\_12[] = {0xE0, 0xFE, 0xE0, 0xFE, 0xF1, 0xFE, 0xF1, 0xFE};

    return memcmp(key, s\_weak\_key\_01, 8) == 0 ||

           memcmp(key, s\_weak\_key\_02, 8) == 0 ||

           memcmp(key, s\_weak\_key\_03, 8) == 0 ||

           memcmp(key, s\_weak\_key\_04, 8) == 0 ||

           memcmp(key, s\_weak\_key\_05, 8) == 0 ||

           memcmp(key, s\_weak\_key\_06, 8) == 0 ||

           memcmp(key, s\_weak\_key\_07, 8) == 0 ||

           memcmp(key, s\_weak\_key\_08, 8) == 0 ||

           memcmp(key, s\_weak\_key\_09, 8) == 0 ||

           memcmp(key, s\_weak\_key\_10, 8) == 0 ||

           memcmp(key, s\_weak\_key\_11, 8) == 0 ||

           memcmp(key, s\_weak\_key\_12, 8) == 0;

}

/// @brief Check if the key is acceptable in terms of DES

/// @param key

/// @return true if the key is acceptable, false otherwise

bool \_\_des\_is\_key\_acceptable(byte\_t \*key)

{

    return !\_\_des\_is\_key\_weak(key) && !\_\_des\_is\_key\_semi\_weak(key);

}

/// @brief Generate 16(+1) sub keys from the main key

/// @param main\_key 64 bit key

/// @param key\_sets array of (+1)16 key sets

void \_\_des\_generate\_sub\_keys(byte\_t \*main\_key, des\_key\_set \*key\_sets)

{

    int i, j;

    int shift\_size;

    byte\_t shift\_byte,

        first\_shift\_bits,

        second\_shift\_bits,

        third\_shift\_bits,

        fourth\_shift\_bits;

    // Zero out first key set's k

    for (i = 0; i < 8; i++)

    {

        key\_sets[0].k[i] = 0;

    }

    for (j = 1; j < 17; j++)

    {

        for (i = 0; i < 8; i++)

        {

            key\_sets[j].k[i] = 0;

        }

    }

    // Generate first key set's k

    for (i = 0; i < 56; i++)

    {

        shift\_size = \_\_des\_initial\_key\_permutaion[i];

        shift\_byte = 0x80 >> ((shift\_size - 1) % 8);

        shift\_byte &= main\_key[(shift\_size - 1) / 8];

        shift\_byte <<= ((shift\_size - 1) % 8);

        key\_sets[0].k[i / 8] |= (shift\_byte >> i % 8);

    }

    // Copy first 3 bytes of k to c

    for (i = 0; i < 3; i++)

    {

        key\_sets[0].c[i] = key\_sets[0].k[i];

    }

    // Copy last byte of k to c and mask it

    key\_sets[0].c[3] = key\_sets[0].k[3] & 0xF0;

    // Copy last 4 bytes of k to d

    for (i = 0; i < 3; i++)

    {

        key\_sets[0].d[i] = (key\_sets[0].k[i + 3] & 0x0F) << 4;

        key\_sets[0].d[i] |= (key\_sets[0].k[i + 4] & 0xF0) >> 4;

    }

    // Mask last byte of d

    key\_sets[0].d[3] = (key\_sets[0].k[6] & 0x0F) << 4;

    // Generate 16 sub keys

    for (i = 1; i < 17; i++)

    {

        // Copy previous key set to current

        for (j = 0; j < 4; j++)

        {

            key\_sets[i].c[j] = key\_sets[i - 1].c[j];

            key\_sets[i].d[j] = key\_sets[i - 1].d[j];

        }

        shift\_size = \_\_des\_key\_shift\_sizes[i];

        if (shift\_size == 1)

        {

            shift\_byte = 0x80;

        }

        else

        {

            shift\_byte = 0xC0;

        }

        // Process C

        first\_shift\_bits = shift\_byte & key\_sets[i].c[0];

        second\_shift\_bits = shift\_byte & key\_sets[i].c[1];

        third\_shift\_bits = shift\_byte & key\_sets[i].c[2];

        fourth\_shift\_bits = shift\_byte & key\_sets[i].c[3];

        key\_sets[i].c[0] <<= shift\_size;

        key\_sets[i].c[0] |= (second\_shift\_bits >> (8 - shift\_size));

        key\_sets[i].c[1] <<= shift\_size;

        key\_sets[i].c[1] |= (third\_shift\_bits >> (8 - shift\_size));

        key\_sets[i].c[2] <<= shift\_size;

        key\_sets[i].c[2] |= (fourth\_shift\_bits >> (8 - shift\_size));

        key\_sets[i].c[3] <<= shift\_size;

        key\_sets[i].c[3] |= (first\_shift\_bits >> (4 - shift\_size));

        // Process D

        first\_shift\_bits = shift\_byte & key\_sets[i].d[0];

        second\_shift\_bits = shift\_byte & key\_sets[i].d[1];

        third\_shift\_bits = shift\_byte & key\_sets[i].d[2];

        fourth\_shift\_bits = shift\_byte & key\_sets[i].d[3];

        key\_sets[i].d[0] <<= shift\_size;

        key\_sets[i].d[0] |= (second\_shift\_bits >> (8 - shift\_size));

        key\_sets[i].d[1] <<= shift\_size;

        key\_sets[i].d[1] |= (third\_shift\_bits >> (8 - shift\_size));

        key\_sets[i].d[2] <<= shift\_size;

        key\_sets[i].d[2] |= (fourth\_shift\_bits >> (8 - shift\_size));

        key\_sets[i].d[3] <<= shift\_size;

        key\_sets[i].d[3] |= (first\_shift\_bits >> (4 - shift\_size));

        // Merge C and D to generate K

        for (j = 0; j < 48; j++)

        {

            shift\_size = \_\_des\_sub\_key\_permutation[j];

            if (shift\_size <= 28)

            {

                shift\_byte = 0x80 >> ((shift\_size - 1) % 8);

                shift\_byte &= key\_sets[i].c[(shift\_size - 1) / 8];

                shift\_byte <<= ((shift\_size - 1) % 8);

            }

            else

            {

                shift\_byte = 0x80 >> ((shift\_size - 29) % 8);

                shift\_byte &= key\_sets[i].d[(shift\_size - 29) / 8];

                shift\_byte <<= ((shift\_size - 29) % 8);

            }

            key\_sets[i].k[j / 8] |= (shift\_byte >> j % 8);

        }

    }

}

/// @brief Process a 64 bit block of data using DES

/// @param data\_block

/// @param processed\_block

/// @param key\_sets array of 16(+1) key sets

/// @param mode 1 for encryption, 0 for decryption

void \_\_des\_process\_data\_block(byte\_t \*data\_block,

                              byte\_t \*processed\_block,

                              des\_key\_set \*key\_sets,

                              int mode)

{

    int i, k;

    int shift\_size;

    byte\_t shift\_byte;

    byte\_t initial\_permutation[8];

    memset(initial\_permutation, 0, 8);

    memset(processed\_block, 0, 8);

    // Initial permutation

    for (i = 0; i < 64; i++)

    {

        shift\_size = \_\_des\_initial\_message\_permutation[i];

        shift\_byte = 0x80 >> ((shift\_size - 1) % 8);

        shift\_byte &= data\_block[(shift\_size - 1) / 8];

        shift\_byte <<= ((shift\_size - 1) % 8);

        initial\_permutation[i / 8] |= (shift\_byte >> i % 8);

    }

    // Split message into two 32-bit pieces

    byte\_t l[4], r[4];

    for (i = 0; i < 4; i++)

    {

        l[i] = initial\_permutation[i];

        r[i] = initial\_permutation[i + 4];

    }

    byte\_t ln[4], rn[4], er[6], ser[4];

    // 16 rounds of Feistel network

    int key\_index;

    for (k = 1; k <= 16; k++)

    {

        memcpy(ln, r, 4);

        memset(er, 0, 6);

        // Expansion permutation (E)

        for (i = 0; i < 48; i++)

        {

            shift\_size = \_\_des\_message\_expansion[i];

            shift\_byte = 0x80 >> ((shift\_size - 1) % 8);

            shift\_byte &= r[(shift\_size - 1) / 8];

            shift\_byte <<= ((shift\_size - 1) % 8);

            er[i / 8] |= (shift\_byte >> i % 8);

        }

        // If decryption mode, use keys in reverse order

        if (mode == DES\_DECRYPTION\_MODE)

        {

            key\_index = 17 - k;

        }

        else

        {

            key\_index = k;

        }

        // XOR with key

        for (i = 0; i < 6; i++)

        {

            er[i] ^= key\_sets[key\_index].k[i];

        }

        byte\_t row, column;

        for (i = 0; i < 4; i++)

        {

            ser[i] = 0;

        }

        // S-Box substitution

        // 0000 0000 0000 0000 0000 0000

        // rccc crrc cccr rccc crrc cccr

        // Byte 1

        row = 0;

        row |= ((er[0] & 0x80) >> 6);

        row |= ((er[0] & 0x04) >> 2);

        column = 0;

        column |= ((er[0] & 0x78) >> 3);

        ser[0] |= ((byte\_t)\_\_des\_S1[row \* 16 + column] << 4);

        row = 0;

        row |= (er[0] & 0x02);

        row |= ((er[1] & 0x10) >> 4);

        column = 0;

        column |= ((er[0] & 0x01) << 3);

        column |= ((er[1] & 0xE0) >> 5);

        ser[0] |= (byte\_t)\_\_des\_S2[row \* 16 + column];

        // Byte 2

        row = 0;

        row |= ((er[1] & 0x08) >> 2);

        row |= ((er[2] & 0x40) >> 6);

        column = 0;

        column |= ((er[1] & 0x07) << 1);

        column |= ((er[2] & 0x80) >> 7);

        ser[1] |= ((byte\_t)\_\_des\_S3[row \* 16 + column] << 4);

        row = 0;

        row |= ((er[2] & 0x20) >> 4);

        row |= (er[2] & 0x01);

        column = 0;

        column |= ((er[2] & 0x1E) >> 1);

        ser[1] |= (byte\_t)\_\_des\_S4[row \* 16 + column];

        // Byte 3

        row = 0;

        row |= ((er[3] & 0x80) >> 6);

        row |= ((er[3] & 0x04) >> 2);

        column = 0;

        column |= ((er[3] & 0x78) >> 3);

        ser[2] |= ((byte\_t)\_\_des\_S5[row \* 16 + column] << 4);

        row = 0;

        row |= (er[3] & 0x02);

        row |= ((er[4] & 0x10) >> 4);

        column = 0;

        column |= ((er[3] & 0x01) << 3);

        column |= ((er[4] & 0xE0) >> 5);

        ser[2] |= (byte\_t)\_\_des\_S6[row \* 16 + column];

        // Byte 4

        row = 0;

        row |= ((er[4] & 0x08) >> 2);

        row |= ((er[5] & 0x40) >> 6);

        column = 0;

        column |= ((er[4] & 0x07) << 1);

        column |= ((er[5] & 0x80) >> 7);

        ser[3] |= ((byte\_t)\_\_des\_S7[row \* 16 + column] << 4);

        row = 0;

        row |= ((er[5] & 0x20) >> 4);

        row |= (er[5] & 0x01);

        column = 0;

        column |= ((er[5] & 0x1E) >> 1);

        ser[3] |= (byte\_t)\_\_des\_S8[row \* 16 + column];

        for (i = 0; i < 4; i++)

        {

            rn[i] = 0;

        }

        // Straight permutation (P)

        for (i = 0; i < 32; i++)

        {

            shift\_size = \_\_des\_right\_sub\_msg\_permut[i];

            shift\_byte = 0x80 >> ((shift\_size - 1) % 8);

            shift\_byte &= ser[(shift\_size - 1) / 8];

            shift\_byte <<= ((shift\_size - 1) % 8);

            rn[i / 8] |= (shift\_byte >> i % 8);

        }

        for (i = 0; i < 4; i++)

        {

            rn[i] ^= l[i];

        }

        for (i = 0; i < 4; i++)

        {

            l[i] = ln[i];

            r[i] = rn[i];

        }

    }

    // Combine R and L, pre-end permutation

    byte\_t pre\_end\_permutation[8];

    for (i = 0; i < 4; i++)

    {

        pre\_end\_permutation[i] = r[i];

        pre\_end\_permutation[4 + i] = l[i];

    }

    for (i = 0; i < 64; i++)

    {

        shift\_size = \_\_des\_final\_msg\_permut[i];

        shift\_byte = 0x80 >> ((shift\_size - 1) % 8);

        shift\_byte &= pre\_end\_permutation[(shift\_size - 1) / 8];

        shift\_byte <<= ((shift\_size - 1) % 8);

        processed\_block[i / 8] |= (shift\_byte >> i % 8);

    }

}

/// @brief Encrypt a 64 bit block of data using DES

/// @param data\_block

/// @param processed\_block

/// @param key\_sets array of 16(+1) key sets

void \_\_des\_encrypt\_block(byte\_t \*data\_block,

                         byte\_t \*processed\_block,

                         des\_key\_set \*key\_sets)

{

    \_\_des\_process\_data\_block(data\_block,

                             processed\_block,

                             key\_sets,

                             DES\_ENCRYPTION\_MODE);

}

/// @brief Decrypt a 64 bit block of data using DES

/// @param data\_block

/// @param processed\_block

/// @param key\_sets array of 16(+1) key sets

void \_\_des\_decrypt\_block(byte\_t \*data\_block,

                         byte\_t \*processed\_block,

                         des\_key\_set \*key\_sets)

{

    \_\_des\_process\_data\_block(data\_block,

                             processed\_block,

                             key\_sets,

                             DES\_DECRYPTION\_MODE);

}

/// @brief Generate a random 64 bit key

/// @param key

void des\_key(byte\_t \*key)

{

    do

    {

        int i;

        for (i = 0; i < 8; i++)

        {

            key[i] = rand() % 255;

        }

    } while (!\_\_des\_is\_key\_acceptable(key));

}

/// @brief Encrypt data using DES

/// @param data

/// @param data\_size

/// @param enc\_data

/// @param enc\_data\_size

/// @param des\_key 64 bit key

void des\_encrypt(byte\_t \*data, size\_t data\_size,

                 byte\_t \*enc\_data, size\_t \*enc\_data\_size,

                 byte\_t \*des\_key)

{

    MASSERT(data != NULL, "data cannot be NULL");

    MASSERT(data\_size > 0, "data\_size must be greater than 0");

    MASSERT(data\_size % 8 == 0, "data\_size must be a multiple of 8");

    MASSERT(enc\_data != NULL, "enc\_data cannot be NULL");

    MASSERT(des\_key != NULL, "des\_key cannot be NULL");

    byte\_t \*data\_block = ALLOC(byte\_t, 8);

    MASSERT(data\_block != NULL, "Memory allocation failed");

    byte\_t \*processed\_block = ALLOC(byte\_t, 8);

    MASSERT(processed\_block != NULL, "Memory allocation failed");

    des\_key\_set \*key\_sets = ALLOC(des\_key\_set, 17);

    MASSERT(key\_sets != NULL, "Memory allocation failed");

    \_\_des\_generate\_sub\_keys(des\_key, key\_sets);

    unsigned long number\_of\_blocks = data\_size / 8 + (data\_size % 8 ? 1 : 0);

    for (unsigned long block\_count = 0;

         block\_count < number\_of\_blocks;

         block\_count++)

    {

        for (int i = 0; i < 8; i++)

        {

            data\_block[i] = data[block\_count \* 8 + i];

        }

        \_\_des\_encrypt\_block(data\_block, processed\_block, key\_sets);

        for (int i = 0; i < 8; i++)

        {

            enc\_data[block\_count \* 8 + i] = processed\_block[i];

        }

    }

    \*enc\_data\_size = data\_size;

    FREE(data\_block);

    FREE(processed\_block);

    FREE(key\_sets);

}

/// @brief Decrypt data using DES

/// @param data

/// @param data\_size

/// @param dec\_data

/// @param dec\_data\_size

/// @param des\_key 64 bit key

void des\_decrypt(byte\_t \*data, size\_t data\_size,

                 byte\_t \*dec\_data, size\_t \*dec\_data\_size,

                 byte\_t \*des\_key)

{

    MASSERT(data != NULL, "data cannot be NULL");

    MASSERT(data\_size > 0, "data\_size must be greater than 0");

    MASSERT(data\_size % 8 == 0, "data\_size must be a multiple of 8");

    MASSERT(dec\_data != NULL, "dec\_data cannot be NULL");

    MASSERT(des\_key != NULL, "des\_key cannot be NULL");

    byte\_t \*data\_block = ALLOC(byte\_t, 8);

    MASSERT(data\_block != NULL, "Memory allocation failed");

    byte\_t \*processed\_block = ALLOC(byte\_t, 8);

    MASSERT(processed\_block != NULL, "Memory allocation failed");

    des\_key\_set \*key\_sets = ALLOC(des\_key\_set, 17);

    MASSERT(key\_sets != NULL, "Memory allocation failed");

    \_\_des\_generate\_sub\_keys(des\_key, key\_sets);

    unsigned long number\_of\_blocks = data\_size / 8 + (data\_size % 8 ? 1 : 0);

    for (unsigned long block\_count = 0;

         block\_count < number\_of\_blocks;

         block\_count++)

    {

        for (int i = 0; i < 8; i++)

        {

            data\_block[i] = data[block\_count \* 8 + i];

        }

        \_\_des\_decrypt\_block(data\_block, processed\_block, key\_sets);

        for (int i = 0; i < 8; i++)

        {

            dec\_data[block\_count \* 8 + i] = processed\_block[i];

        }

    }

    \*dec\_data\_size = data\_size;

    FREE(data\_block);

    FREE(processed\_block);

    FREE(key\_sets);

}

Пример работы программы:

Изображение выглядит как текст, снимок экрана, мультимедиа, программное обеспечение

Автоматически созданное описание

Рисунок 21. Генерация ключа DES.

Изображение выглядит как электроника, текст, снимок экрана, программное обеспечение

Автоматически созданное описание

Рисунок 22. Шифрование DES.

Изображение выглядит как электроника, текст, снимок экрана, программное обеспечение

Автоматически созданное описание

Рисунок 23. Расшифрование DES.

2.4. Алгоритм перемешивания

Для повышения криптостойкости, данные для шифрования можно перемешать. Например, следующим образом:

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Автоматически созданное описание

Рисунок 24. Схема алгоритма перемешивания.

Исходный код реализации алгоритма:

С++

/// @brief Shuffle data for encryption

/// @param data

/// @param data\_size

void byte\_shuffle(byte\_t \*data, size\_t data\_size)

{

    // b = (b+a) % 0xFF

    byte\_t b = 0x00;

    for (size\_t i = 0; i < data\_size; i++)

    {

        b = (b + data[i]) % 0xFF;

        data[i] = b;

    }

}

Если при шифровании было применено перемешивание, то при расшифровании необходимо выполнить этот алгоритм в обратном порядке.

Исходный код реализации обратного алгоритма:

С++

/// @brief Unshuffle data after decryption

/// @param data

/// @param data\_size

void byte\_unshuffle(byte\_t \*data, size\_t data\_size)

{

    byte\_t b = 0x00;

    for (size\_t i = 0; i < data\_size; i++)

    {

        byte\_t current = data[i];

        data[i] = (data[i] - b + 0xFF) % 0xFF;

        b = current;

    }

}

Приложения

Приложение 1. Ссылка на исходный код программы.

<https://github.com/Kseen715/cifs.cpp>