


**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
1. Теоритические сведения	4
1.1. Алгоритм RSA	4
1.2. Алгоритм Эль-Гамала	6
1.2.1. Подпись Эль-Гамала.....	7
1.3. Сеть Фейстеля	10
1.4. Алгоритм DES	10
2. Практическая часть	14
2.1. Реализация алгоритма RSA.....	14
2.2. Реализация алгоритма Эль-Гамала.....	17
2.2.1. Реализация алгоритма подписи Эль-Гамала	20
2.3. Реализация алгоритма DES.....	23
2.4. Алгоритм перемешивания.....	36
3. Разработка собственного алгоритма шифрования	37
3.1. Модификация DES. DES-Perlín-64.....	37
Приложения.....	59

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 p и q .
- 2) Вычисляется их произведение $n = p \cdot q$, которое называется модулем.
- 3) Вычисляется значение функции Эйлера: $\phi(n) = (p - 1) \cdot (q - 1)$.
- 4) Выбирается целое число e , которое меньше $\phi(n)$ и взаимно простое с $\phi(n)$.
- 5) Вычисляется число d , которое является мультипликативно обратным к e по модулю $\phi(n)$, $d \cdot e \equiv 1 \pmod{\phi(n)}$.

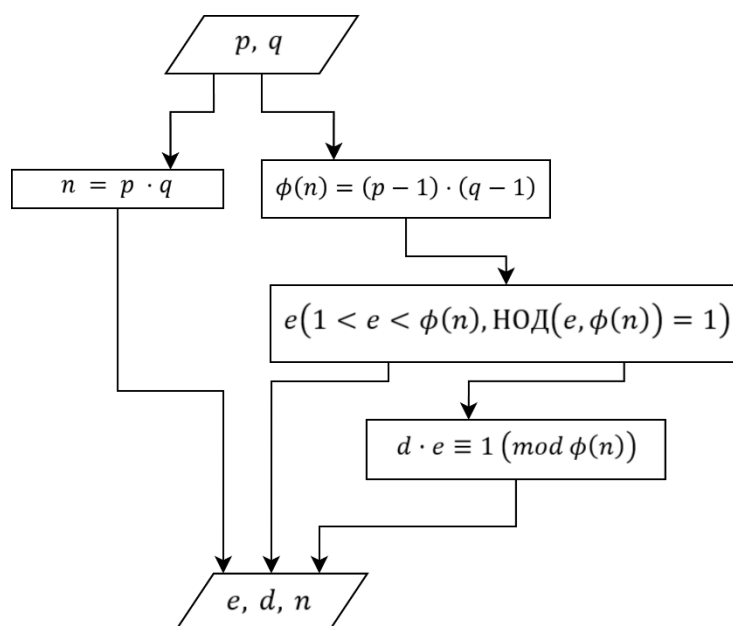


Рисунок 1. Схема создания ключей RSA.

Шифрование:

- 1) Открытый ключ состоит из пары чисел (e, n) .
- 2) Чтобы зашифровать сообщение m , вычисляется $c = m^e \pmod n$, где c — это зашифрованный текст.

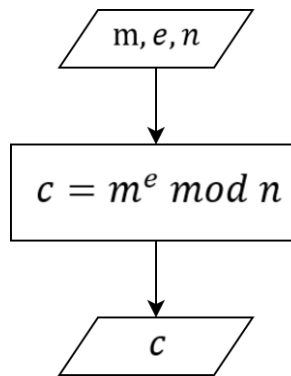


Рисунок 2. Схема шифрования RSA.

Расшифрование:

- 1) Секретный ключ состоит из пары чисел (d, n) .
- 2) Чтобы расшифровать зашифрованный текст c , вычисляется $m = c^d \bmod n$, где m — это исходное сообщение.

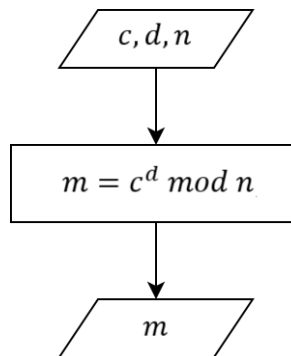


Рисунок 3. Схема расшифрования RSA.

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-Гамала

Алгоритм Эль-Гамала — 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 p .
- 2) Выбирается целое число g — первообразный корень по модулю p .
- 3) Выбирается случайное целое число x , такое что $1 < x < p - 1$.
- 4) Вычисляется $y = g^x \bmod p$.
- 5) Открытый ключ состоит из тройки чисел (y, g, p) , а закрытый ключ — из числа x .

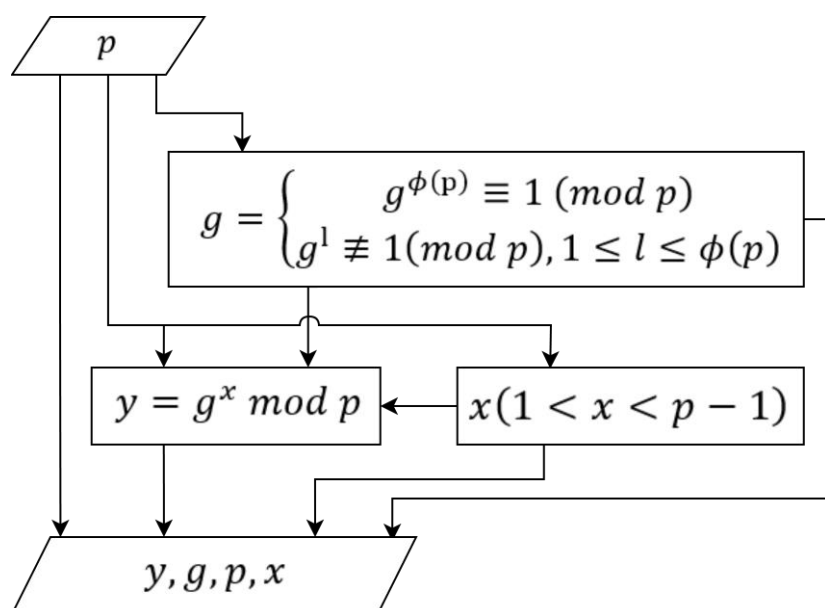


Рисунок 4. Схема генерации ключей для алгоритма Эль-Гамала.

Шифрование:

- 1) Сообщение m должно быть меньше числа p .
- 2) Выбирается сессионный ключ — случайное целое число k , такое что $1 < k < p - 1$.
- 3) Вычисляются числа $a = g^k \bmod p$ и $b = y^k m \bmod p$.
- 4) Пара чисел (a, b) является шифротекстом.

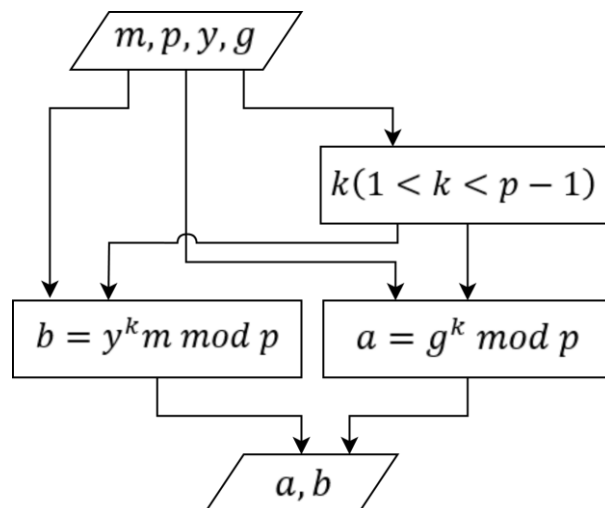


Рисунок 5. Схема шифрования алгоритмом Эль-Гамала.

Расшифрование:

Зная закрытый ключ x , исходное сообщение m можно вычислить из шифротекста (a, b) по формуле $m = b \cdot a^{-x} \bmod p$.

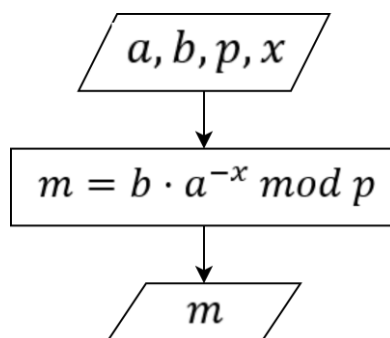


Рисунок 6. Схема дешифрования алгоритмом эль-Гамала.

Особенности и безопасность:

- Безопасность алгоритма основана на сложности задачи вычисления дискретных логарифмов.
- Алгоритм Эль-Гамала 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-Гамала

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

Генерация ключей:

1. Генерируется случайное простое число p .
2. Выбирается целое число g — первообразный корень по модулю p .
3. Выбирается случайное целое число x , такое что $1 < x < p - 1$.
4. Вычисляется $y = g^x \bmod p$.
5. Открытый ключ состоит из тройки чисел (y, g, p) , а закрытый ключ — из числа x .

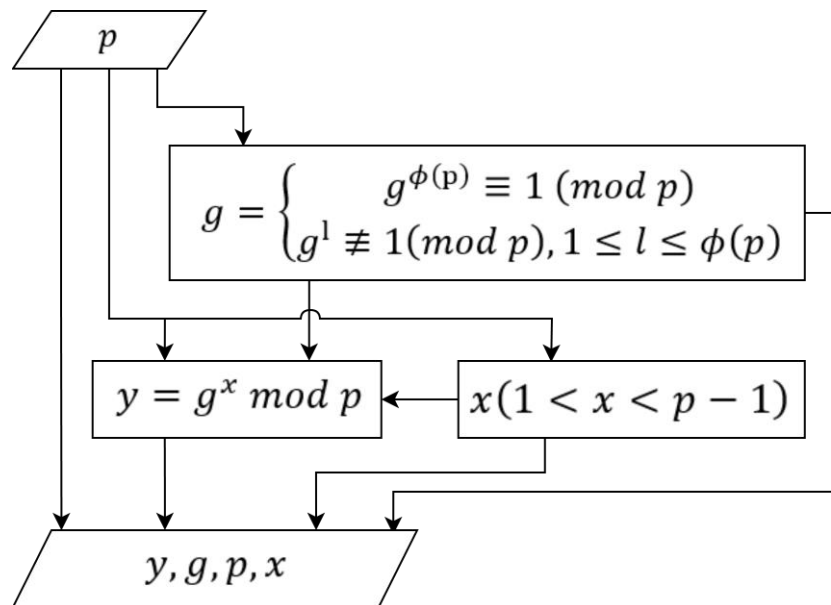


Рисунок 7. Схема генерации ключей для подписи алгоритмом Эль-Гамала.

Создание подписи:

1. Выбирается случайное число k , взаимно простое с $p - 1$.
2. Вычисляется $r = g^k \bmod p$.
3. Вычисляется $s = (H(m) - x \cdot r) \cdot k^{-1} \bmod (p - 1)$, где $H(m)$ — это хеш-значение сообщения m , а k^{-1} — мультипликативно обратное k по модулю $p - 1$.
4. Подпись сообщения m состоит из пары чисел (r, s) .

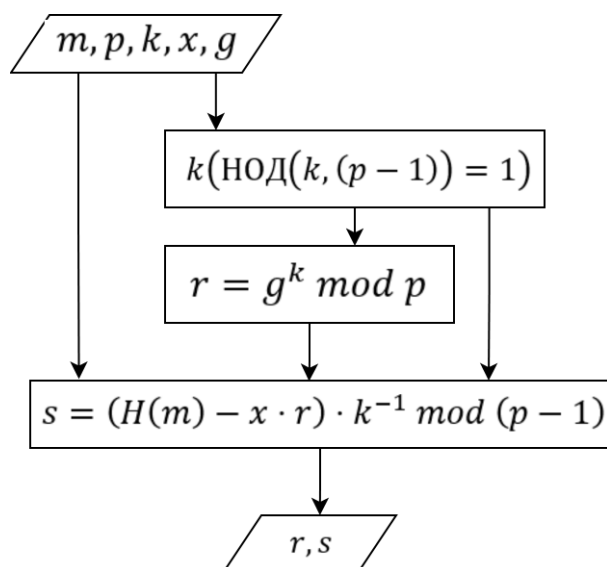


Рисунок 8. Схема подписи алгоритмом Эль-Гамала.

Проверка подписи:

1. Вычисляется $u_1 = H(m) \cdot s^{-1} \bmod (p - 1)$.
2. Вычисляется $u_2 = r \cdot s^{-1} \bmod (p - 1)$.
3. Вычисляется $v = (g^{u_1} \cdot y^{u_2}) \bmod p$.
4. Если $v = r$, то подпись считается действительной.

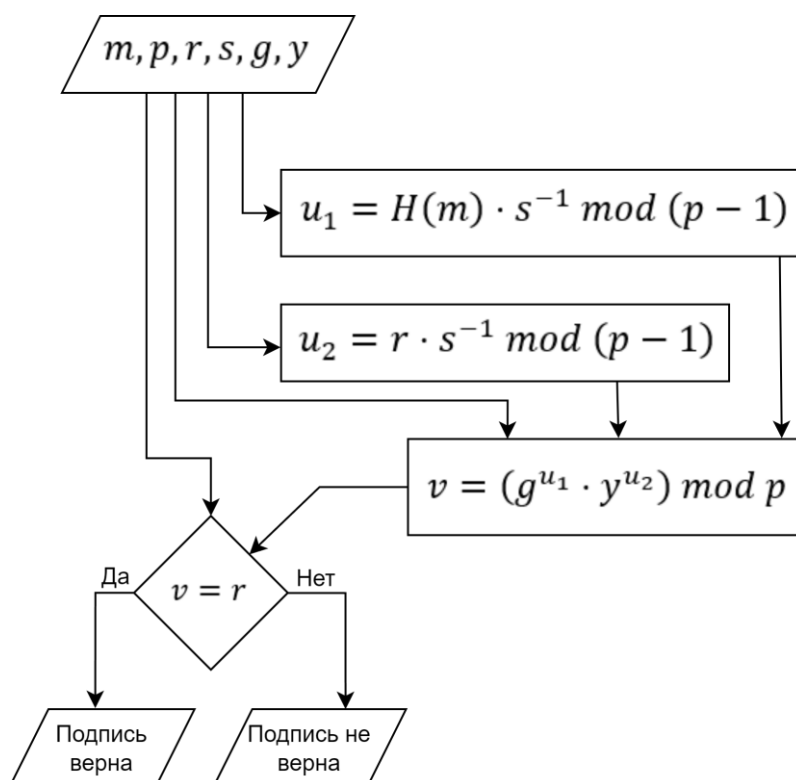


Рисунок 9. Схема проверки подписи алгоритмом Эль-Гамала.

Безопасность алгоритма:

Безопасность алгоритма подписи Эль-Гамала 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-Гамала 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 L_0 и правую R_0 .

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

- Выбирается подключ K_i для текущего раунда.
- Применяется функция F к правой половине R_{i-1} и подключу K_i , результат обозначается как $F(R_{i-1}, K_i)$.
- Результат функции F складывается по модулю 2 (XOR) с левой половиной L_{i-1} , получается новая правая половина R_i .
- Правая половина R_{i-1} становится новой левой половиной L_i .

3. Выходные данные получаются путём объединения последних половин R_n и L_n после n 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.

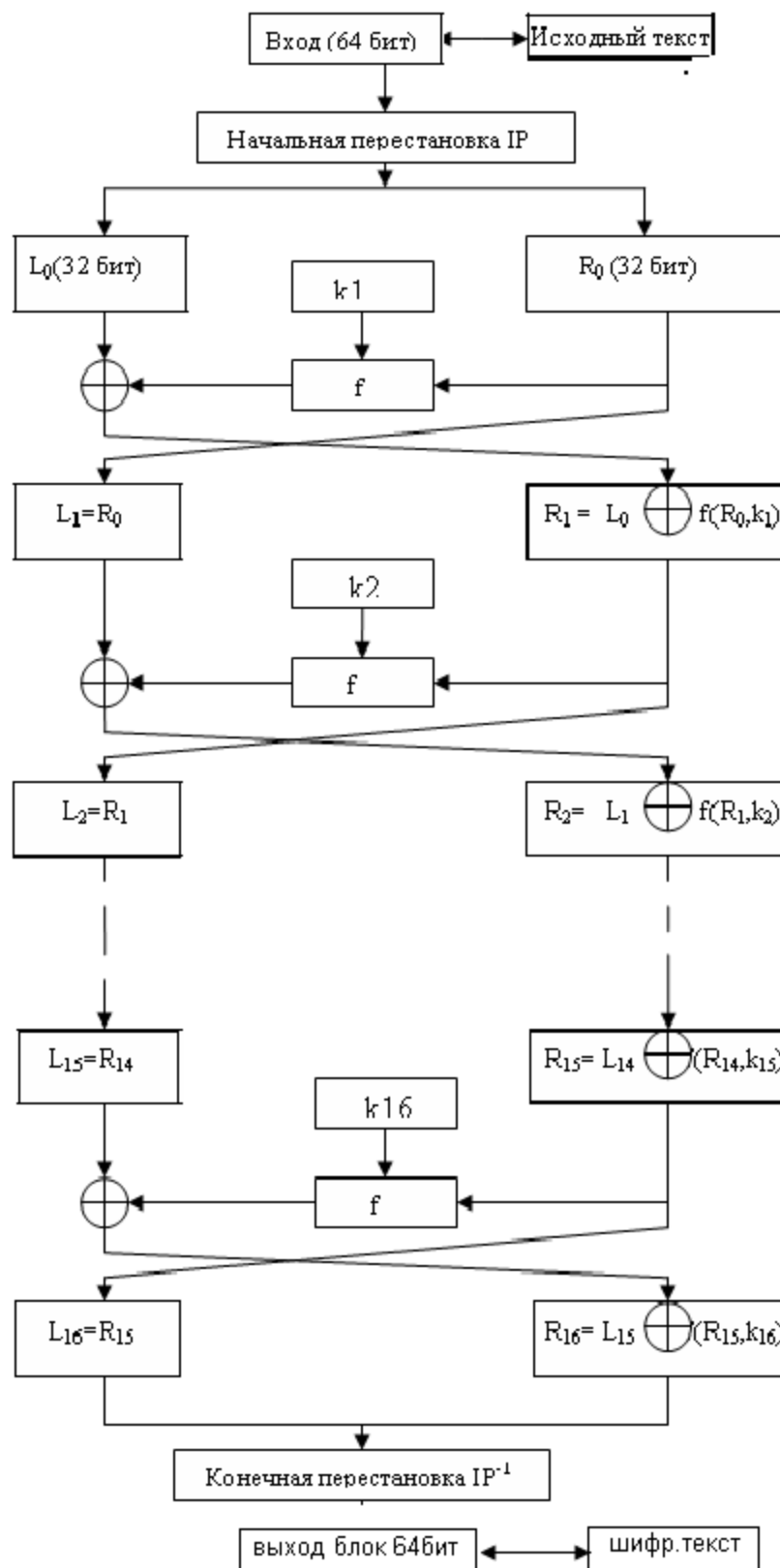


Рисунок 10. Схема шифрования алгоритма DES.

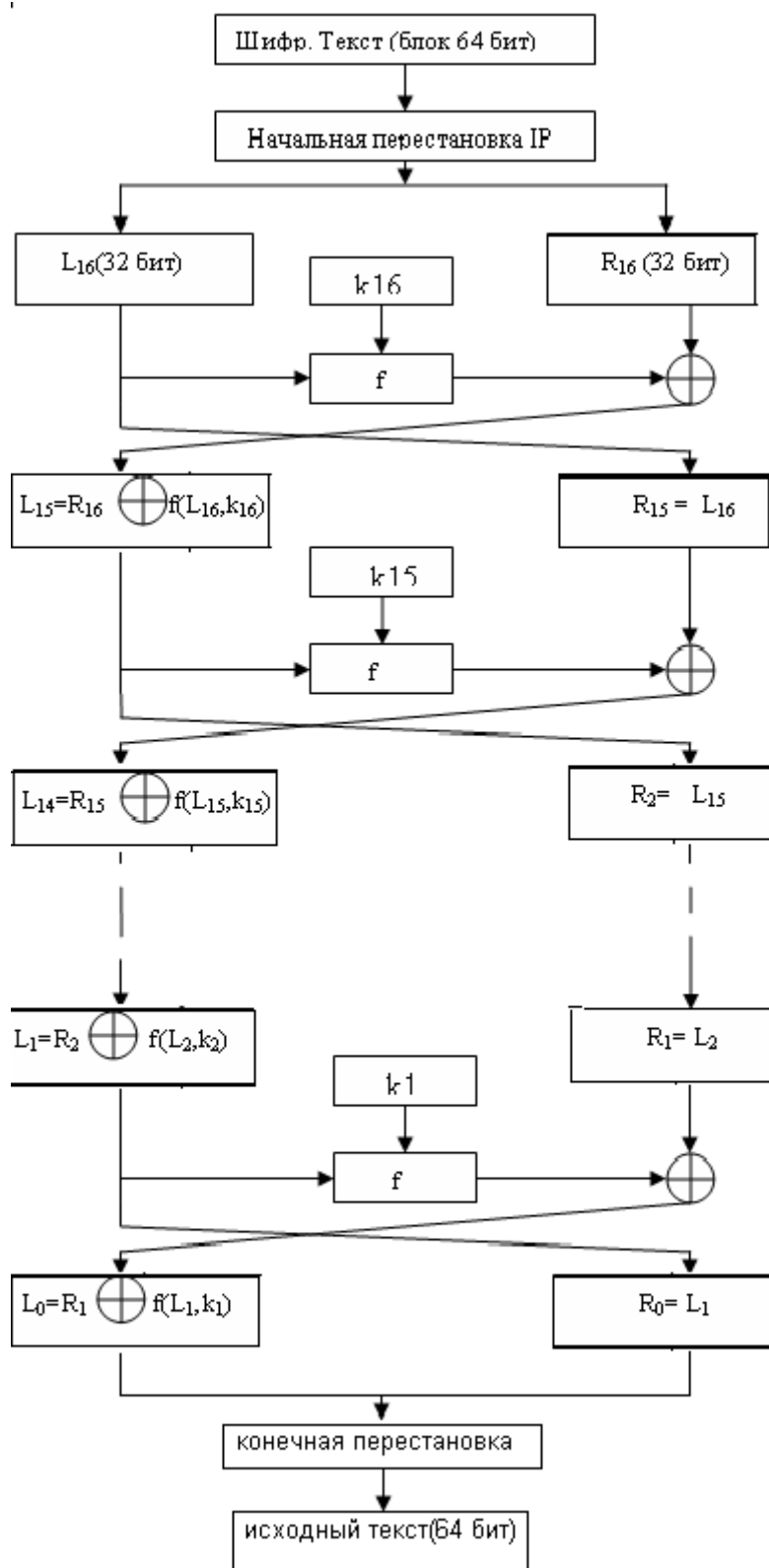


Рисунок 11. Схема расшифрования алгоритма DES.

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()];
}

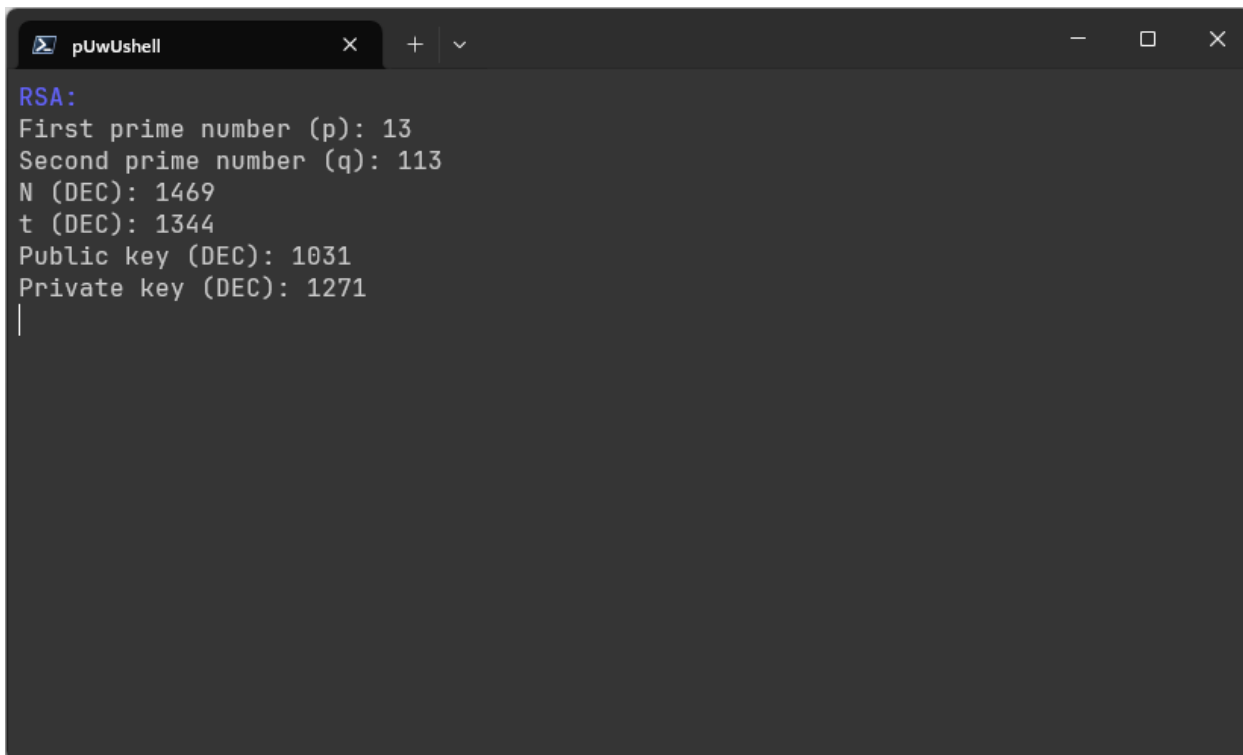
/// @brief Generate RSA decryption key
/// @param cif_key RSA encryption key
/// @param t RSA t parameter
/// @return RSA decryption key
uint_t rsa_private_key(uint_t cif_key, uint_t t)
{
    for (uint_t i = 1; i < t; i++)
    {
        if ((cif_key * i) % t == 1)
        {
            return i;
        }
    }
    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);
}

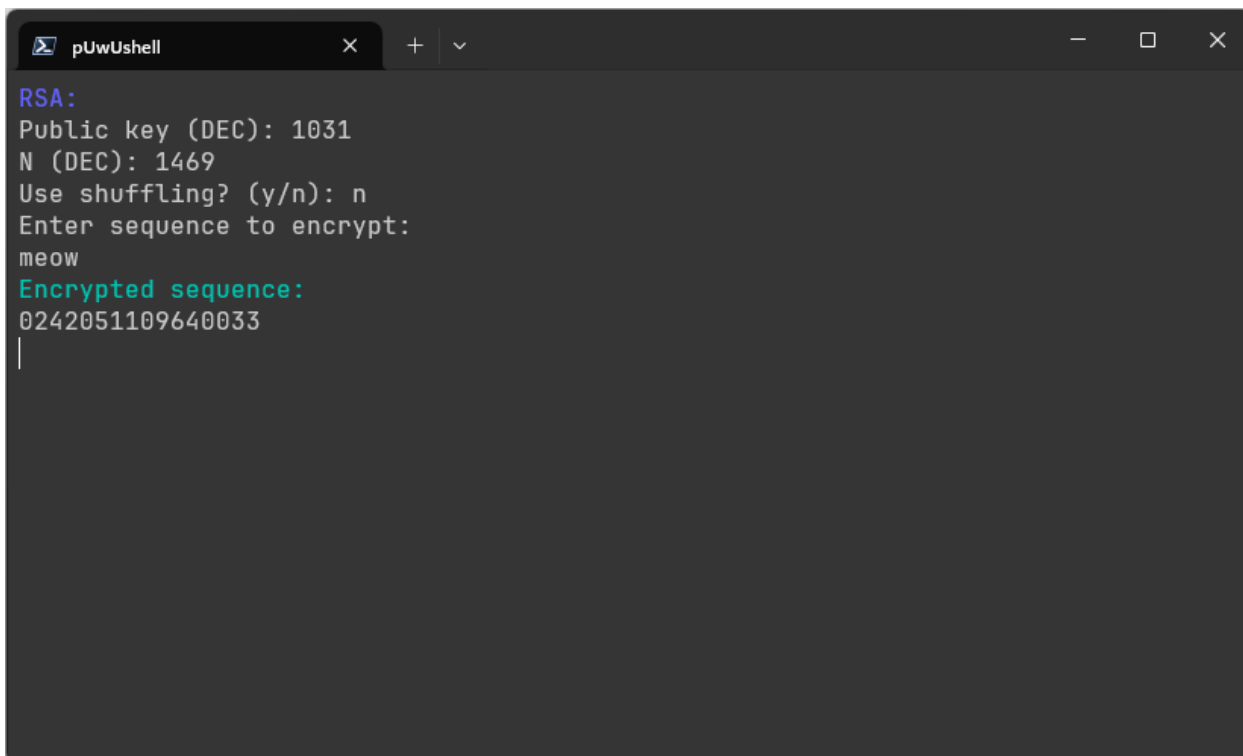
```

Пример работы программы:



```
pUwUshell
RSA:
First prime number (p): 13
Second prime number (q): 113
N (DEC): 1469
t (DEC): 1344
Public key (DEC): 1031
Private key (DEC): 1271
|
```

Рисунок 12. Создание ключей RSA.



```
pUwUshell
RSA:
Public key (DEC): 1031
N (DEC): 1469
Use shuffling? (y/n): n
Enter sequence to encrypt:
meow
Encrypted sequence:
0242051109640033
|
```

Рисунок 13. Шифрование алгоритмом RSA.


```
pUwUshell
RSA:
Private key (DEC): 1271
N (DEC): 1469
Use shuffling? (y/n): n
Enter sequence to decrypt:
0242051109640033
Decrypted sequence (DEC):
(4){109, 101, 111, 119}
Decrypted sequence (ASCII):
meow
|
```

Рисунок 14. Расшифрование алгоритмом RSA.

2.2. Реализация алгоритма Эль-Гамала

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 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

```

```

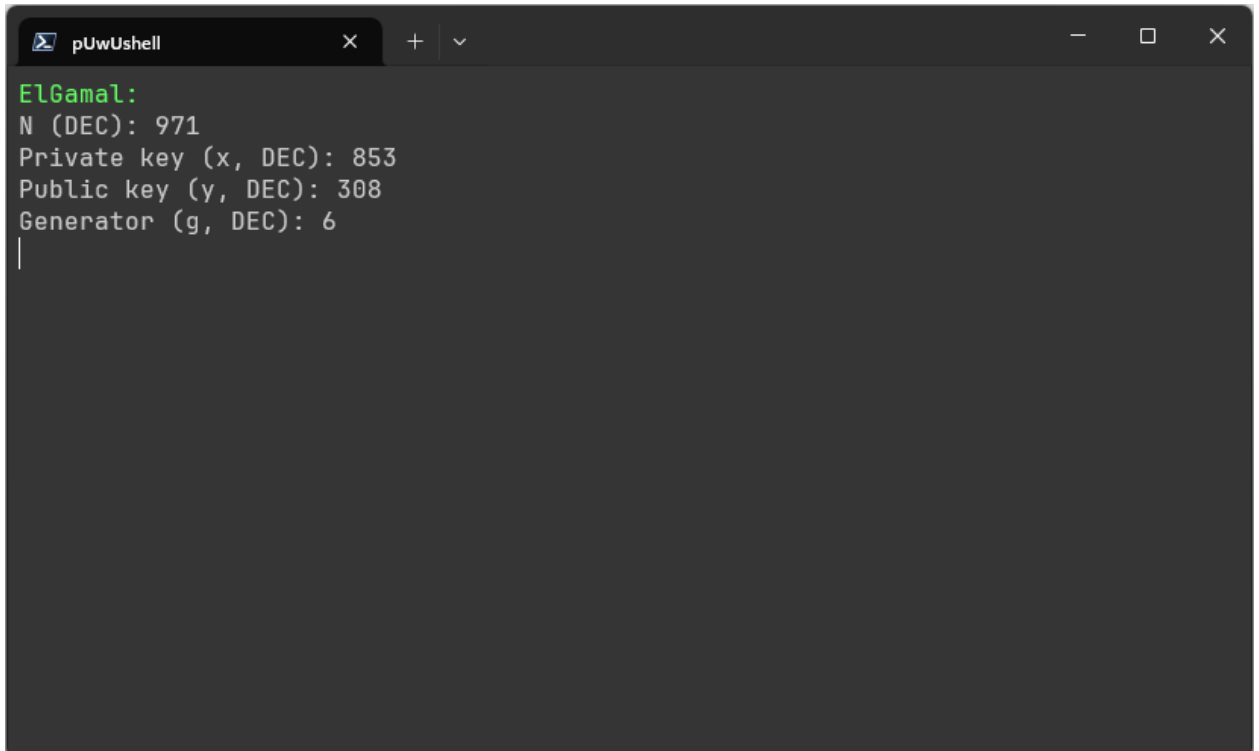
/// @param x ElGamal session key
/// @param p modulo
void elg_public_key(uint_t *key_y, uint_t *key_g, uint_t x, uint_t p)
{
    *key_g = primitive_root_mod(p);
    *key_y = __elg_y(*key_g, x, p);
}

/// @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:



```

pUwUshell
ElGamal:
N (DEC): 971
Private key (x, DEC): 853
Public key (y, DEC): 308
Generator (g, DEC): 6
|

```

Рисунок 15. Генерация ключей для алгоритма Эль-Гамала.

```
pUwUshell
ElGamal:
Public key (y, DEC): 735
Generator (g, DEC): 6
N (DEC): 971
Use shuffling? (y/n): n
Enter sequence to encrypt:
cat
Encrypted sequence:
368222189309263176
|
```

Рисунок 16. Шифрование алгоритмом Эль-Гамала.

```
pUwUshell
ElGamal:
Private key (x, DEC): 318
N (DEC): 971
Use shuffling? (y/n): n
Enter sequence to decrypt:
368222189309263176
Decrypted sequence (DEC):
(3){99, 97, 116}
Decrypted sequence (ASCII):
cat
|
```

Рисунок 17. Расшифрование алгоритмом Эль-Гамала.

2.2.1. Реализация алгоритма подписи Эль-Гамала

Исходный код реализации:

```
C++
/// @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:

```

pUwUshell
ElGamal Signature:
N (DEC): 257
Private key (x, DEC): 92
Public key (y, DEC): 146
Generator (g, DEC): 3
|

```

Рисунок 18. Генерация ключей для подписи с использованием алгоритма Эль-Гамала.

```
pUwUshell
ElGamal Signature:
Private key (x, DEC): 92
Generator (g, DEC): 3
N (DEC): 257
Enter sequence to sign:
cat
Signature:
027229027143027064
|
```

Рисунок 19. Создание подписи алгоритмом Эль-Гамала.

```
pUwUshell
ElGamal Signature:
Public key (y, DEC): 146
Generator (g, DEC): 3
N (DEC): 257
Enter sequence to check:
cat
Enter signature:
027229027143027064
Signature is valid
|
```

Рисунок 20. Проверка подписи алгоритмом Эль-Гамала.

2.3. Реализация алгоритма DES

Исходный код реализации:

```
C++
#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_permutation[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] = rn[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];
        }

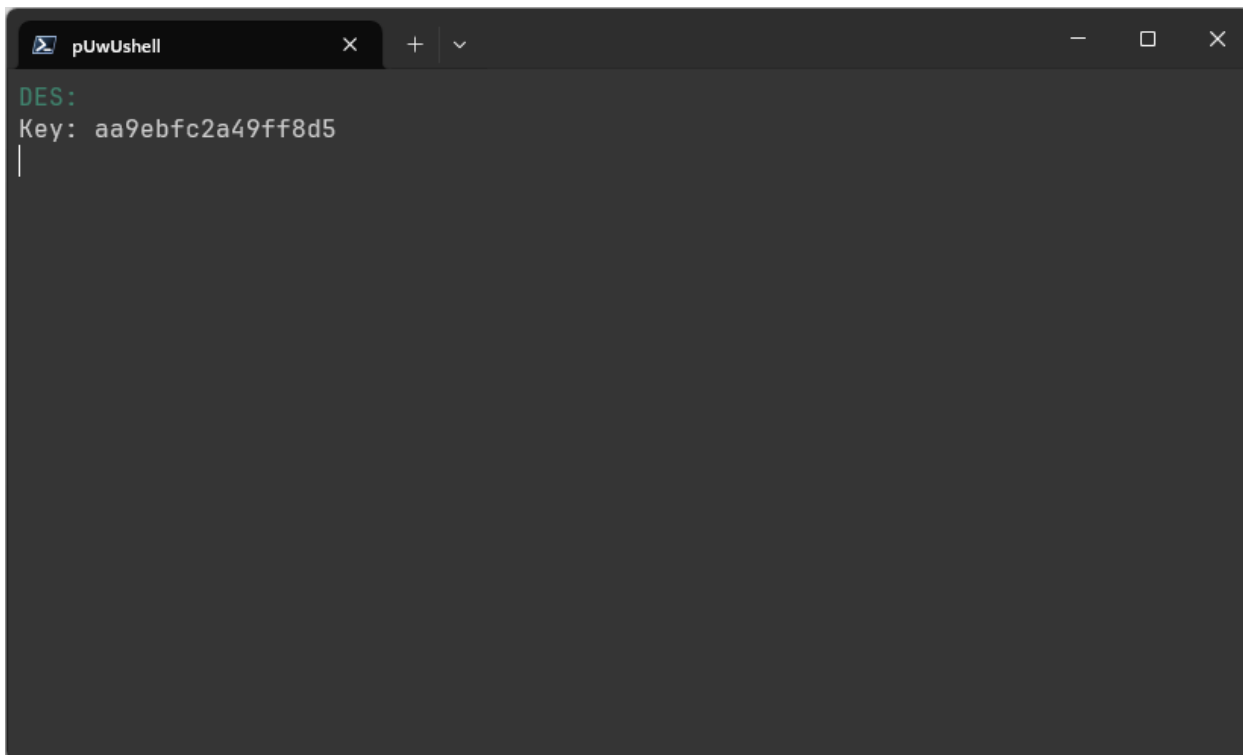
        __des_decrypt_block(data_block, processed_block, key_sets);

        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);
}

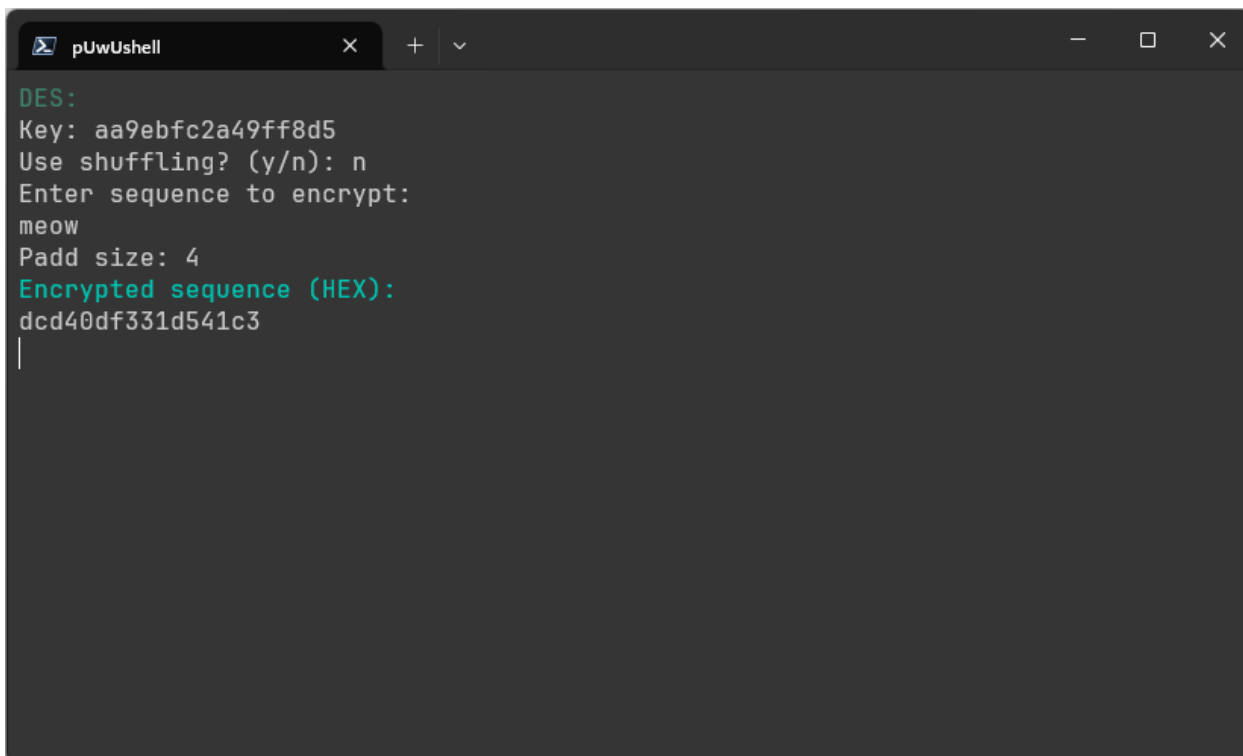
```

Пример работы программы:



```
pUwUshell
DES:
Key: aa9ebfc2a49ff8d5
|
```

Рисунок 21. Генерация ключа DES.



```
pUwUshell
DES:
Key: aa9ebfc2a49ff8d5
Use shuffling? (y/n): n
Enter sequence to encrypt:
meow
Padd size: 4
Encrypted sequence (HEX):
dcd40df331d541c3
|
```

Рисунок 22. Шифрование DES.

```

pUwUshell
DES:
Key: aa9ebfc2a49ff8d5
Use shuffling? (y/n): n
Enter sequence to decrypt (HEX):
dcd40df331d541c3
Decrypted sequence (HEX):
6d656f7700000000
Decrypted sequence (ASCII):
meow
|

```

Рисунок 23. Расшифрование DES.

2.4. Алгоритм перемешивания

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

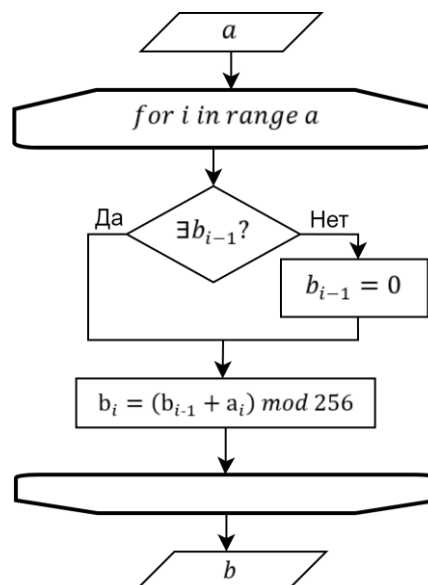


Рисунок 24. Схема алгоритма перемешивания.

Исходный код реализации алгоритма:

```

C++
/// @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:

```

C++
/// @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;
    }
}

```

3. Разработка собственного алгоритма шифрования

3.1. Модификация DES. DES-Perlin-64

Модифицируем классический DES так, чтобы шифрование больше зависело от ключа. Для этого можно каким-то образом генерировать таблицы перестановок. Будем использовать шум Перлина и генерировать двумерный шум, а затем выбирать из него нужные нам данные.

В нашем случае возникает проблема – ключ для шума Перлина имеет длину в 4096 бит, а DES – только 64 бит. Нам требуется его расширить. Для будем использовать следующий алгоритм:

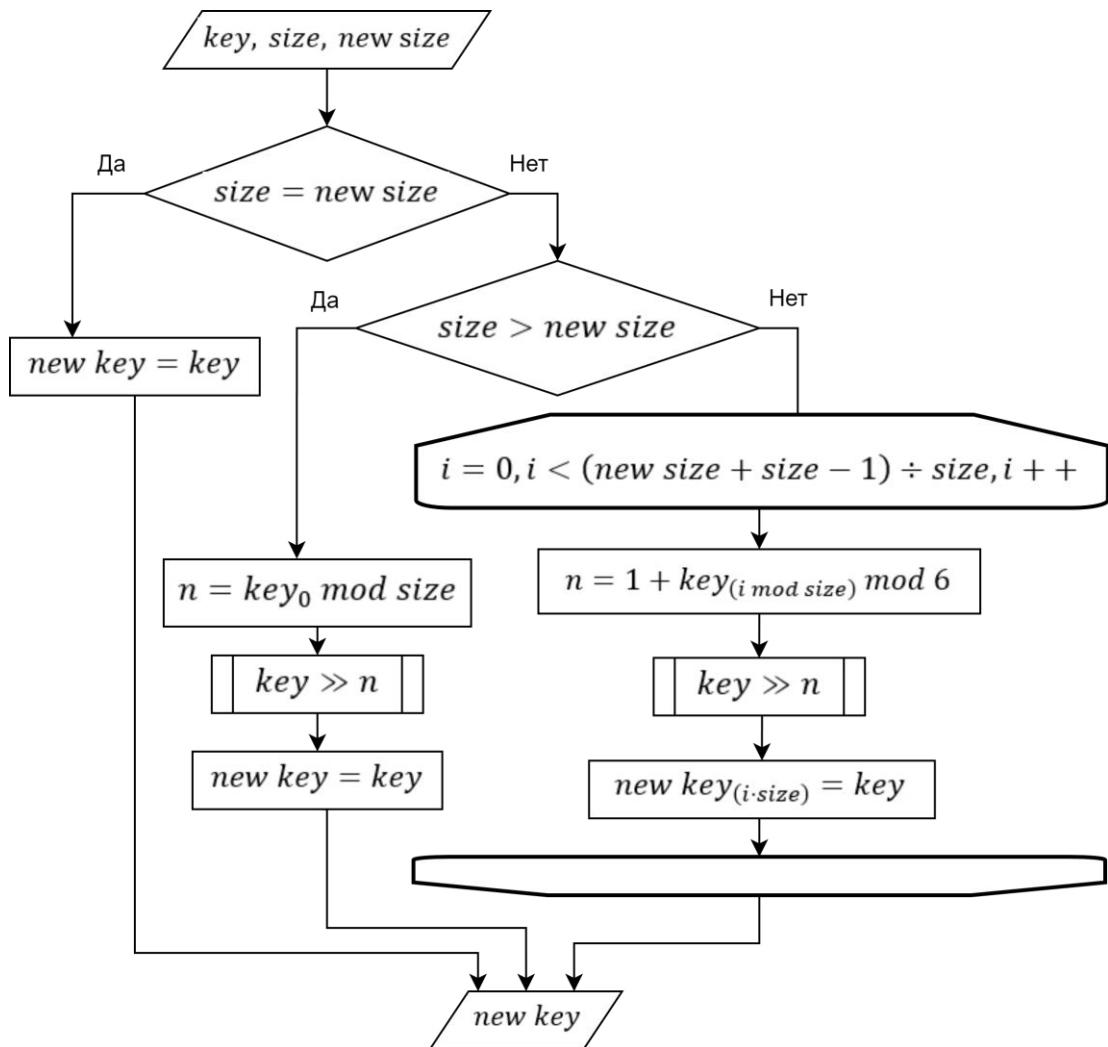


Рисунок 25. Схема расширения ключа.

Исходный код реализации алгоритма:

```

C++
/// @brief Rotate full array to the right on n bits.
/// @param data
/// @param size
/// @param n
void ror_bit_array(byte_t *data, size_t size, size_t n)
{
    n = n % (size * 8); // Ensure n is within the bit size of the array
    if (n == 0)
        return; // If no rotation is needed, return

    bool carry1 = false;
    bool carry2 = false;
    for (size_t i = 0; i < n; i++)
    {
        for (size_t j = 0; j < size; j++)
        {
            carry1 = data[j] & 0x01;
            data[j] >>= 1;
            if (carry2)
                data[j] |= 0b10000000;
            carry2 = carry1;
        }
        if (carry2)
            data[0] |= 0b10000000;
    }
}

```

```

        carry1 = false;
        carry2 = false;
    }
}

/// @brief Resize key in a smart way.
/// If key is bigger than new_key_size, it will be ror'd on n's byte and then
/// cutted. If key is smaller than new_key_size, it will be filled with ror'd
/// on n's byte key.
/// @param input_key
/// @param input_key_size
/// @param new_key
/// @param new_key_size
void resize_key(byte_t *input_key, size_t input_key_size,
                byte_t *new_key, size_t new_key_size)
{
    byte_t *input_copy = ALLOC(byte_t, input_key_size);
    MASSERT(input_copy != NULL, "Memory allocation error");
    memcpy(input_copy, input_key, input_key_size);

    if (input_key_size == new_key_size)
    {
        memcpy(new_key, input_copy, new_key_size);
    }
    else if (input_key_size > new_key_size)
    {
        size_t n = input_copy[0] % input_key_size;
        ror_bit_array(input_copy, input_key_size, n);
        memcpy(new_key, input_copy, new_key_size);
    }
    else
    {
        // for every full input key repetition, ror it on i's byte of input key
        for (size_t i = 0; i < (new_key_size + input_key_size - 1) /
                                input_key_size;
            i++)
        {
            size_t n = (input_key[i % input_key_size] % 6) + 1;
            ror_bit_array(input_copy, input_key_size, n);
            memcpy(new_key + i * input_key_size, input_copy, input_key_size);
        }
    }
    FREE(input_copy);
}

```

Так как мы расширяем ключ в 64 раза, имея только 64 бита исходной информации, необходимо исследовать, не будут ли созданы нежелательные закономерности.

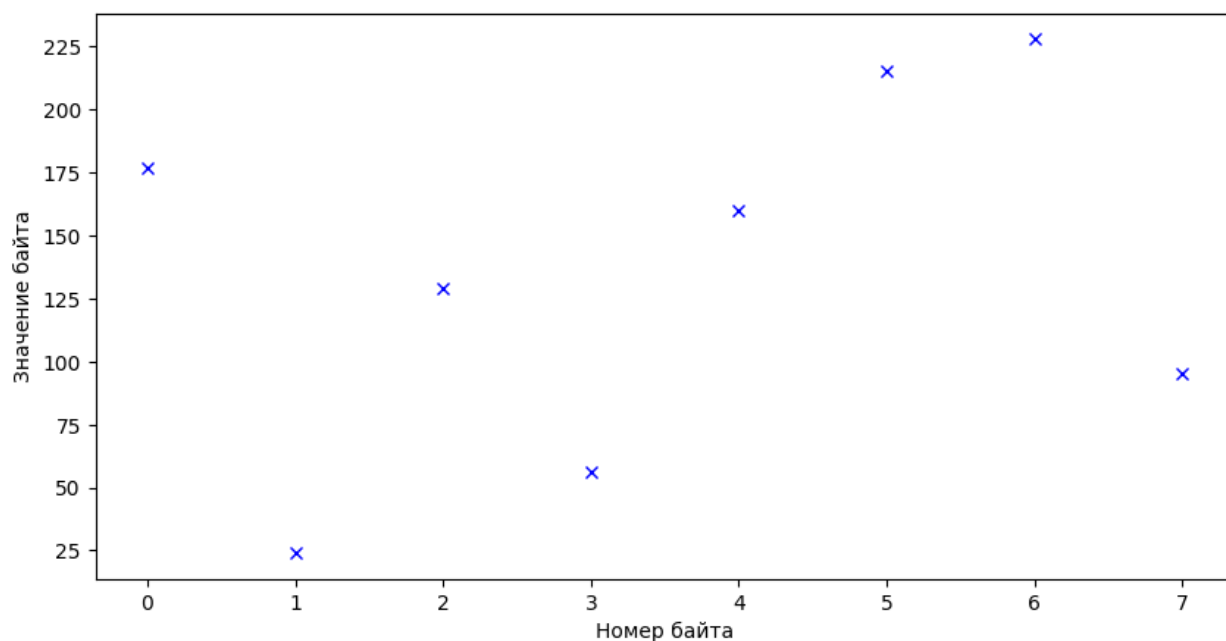


Рисунок 26. Исходные значения ключа.

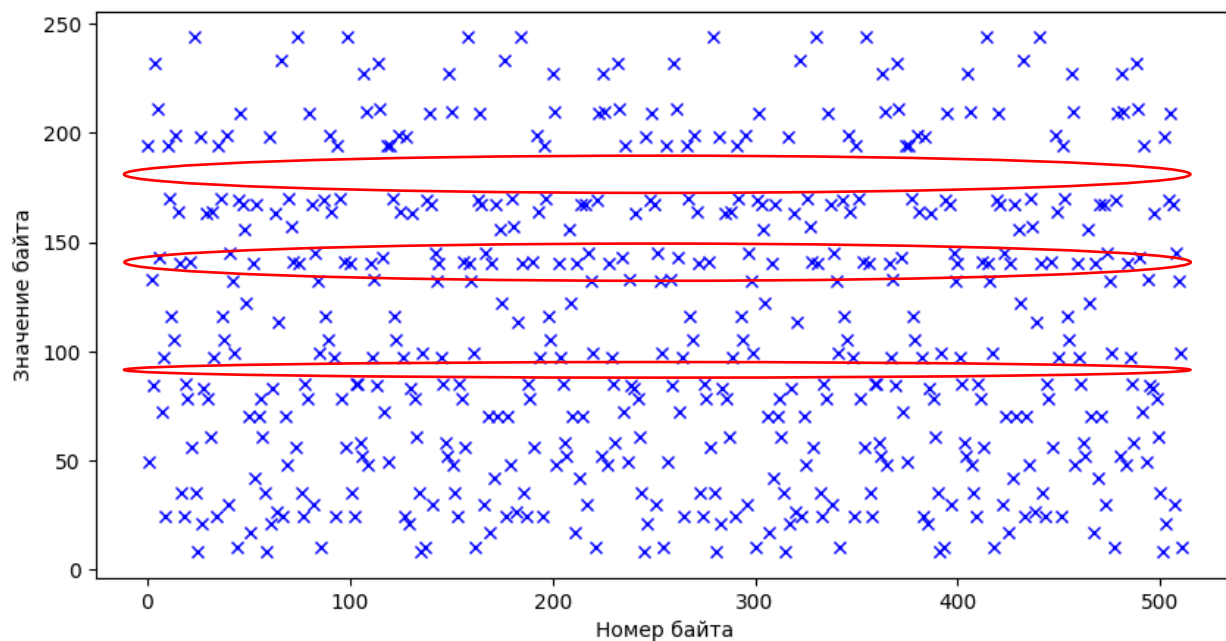


Рисунок 27. Значения ключа после расширения.

На графике видно, что образовалось три подозрительные области: в первой и третьей – значения отсутствуют полностью, во второй – их концентрация значительно повышена. Следовательно, возникли закономерности, которые можно будет отследить при атаке на шифрование. В будущем следует рассмотреть использование более длинного исходного ключа.

Реализуем шум Перлина.

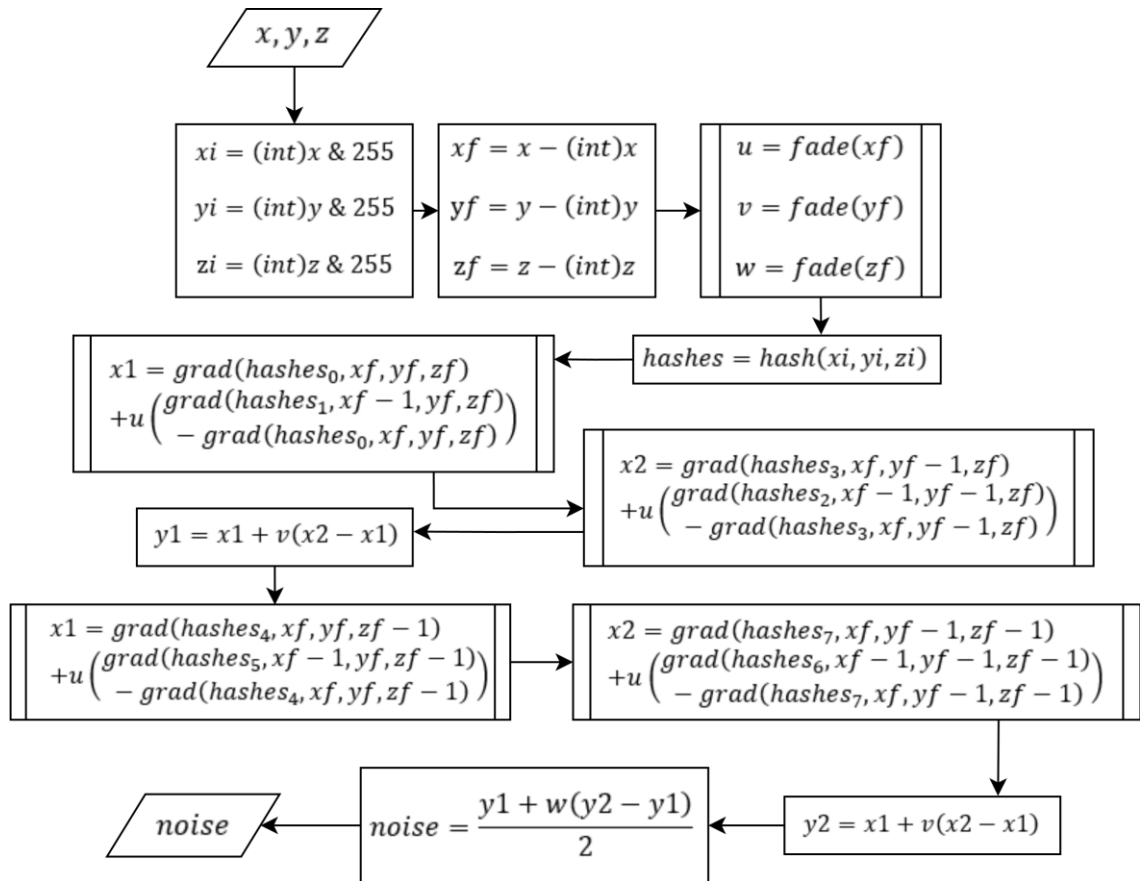


Рисунок 28. Схема алгоритма шума Перлина.

Исходный код реализации алгоритма генерации шума Перлина:

```

C++
/// @brief Linearly interpolate between the lo and hi values,
/// priority given by the param t
/// @param lo lower bound
/// @param hi upper bound
/// @param t priority value
/// @return float interpolated value
float __lerp(float lo, float hi, float t)
{
    return lo + t * (hi - lo);
}

/// @brief Calculate the dot product between the random chosen gradient vector
/// and the distance vector
/// @param hash The hash value to choose the gradient vector
/// @param x_comp x component of the distance vector
/// @param y_comp y component of the distance vector
/// @param z_comp z component of the distance vector
/// @return float dot product value
float __pn_grad(int hash, float x_comp, float y_comp, float z_comp)
{
    // use the first 4 bits of the hash to generate 12 random vectors
    // and "dot" them with (x_comp, y_comp, z_comp)

    int h = hash & 0xF;
    float w = h < 8 /* 0b1000 */
        ? x_comp
        : y_comp;

```

```

float t = h < 4 /* 0b100 */
        ? y_comp
        : (h == 12 || h == 14
           ? x_comp
           : z_comp);

// from the first two bits decide if w or t are positive or negative
return ((h & 1) == 0 ? w : -w) + ((h & 2) == 0 ? t : -t);
}

/// @brief Fade function to smooth the interpolation, which has slope of zero
/// as it reaches the extremes 0 or 1. This is for the smoothness in the noise
/// value while interpolating
/// @param tf The input value to fade
/// @return float The faded value
float __pn_fadefunc(float tf)
{
    return tf * tf * tf * (tf * (6 * tf - 15) + 10);
}

/// @brief Increment the number and wrap around the repetition amount
/// @param num The number to increment
/// @return int The incremented number
int __pn_rep_inc(int num)
{
    num++;
    num = __pn_rep_amt > 0 ? num % __pn_rep_amt : num;
    return num;
}

/// @brief Calculate the hashes of all the unit cell co-ords
/// @param xi The x co-ordinate of the unit cell
/// @param yi The y co-ordinate of the unit cell
/// @param zi The z co-ordinate of the unit cell
/// @return int* Stores the hashes into the hash_arr and returns a pointer
/// to the array
int *__pn_hash(int xi, int yi, int zi)
{
    int *hash_arr = ALLOC(int, 8);
    /*
     > There will be 8 hashes for each cell point.
     Here's the mapping:

     [0] : "aaa"
     [1] : "baa"
     [2] : "bba"
     [3] : "aba"
     [4] : "aab"
     [5] : "bab"
     [6] : "bbb"
     [7] : "abb"
    */
    hash_arr[0] /*aaa*/ = __pn_P[__pn_P[__pn_P[xi] + yi] + zi];
    hash_arr[1] /*baa*/ = __pn_P[__pn_P[__pn_P[__pn_rep_inc(xi)] + yi] + zi];
    hash_arr[2] /*bba*/ = __pn_P[__pn_P[__pn_P[__pn_rep_inc(xi)] +
                                         __pn_rep_inc(yi)] +
                                         zi];
    hash_arr[3] /*aba*/ = __pn_P[__pn_P[__pn_P[xi] + __pn_rep_inc(yi)] + zi];
    hash_arr[4] /*aab*/ = __pn_P[__pn_P[__pn_P[xi] + yi] + __pn_rep_inc(zi)];
    hash_arr[5] /*bab*/ = __pn_P[__pn_P[__pn_P[__pn_rep_inc(xi)] + yi] +
                                         __pn_rep_inc(zi)];
    hash_arr[6] /*bbb*/ = __pn_P[__pn_P[__pn_P[__pn_rep_inc(xi)] +
                                         __pn_rep_inc(yi)] +
                                         __pn_rep_inc(zi)];
    hash_arr[7] /*abb*/ = __pn_P[__pn_P[__pn_P[xi] + __pn_rep_inc(yi)] +

```

```

        __pn_rep_inc(zi)];

    return hash_arr;
}

/// @brief Generate the perlin noise value for the input co-ordinates
/// if repeat is on, make sure the input co-ordinate map to their "local"
/// co-ordinates i.e. make sure the co-ordinates wrap-around
/// @param inp_x The x co-ordinate of the input point
/// @param inp_y The y co-ordinate of the input point
/// @param inp_z The z co-ordinate of the input point
/// @return float The perlin noise value
float pn_noise(float inp_x, float inp_y, float inp_z)
{
    float x = inp_x;
    float y = inp_y;
    float z = inp_z;
    // the *i represent the co-ordinates of the unit cell in which
    // our input point is located.
    // the *f represent the relative co-ordinates of input point
    // relative to the unit cell i.e. (0.5, 0.5, 0.5) will be at the center
    // of the unit cell
    int xi, yi, zi;
    float xf, yf, zf;
    float u, v, w; // for fading the *f values

    if (__pn_rep_amt > 0)
    {
        x = remainderf(x, (float)__pn_rep_amt);
        y = remainderf(y, (float)__pn_rep_amt);
        z = remainderf(z, (float)__pn_rep_amt);
    }

    // init the *i and *f
    // the *i are bound to 255 to avoid overflow while creating the hashes i.e.
    // accessing the P[] array
    xi = (int)x & 255;
    yi = (int)y & 255;
    zi = (int)z & 255;

    xf = x - (int)x;
    yf = y - (int)y;
    zf = z - (int)z;

    // fade the *f for smoother interpolation
    u = __pn_fadefunc(xf);
    v = __pn_fadefunc(yf);
    w = __pn_fadefunc(zf);

    // get the hashes of all the unit cell co-ords
    int *hashes = __pn_hash(xi, yi, zi);

    // calculate the dot product between the gradient vectors
    // and the distance vectors and linearly interpolate between them
    // ...

    float x1 = __lerp(__pn_grad(hashes[0], xf, yf, zf),
        __pn_grad(hashes[1], xf - 1, yf, zf), u);

    float x2 = __lerp(__pn_grad(hashes[3], xf, yf - 1, zf),
        __pn_grad(hashes[2], xf - 1, yf - 1, zf), u);

    float y1 = __lerp(x1, x2, v); // 1

    // no need to redefine can overwrite the previously

```

```

// "lerp-ed" values safely
x1 = __lerp(__pn_grad(hashes[4], xf, yf, zf - 1),
            __pn_grad(hashes[5], xf - 1, yf, zf - 1), u);

x2 = __lerp(__pn_grad(hashes[7], xf, yf - 1, zf - 1),
            __pn_grad(hashes[6], xf - 1, yf - 1, zf - 1), u);

float y2 = __lerp(x1, x2, v); // 2

FREE(hashes);

// lerp the two y values and map em in the range [0, 1]
return (__lerp(y1, y2, w) + 1) / 2;
}

```

Анализ шума Перлина с разными размерами исходного ключа:

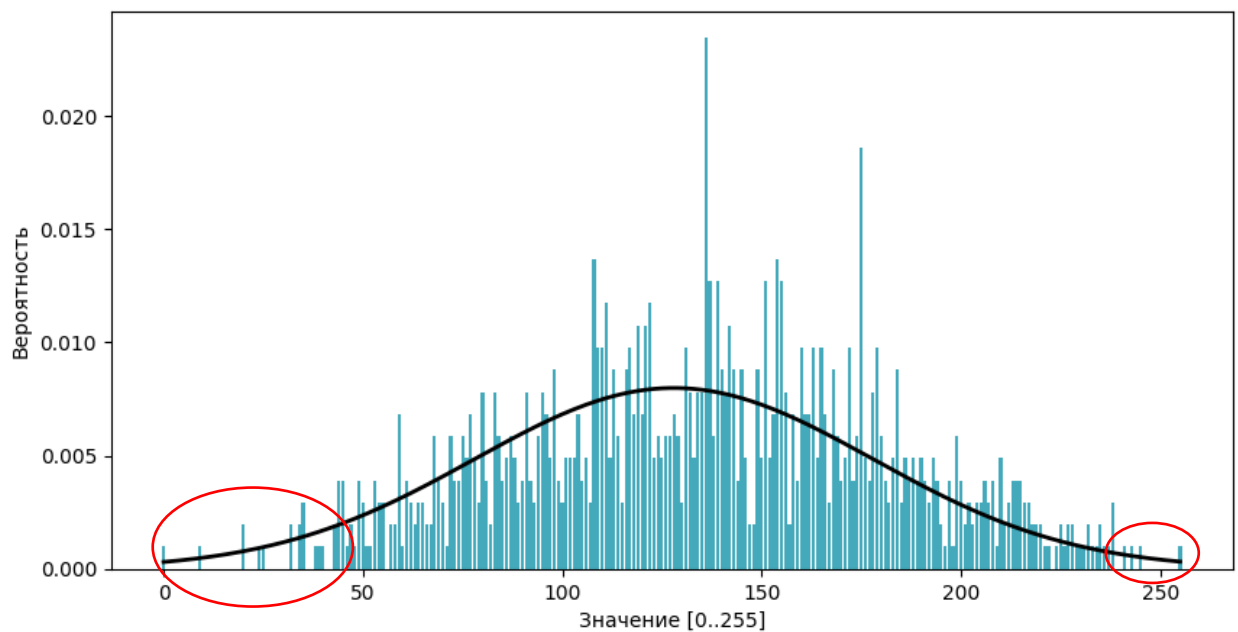


Рисунок 29. Распределение значений, сгенерированных шумом Перлина при длине ключа в 64 бита (синим) и нормальное распределение (черным).

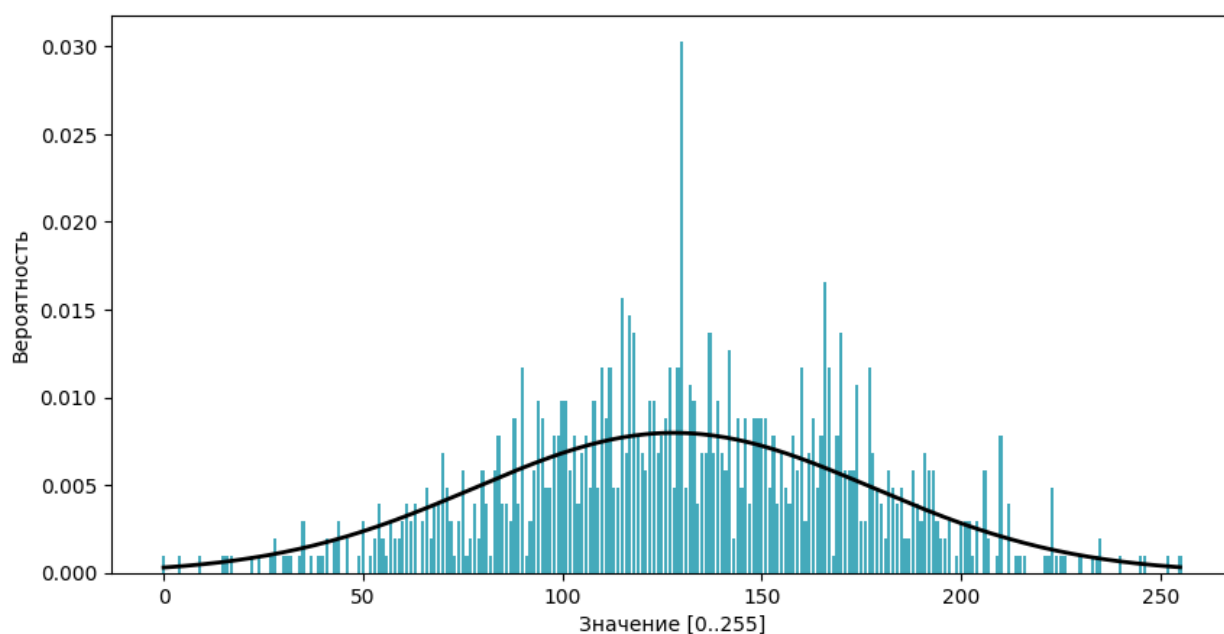


Рисунок 30. Распределение значений, сгенерированных шумом Перлина при длине ключа в 512 бит (синим) и нормальное распределение (черным).

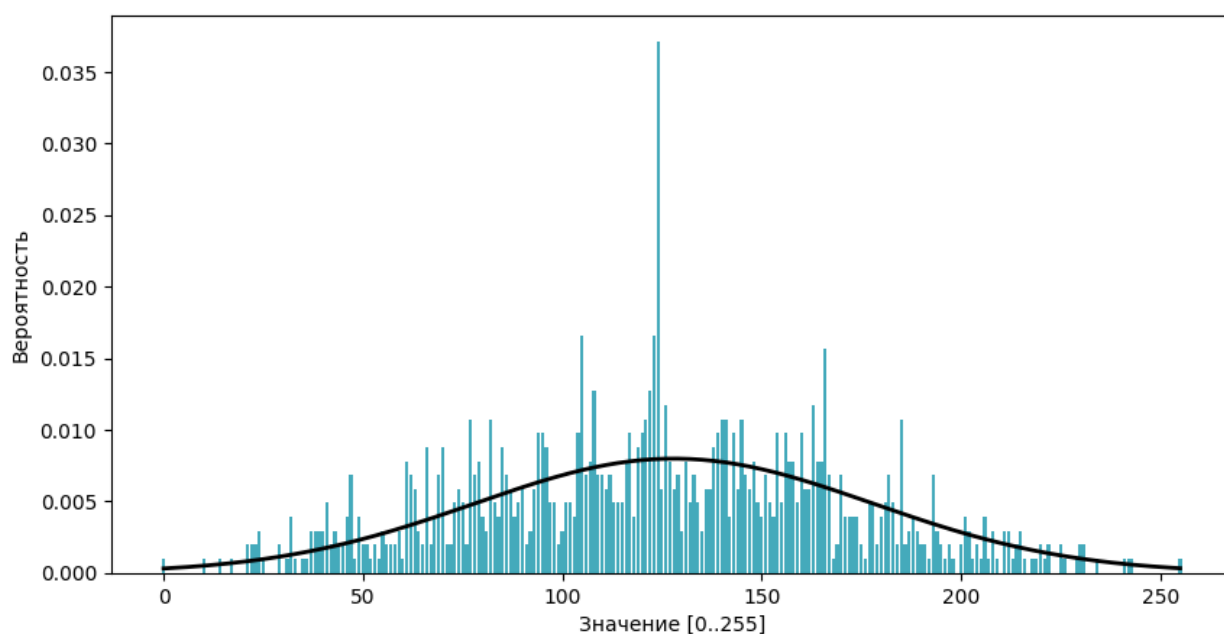


Рисунок 31. Распределение значений, сгенерированных шумом Перлина при длине ключа в 4096 бит (синим) и нормальное распределение (черным).

Исходя из данных, полученных в ходе исследования, можно сделать следующие выводы:

- при длине ключа в 64 бита наблюдаются «пустые» места в распределении (рисунок 29), это значит что в полученной последовательности могут наблюдаться нежелательные закономерности;

– если не считать выбросы в статистике, при длине ключа в 4096 бит наблюдается распределение наиболее близкое к нормальному, количество «пустых» мест минимально.

Это означает, что в ключе длиной в 64 бита недостаточно информации для создания «нормального» шума Перлина. Однако для демонстрационных целей этого должно быть достаточно.

Теперь когда у нас есть возможность создавать таблицы перестановок из ключа, реализуем такую модификацию DES: алгоритм шифрования идентичен оригинальному, но все таблицы, кроме тех, что проблематично генерировать, будем рассчитывать из ключа.

Исходный код реализации DES-Perlin-64:

```
C++
#define DES_PERLIN_ENCRYPTION_MODE 1
#define DES_PERLIN_DECRYPTION_MODE 0

class des_perlin
{
public:
    des_perlin()
    {
    }

    ~des_perlin() = default;

    /// @brief Generate a random 64 bit key
    /// @param key
    void generate_key(byte_t *key)
    {
        int i;
        for (i = 0; i < 8; i++)
        {
            key[i] = rand() % 255;
        }
    }

    /// @brief Encrypt data using DES
    /// @param data
    /// @param data_size
    /// @param enc_data
    /// @param enc_data_size
    /// @param des_key 64 bit key
    void 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");
    }
};
```

```

key_set *key_sets = ALLOC(key_set, 17);
MASSERT(key_sets != NULL, "Memory allocation failed");

if (!is_tables_set || cmp_arrays(last_key, 8, des_key, 8) != 0)
{
    __generate_tables(des_key);
}

if (!is_key_set || cmp_arrays(last_key, 8, des_key, 8) != 0)
{
    __generate_sub_keys(des_key);
}

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];
    }

    __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 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");

    key_set *key_sets = ALLOC(key_set, 17);
    MASSERT(key_sets != NULL, "Memory allocation failed");

    if (!is_tables_set || cmp_arrays(last_key, 8, des_key, 8) != 0)

```

```

    {
        __generate_tables(des_key);
    }

    if (!is_key_set || cmp_arrays(last_key, 8, des_key, 8) != 0)
    {
        __generate_sub_keys(des_key);
    }

    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];
        }

        __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);
}

private:
class key_set
{
public:
    byte_t k[8];
    byte_t c[4];
    byte_t d[4];
};
// unique
byte_t __initial_key_permutaion[56];

// unique
// Initial permutation (IP)
byte_t __initial_message_permutation[64];

// 17
int __key_shift_sizes[17] = {-1,
                             1, 1, 2, 2, 2, 2, 2, 2,
                             1, 2, 2, 2, 2, 2, 2, 1};

// unique
// Subkey permutation
byte_t __sub_key_permutation[48];

// Expansion table (E)
byte_t __message_expansion[48];

// S_i transformation tables
byte_t __S1[64];
byte_t __S2[64];

```



```

byte_t __S3[64];
byte_t __S4[64];
byte_t __S5[64];
byte_t __S6[64];
byte_t __S7[64];
byte_t __S8[64];

// unique
// Permutation table (P)
byte_t __right_sub_msg_permut[32];

// unique
// Final permutation (IP-1)
byte_t __final_msg_permut[64];

bool is_key_set = false;
bool is_tables_set = false;

byte_t last_key[8];
key_set key_sets[17];

/// @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 __generate_sub_keys(byte_t *main_key)
{
    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 = __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 = __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 = __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);
    }
    is_key_set = true;
}

/// @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 __process_data_block(byte_t *data_block,
                        byte_t *processed_block,
                        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 = __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 = __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_PERLIN_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)__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)__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)__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)__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)__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)__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)__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)__S8[row * 16 + column];

for (i = 0; i < 4; i++)
{
    rn[i] = 0;

```

```

    }

    // Straight permutation (P)
    for (i = 0; i < 32; i++)
    {
        shift_size = __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 = __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 __encrypt_block(byte_t *data_block,
                    byte_t *processed_block,
                    key_set *key_sets)
{
    __process_data_block(data_block,
                        processed_block,
                        key_sets,
                        DES_PERLIN_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 __decrypt_block(byte_t *data_block,
                    byte_t *processed_block,
                    key_set *key_sets)

```

```

{
    __process_data_block(data_block,
                        processed_block,
                        key_sets,
                        DES_PERLIN_DECRYPTION_MODE);
}

/// @brief Check if array contains unique values between 1 and "size".
/// @param data
/// @param size
/// @return
bool is_key_acceptable(byte_t *data, size_t size)
{
    for (size_t i = 0; i < size; i++)
    {
        if (data[i] < 1 || data[i] > size)
        {
            return false;
        }
    }

    for (size_t i = 0; i < size; i++)
    {
        for (size_t j = i + 1; j < size; j++)
        {
            if (data[i] == data[j])
            {
                return false;
            }
        }
    }

    return true;
}

/// @brief Remap key data, so it contains all values between 1 and "size".
/// Use current data as orders
/// @param data
/// @param size
void make_key_acceptable(byte_t *data, size_t size)
{
    while (!is_key_acceptable(data, size))
    {
        // find lowest and replace it with 1
        byte_t lowest = 255;
        for (size_t i = 0; i < size; i++)
        {
            if (data[i] < lowest)
            {
                lowest = data[i];
            }
        }

        for (size_t i = 0; i < size; i++)
        {
            data[i] = (data[i] - lowest) + 1;
        }

        // find highest and replace it with size
        byte_t highest = 0;
        for (size_t i = 0; i < size; i++)
        {
            if (data[i] > highest)
            {
                highest = data[i];
            }
        }
    }
}

```

```

    }
}

for (size_t i = 0; i < size; i++)
{
    data[i] = (data[i] * size) / highest;
}

for (size_t i = 0; i < size; i++)
{
    if (data[i] == 0)
    {
        data[i] = 1;
    }
}
make_array_unique(data, size);
}
}

void __generate_tables(byte_t *key)
{
    // 56 + 64 + 48 + 48 + 64 * 8 + 32 + 64 = 824 bytes
    float perlin_buffer[1024];
    byte_t int_perlin_buffer[1024];
    memset(perlin_buffer, 0, 1024 * sizeof(float));

    byte_t expanded_key[256];
    resize_key(key, 8, expanded_key, 256);
    pn_init(expanded_key);
    pn_octave_noise_2d(perlin_buffer, 32, 32, 0.3, 2);
    normalize_array(perlin_buffer, 1024);

    intify_array(perlin_buffer, 1024, int_perlin_buffer);

    memcpy(__initial_key_permutation, int_perlin_buffer, 56);
    make_key_acceptable(__initial_key_permutation, 56);

    memcpy(__initial_message_permutation, int_perlin_buffer + 56, 64);
    make_key_acceptable(__initial_message_permutation, 64);
    memcpy(__initial_message_permutation,
        __des_initial_message_permutation, 64.);

    memcpy(__sub_key_permutation, int_perlin_buffer + 120, 48);
    make_key_acceptable(__sub_key_permutation, 48);

    memcpy(__message_expansion, int_perlin_buffer + 168, 48);
    memcpy(__message_expansion, __des_message_expansion, 48);

    memcpy(__S1, int_perlin_buffer + 216, 64);
    make_key_acceptable(__S1, 64);
    memcpy(__S2, int_perlin_buffer + 280, 64);
    make_key_acceptable(__S2, 64);
    memcpy(__S3, int_perlin_buffer + 344, 64);
    make_key_acceptable(__S3, 64);
    memcpy(__S4, int_perlin_buffer + 408, 64);
    make_key_acceptable(__S4, 64);
    memcpy(__S5, int_perlin_buffer + 472, 64);
    make_key_acceptable(__S5, 64);
    memcpy(__S6, int_perlin_buffer + 536, 64);
    make_key_acceptable(__S6, 64);
    memcpy(__S7, int_perlin_buffer + 600, 64);
    make_key_acceptable(__S7, 64);
    memcpy(__S8, int_perlin_buffer + 664, 64);
    make_key_acceptable(__S8, 64);
}

```



```

memcpy(__right_sub_msg_permut, int_perlin_buffer + 728, 32);
make_key_acceptable(__right_sub_msg_permut, 32);

memcpy(__final_msg_permut, int_perlin_buffer + 760, 64);
make_key_acceptable(__final_msg_permut, 64);
memcpy(__final_msg_permut, __des_final_msg_permut, 64);

bool verbose_sets = 0;
if (verbose_sets)
{
    printf("\n");
    print_array_hex_line(__initial_key_permutaion, 56);
    printf("\n");
    print_array_hex_line(__des_initial_key_permutaion, 56);
    printf("\n");

    print_array_hex_line(__initial_message_permutation, 64);
    printf("\n");
    print_array_hex_line(__des_initial_message_permutation, 64);
    printf("\n");

    print_array_hex_line(__sub_key_permutation, 48);
    printf("\n");
    print_array_hex_line(__des_sub_key_permutation, 48);
    printf("\n");

    print_array_hex_line(__message_expansion, 48);
    printf("\n");
    print_array_hex_line(__des_message_expansion, 48);
    printf("\n");

    print_array_hex_line(__S1, 64);
    printf("\n");
    print_array_hex_line(__des_S1, 64);
    printf("\n");

    print_array_hex_line(__S2, 64);
    printf("\n");
    print_array_hex_line(__des_S2, 64);
    printf("\n");

    print_array_hex_line(__S3, 64);
    printf("\n");
    print_array_hex_line(__des_S3, 64);
    printf("\n");

    print_array_hex_line(__S4, 64);
    printf("\n");
    print_array_hex_line(__des_S4, 64);
    printf("\n");

    print_array_hex_line(__S5, 64);
    printf("\n");
    print_array_hex_line(__des_S5, 64);
    printf("\n");

    print_array_hex_line(__S6, 64);
    printf("\n");
    print_array_hex_line(__des_S6, 64);
    printf("\n");

    print_array_hex_line(__S7, 64);
    printf("\n");
    print_array_hex_line(__des_S7, 64);

```

```

        printf("\n");

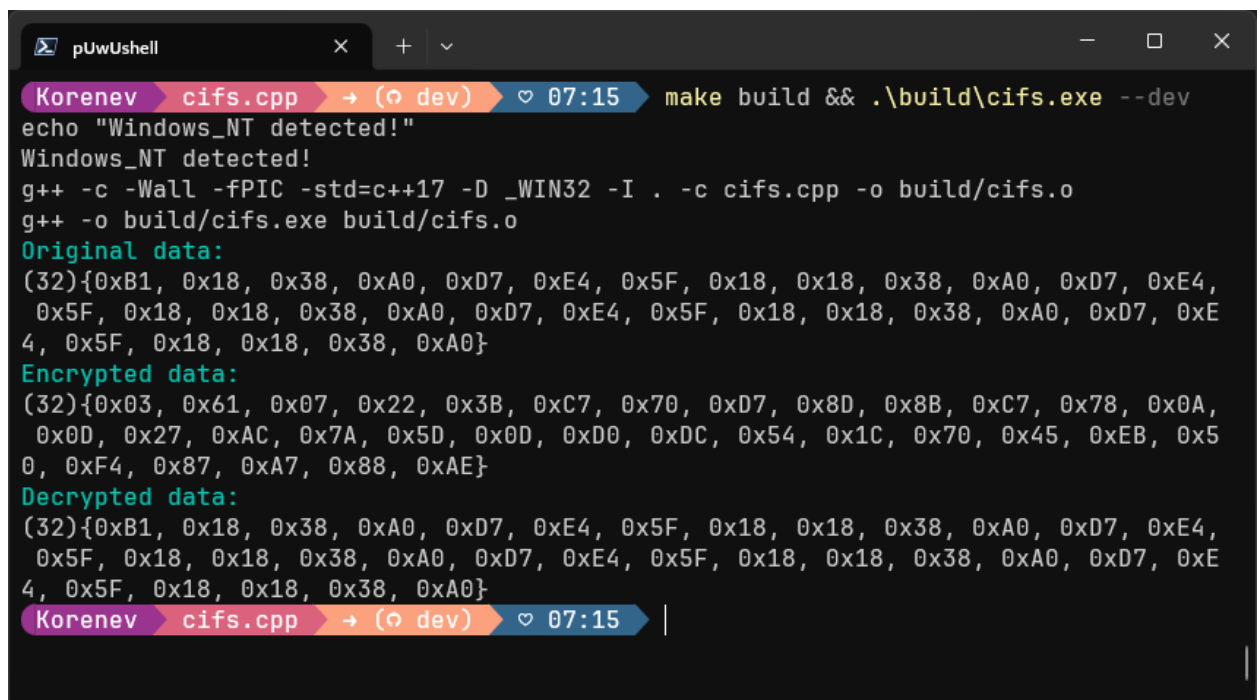
        print_array_hex_line(__S8, 64);
        printf("\n");
        print_array_hex_line(__des_S8, 64);
        printf("\n");

        print_array_hex_line(__right_sub_msg_permut, 32);
        printf("\n");
        print_array_hex_line(__des_right_sub_msg_permut, 32);
        printf("\n");

        print_array_hex_line(__final_msg_permut, 64);
        printf("\n");
        print_array_hex_line(__des_final_msg_permut, 64);
        printf("\n");
        printf("\n");
    }
    is_tables_set = true;
}
};

```

Пример работы алгоритма:



```

pUwUshell
Korenev cifs.cpp → (dev) 07:15 make build && .\build\cifs.exe --dev
echo "Windows_NT detected!"
Windows_NT detected!
g++ -c -Wall -fPIC -std=c++17 -D _WIN32 -I . -c cifs.cpp -o build/cifs.o
g++ -o build/cifs.exe build/cifs.o
Original data:
(32){0xB1, 0x18, 0x38, 0xA0, 0xD7, 0xE4, 0x5F, 0x18, 0x18, 0x38, 0xA0, 0xD7, 0xE4,
    0x5F, 0x18, 0x18, 0x38, 0xA0, 0xD7, 0xE4, 0x5F, 0x18, 0x18, 0x38, 0xA0, 0xD7, 0xE
    4, 0x5F, 0x18, 0x18, 0x38, 0xA0}
Encrypted data:
(32){0x03, 0x61, 0x07, 0x22, 0x3B, 0xC7, 0x70, 0xD7, 0x8D, 0x8B, 0xC7, 0x78, 0x0A,
    0x0D, 0x27, 0xAC, 0x7A, 0x5D, 0x0D, 0xD0, 0xDC, 0x54, 0x1C, 0x70, 0x45, 0xEB, 0x5
    0, 0xF4, 0x87, 0xA7, 0x88, 0xAE}
Decrypted data:
(32){0xB1, 0x18, 0x38, 0xA0, 0xD7, 0xE4, 0x5F, 0x18, 0x18, 0x38, 0xA0, 0xD7, 0xE4,
    0x5F, 0x18, 0x18, 0x38, 0xA0, 0xD7, 0xE4, 0x5F, 0x18, 0x18, 0x38, 0xA0, 0xD7, 0xE
    4, 0x5F, 0x18, 0x18, 0x38, 0xA0}
Korenev cifs.cpp → (dev) 07:15 |

```

Рисунок 32. Шифрование и дешифрование алгоритмом DES-Perlin-64.

Приложения

Приложение 1. Ссылка на исходный код программы.

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