Московский Авиационный институт

(Национальный исследовательский университет)

Институт №8

«Компьютерные науки и прикладная математика»

Кафедра 813

«Компьютерная математика»

Курсовая работа по дисциплине
«Основы криптографии»

Тема: «Разработка криптографического приложения на основе алгоритмов LUC и DEAL»

Студент: Буреева П.С.

Группа: М80-311Б-19

Преподаватель: Романенков А.М.

Оценка:

Дата:

Москва 2022

Оглавление

Введение

Криптография — это наука о методах шифрования, где данные, подлежащие шифрованию, будут зашифрованы с использованием ключа в данные, которые трудно прочитать другим сторонам, у которых нет ключа дешифрования. Процесс защиты сообщений или информации таким образом, чтобы ее нельзя было понять и прочитать без помощи знаний или специальных инструментов, называется шифрованием. Между тем, процесс преобразования зашифрованной информации в информацию, которую снова можно понять, называется дешифрованием.

Алгоритм DEAL — это блочный криптографический алгоритм симметричного шифрования, представляет собой сеть Фейстеля использующую DES как функцию в раунде сети. Удовлетворяя требованиям AES, DEAL имеет длину блока 128 бит и длину ключа 128, 192 или 256 бит.

Алгоритм LUC — это асимметричный криптографический алгоритм, который использует два простых числа для генерации открытых ключей и секретных ключей.

В курсовой работе алгоритм DEAL используется для шифрования сообщения, в то время как алгоритм LUC используется для шифрования внешнего ключа DEAL перед отправкой ключа получателю. Гибридные криптосистемы могут обеспечить более высокий уровень безопасности.

Безопасность сообщений с использованием гибридного алгоритма шифрования криптографических данных DEAL и LUC выше, поскольку ключ шифрования, сгенерированный с помощью шифрования внешнего ключа DEAL, отличается и длиннее. Это очень затрудняет угадывание ключа, поскольку количество ключевых символов, отправленных получателю, не совпадает с фактическим количеством ключевых символов.

Теоретическая часть

Алгоритм LUC

Описание

LUC [1-5] — это асимметричный алгоритм криптографии с открытым ключом, основанный на продолжениях Лукаса. Эта криптосистема была разработана исследователями из Австралии и Новой Зеландии, в первую очередь П. Смитом и К. Скиннером.

Шифрование представляет собой реализацию, аналогичную протоколам ElGamal, обмена ключами Диффи-Хеллмана и RSA. Сообщение шифруется путем повторения рекуррентного отношения последовательностей Лукаса вместо возведения в степень, присутствующего в RSA и в Диффи-Хеллмане. В лучшем случае LUC не безопаснее RSA.

Безопасность криптосистемы LUC основана на трудности нахождения секретного ключа d. Считалось, что нахождение секретного ключа d из открытых ключей e и n вычислительно эквивалентно разложению на множители составного числа n.

Последовательности Люка

Последовательности Люка [4] — это семейство пар линейных рекуррентных последовательностей второго порядка, впервые рассмотренных Эдуардом Люка.

Последовательности Люка представляют собой пары последовательностей $\{U_n(P,Q)\}$ и $\{V_n(P,Q)\}$, удовлетворяющих одному и тому же рекуррентному соотношению с коэффициентами P и Q:

$$U_{n+2}(P,Q) = P \cdot U_{n+1}(P,Q) - Q \cdot U_n(P,Q), \qquad n \geq 0$$

$$V_{n+2}(P,Q) = P \cdot V_{n+1}(P,Q) - Q \cdot V_n(P,Q), \qquad n \geq 0$$

$$U_0(P,Q) = 0, \qquad U_1(P,Q) = 1,$$

$$V_0(P,Q) = 2, \qquad V_1(P,Q) = P,$$

где P и Q — целые неотрицательные числа.

В основе используется последовательность $V_n(P,Q)$.

Последовательности Люка растут очень быстро, поэтому используется следующее свойство [3]:

$$V_n(P \bmod N, Q \bmod N) = V_n(P,Q) \bmod N, \qquad N \geq 2$$

а также утверждение:

$$V_n(V_k(P,Q), Q^k) = V_{nk}(P,Q)$$

Использование последовательностей Люка в криптографии

Допустим, существуют такие e и d, что

$$V_{de}(X, 1) = X \bmod N \Rightarrow$$

$$V_{de}(X \bmod N, 1) = V_{de}(X, 1) \bmod N = X \bmod N =$$

$$= V_d(V_e(X \bmod N, 1), 1) = X \bmod N$$

Сначала от символьного сообщения берётся хеш-функция, которая возвращает цифровое значение X. В качестве функции шифрования используется:

$$Y = V_e(X \bmod N, 1)$$

А для дешифрования:

$$V_d(Y \bmod N, 1) = X \bmod N$$

В этом алгоритме шифрования открытым ключом будет пара $\{e, N\}$, а закрытым $\{d, N\}$.

Символ Лежандра

Символ Лежандра [6] — функция, используемая в теории чисел. Введён французским математиком А. М. Лежандром.

Пусть a — целое число, p — простое число, отличное от 2. Тогда символ Лежандра $\left(\frac{a}{p}\right)$ определяется следующим образом:

1. $\left(\frac{a}{p}\right) = 0$, если a делится на p;
2. $\left(\frac{a}{p}\right) = 1$, если a является квадратичным вычетом по модулю p, то есть существует такое целое x, что $x^2 \equiv a \pmod{p}$, но при этом не делится на p;
3. $\left(\frac{a}{p}\right) = -1$, если a является квадратичным невычетом по модулю p.

Вероятностные тесты простоты

Тест Ферма

По Теореме Ферма [7], если n — простое число, тогда для любого a справедливо следующее равенство $a^{n-1} \equiv 1\ (mod\ n)$.

Возьмем псевдослучайное a: $\begin{cases}(a,p)=1\\1<a<n\end{cases}$,

тогда при выполнении $a^{n-1} \equiv 1\ (mod\ n)$ вероятность того, что p — простое число, равна $1/2$. Далее возьмем другое a.

Вероятность того, что p — простое число, $1 - 2^{-k}$, где k — число взятия a.

Однако существуют составные числа, для которых сравнение $a^{n-1} \equiv 1\ (mod\ n)$ выполняется для всех a, взаимно простых с n — это числа Кармайкла. Чисел Кармайкла — бесконечное множество, наименьшее число Кармайкла — 561. Тем не менее, тест Ферма довольно эффективен для обнаружения составных чисел.

При использовании алгоритмов быстрого возведения в степень по модулю время работы теста Ферма для одного a оценивается как $O(\log^2 n \times \log\log n \times \log\log\log n)$, где n — проверяемое число.

Тест Соловея-Штрассена

Алгоритм Соловея — Штрассена параметризуется количеством раундов k. В каждом раунде случайным образом выбирается число $a < n$. Если $НОД(a,n) > 1$, то выносится решение, что n — составное. Иначе проверяется справедливость сравнения $a^{(n-1)/2} \equiv \left(\frac{a}{n}\right)\ (\mathrm{mod}\ n)$. Если оно не выполняется, то выносится решение, что n — составное. Если это сравнение выполняется, то a является свидетелем простоты числа n. Далее выбирается другое случайное a и процедура повторяется.

После нахождения k свидетелей простоты в k раундах выносится заключение, что n является простым числом с вероятностью $1 - 2^{-k}$.

Теоретическая сложность вычислений теста оценивается как $O(\log^3 n)$.

Тест Миллера-Рабина

Как и тесты Ферма и Соловея — Штрассена, тест Миллера — Рабина [7] []опирается на проверку ряда равенств, которые выполняются для простых чисел. Если хотя бы одно такое равенство не выполняется, это доказывает что число составное.

Возьмём любое простое число p. Пусть $p - 1 = 2^{\mathrm{т}} \cdot m$, где m — нечётно.

Утверждение:

Для любого числа x выполнено:

- Либо что $x^m \equiv 1 (\mathrm{mod}\ p)$.
- Либо что существует такое $k < n$, что $x^{2^k \cdot m} \equiv 1 (\mathrm{mod}\ p)$.

Тогда заметим, что если есть какое-то число x, для которого не выполнено описанное утверждение, то значит p точно не простое. Число x называется свидетелем простоты числа p, если для него выполнено описанное выше утверждение (при этом p в этом случае необязательно простое).

По теореме Рабина у нечетного составного n не более $\varphi(n)/4$ свидетелей простоты, где $\varphi(n)$ — функция Эйлера. Так как $\varphi(n) < n$, то если взять случайное число x, то оно с вероятностью меньше $1/4$ будет свидетелем простоты числа p.

После нахождения k свидетелей простоты в k раундах выносится заключение, что p является простым числом с вероятностью $1 - 2^{-2k}$.

Время работы оценивается как $O(k \cdot \log^3 n)$.

Генерация ключей

1. Выбираются два простых числа p и q;
2. Вычисляется $N = p \cdot q$;
3. Выбирается число e, взаимнопростое с числом $(p-1)(q-1)(p+$ $+1)(q+1)$:

$$\gcd[(p-1)(q-1)(p+1)(q+1), e] = 1$$

4. Вычисляется $D = P^2 - 4$, где P — шифруемое сообщение;
5. Находятся символы Лежандра: $\left(\frac{D}{p}\right)$ и $\left(\frac{D}{q}\right)$;
6. Находится наименьшее общее кратное:

$$S(N) = lcm[(p - \left(\frac{D}{p}\right)), (q - \left(\frac{D}{q}\right)) \ ]$$

7. Вычисляется $d = e^{-1} \bmod S(N)$.

В итоге получаем открытый ключ $KU = (e, N)$ и закрытый ключ $KR =$ $= (d, N)$.

Шифрование и дешифрование сообщения

Шифруется сообщение при условии, что сообщение $P < N$:

$$C = V_e(P, 1) \bmod N$$

Дешифрование происходит следующим образом:

$$P = V_d(C, 1) \bmod N$$

Алгоритм DEAL

Описание

DEAL [8][9] — это блочный шифр, производный от DES. DEAL был впервые представлен Ларсом Кнудсеном в виде доклада на конференции Selected Areas in Cryptography в 1997 году. Впоследствии этот же алгоритм был представлен Ричардом Аутербриджем на конкурсе AES в 1998 году.

DEAL представляет собой сеть Фейстеля использующую DES как функцию в раунде сети. Удовлетворяя требованиям AES, DEAL имеет длину блока 128 бит и длину ключа 128, 192 или 256 бит. Алгоритм сопоставим по производительности с Triple DES, но по сравнению с конкурентами в конкурсе AES алгоритм оказывается довольно медленным.

К 1998 году стало ясно, что существовавший до этого на правах стандарта шифр DES совершенно не удовлетворяет требованиям времени. Было показано, что за сумму порядка одного миллиона долларов можно создать устройство, перебирающее все ключи DES меньше чем за 3,5 часа. Существовавшая альтернатива — «тройной DES» — так же не обеспечивал требуемой надежности, так как (в некоторых режимах своей работы) уязвим для атаки по подобранному шифро-тексту. Учитывая широкую распространённость DES к тому моменту (в том числе в виде эффективных аппаратных реализаций) Ларсом Кнудсеном был предложен шифр DEAL — шифр с Фейстелевой структурой, использующий DES в качестве раундовой функции и имеющий r раундов. Таким образом DEAL представляет собой шифр с размером блока 128 бит и r — 64 битами раундовых ключей, вычисляемых с помощью алгоритма расписания ключей. Расписание ключей предусматривает что размер исходного ключа один принимает одно из трёх различных значений: 128, 192 или 256 бит. Для первых двух размеров ключей предлагалось положить r = 6, а при размере ключа 256 бит r = 8. Полученный таким образом ключ по скорости сопоставим с Triple DES

Сеть Фейстеля

Сеть Фейстеля — один из методов построения блочных шифров. Сеть состоит из ячеек, называемых ячейками Фейстеля. На вход каждой ячейки поступают данные и ключ. На выходе каждой ячейки получают изменённые данные и изменённый ключ. Все ячейки однотипны, и говорят, что сеть представляет собой определённую многократно повторяющуюся структуру. Ключ выбирается в зависимости от алгоритма шифрования/расшифрования и меняется при переходе от одной ячейки к другой. При шифровании и расшифровании выполняются одни и те же операции; отличается только порядок ключей.

Шифрование

Информация разбивается на блоки одинаковой (фиксированной) длины. Полученные блоки называются входными, так как поступают на вход алгоритма. В случае если длина входного блока меньше, чем размер, который выбранный алгоритм шифрования способен зашифровать единовременно (размер блока), то блок удлиняется каким-либо способом. Как правило, длина блока является степенью двойки, например, составляет 64 бита или 128 бит.

Расшифрование

Расшифровка информации происходит так же, как и шифрование, с тем лишь исключением, что ключи следуют в обратном порядке, то есть не от первого к N-му, а от N-го к первому.

Алгоритмическое описание

1. Блок открытого текста делится на две равные части (L_0, R_0).
2. В каждом раунде вычисляются:

$$L_i = R_{i-1} \oplus f(L_{i-1}, K_{i-1});$$

$$R_i = L_{i-1},$$

где i — номер раунда, $i = 1 \ldots N$; N — количество раундов в выбранном алгоритме шифрования; f — некоторая функция; K_{i-1} — ключ i-го раунда (раундовый ключ).

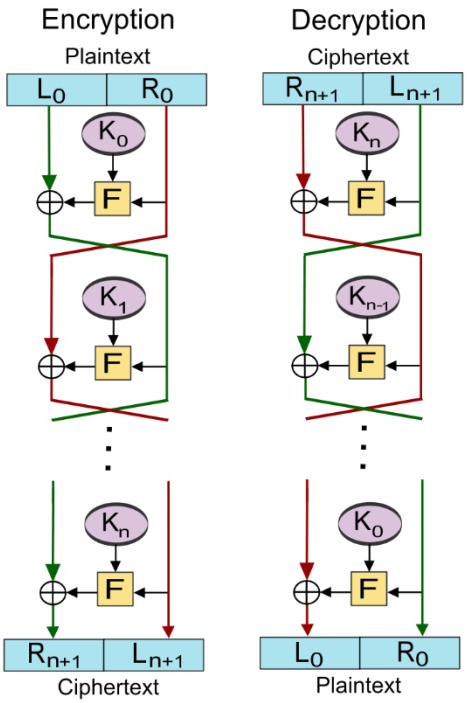

Рисунок 1 Сеть Фейстеля

Результатом выполнения N раундов является (L_N, R_N). В N-м раунде перестановка L_N и R_N не производится, чтобы была возможность использовать ту же процедуру и для расшифрования, просто инвертировав порядок использования ключей.

$$L_{i-1} = R_i \oplus f(L_i, K_{i-1});$$

$$R_{i-1} = L_i.$$

Режимы шифрования

Режим шифрования [7] — метод применения блочного шифра, позволяющий преобразовать последовательность блоков открытых данных в последовательность блоков зашифрованных данных. При этом для шифрования одного блока могут использоваться данные другого блока.

Обычно режимы шифрования используются для изменения процесса шифрования так, чтобы результат шифрования каждого блока был уникальным вне зависимости от шифруемых данных и не позволял сделать какие-либо выводы об их структуре. Это обусловлено, прежде всего, тем, что блочные шифры шифруют данные блоками фиксированного размера, и поэтому существует потенциальная возможность утечки информации о повторяющихся частях данных, шифруемых на одном и том же ключе.

Используемые далее обозначения:

- i — номер блока;
- k — ключ;
- IV — вектор инициализации, размер вектора равен размеру блока шифротекста, является случайным числом;
- R_i — блок сообщения (открытый текст);
- C_{i-1} — зашифрованный блок (шифротекст), полученный на предыдущем шаге шифрования;
- E_k — функция, выполняющая блочное шифрование;
- D_k — функция, выполняющая блочное расшифрование;
- Random Delta — младшие 8 байт вектора инициализации.

ECB — режим электронной кодовой книги

Шифрование: $C_i = E_k(P_i, k)$

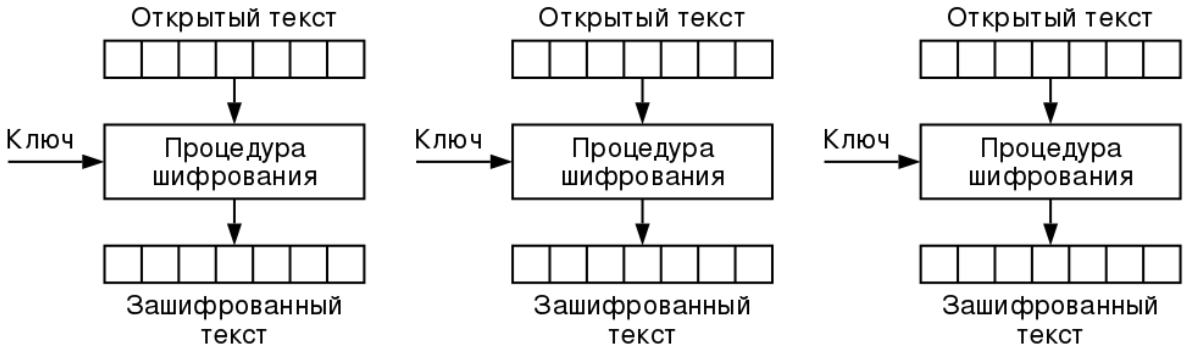

Рисунок 2 Шифрование в режиме ECB

Расшифрование: $P_i = E_k(C_i, k)$

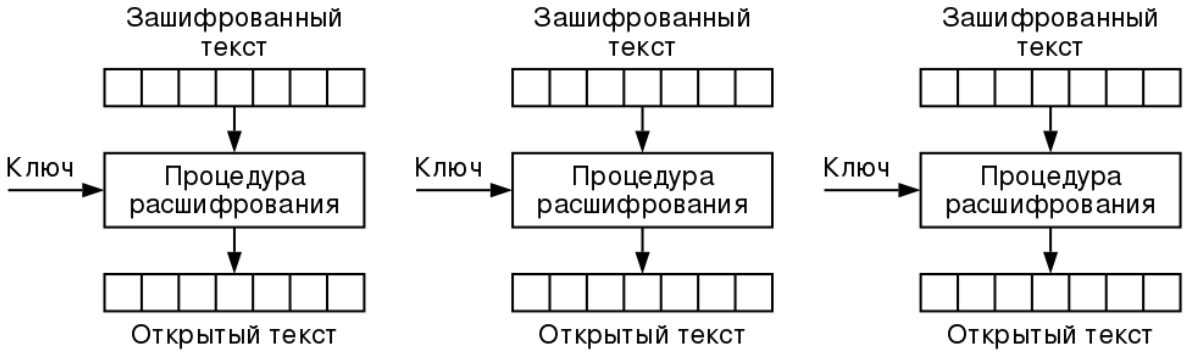

Рисунок 3 Расшифрование в режиме ECB

CBC — режим сцепления блоков шифротекста

Шифрование: $C_o = IV,\ C_i = E_k(P_i \oplus C_{i-1}, k)$

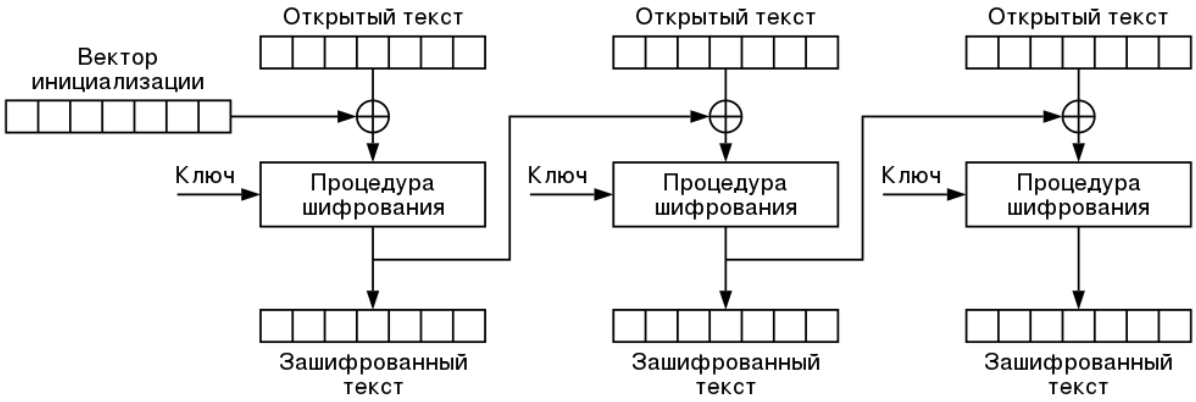

Рисунок 4 Шифрование в режиме CBC

Расшифрование: $C_o = IV,\ P_i = C_{i-1} \oplus D_k(C_i, k)$

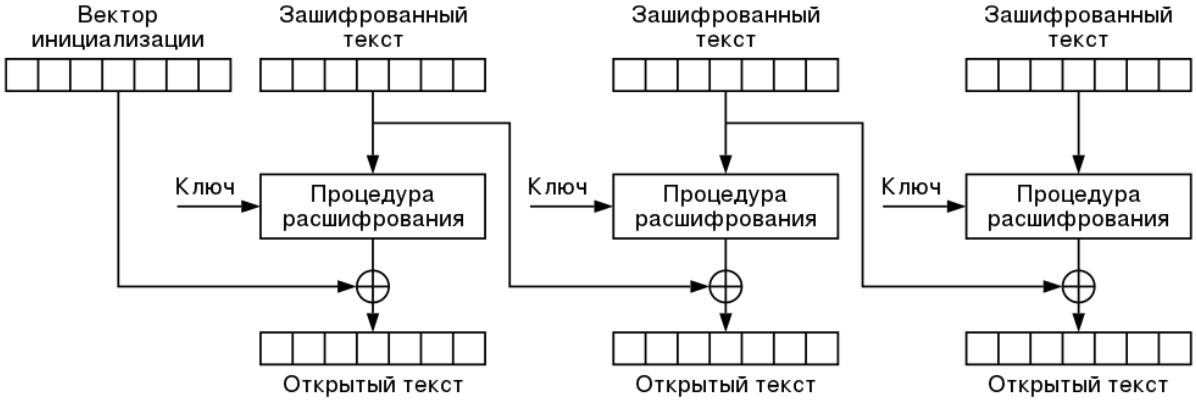

Рисунок 5 Расшифрование в режиме CBC

CFB — режим обратной связи по шифротексту

Шифрование: $C_o = IV,\ C_i = E_k(C_{i-1}, k) \oplus P_i$

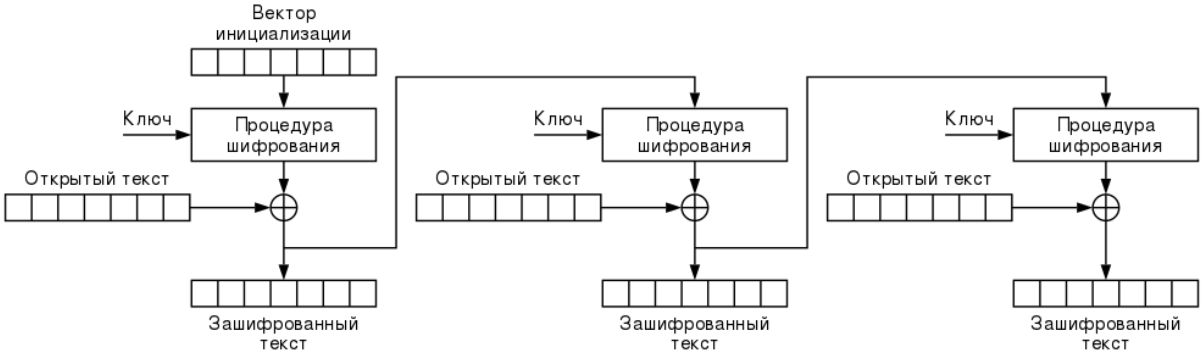

Рисунок 6 Шифрование в режиме CFB

Расшифрование: $C_o = IV,\ P_i = C_{i-1} \oplus D_k(C_i, k)$

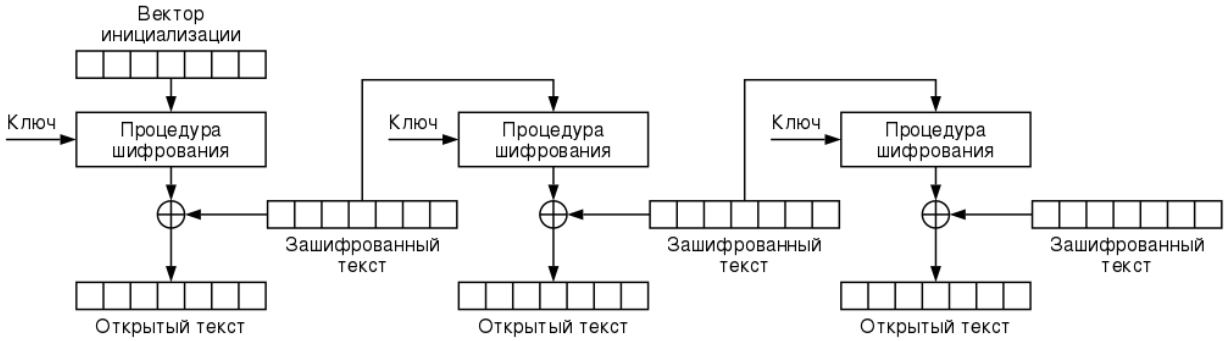

Рисунок 7 Расшифрование в режиме CFB

OFB — режим обратной связи по выходу

Шифрование: $O_o = IV,\ C_0 = E_k(O_{i-1}),\ C_i = P_i \oplus O_i$

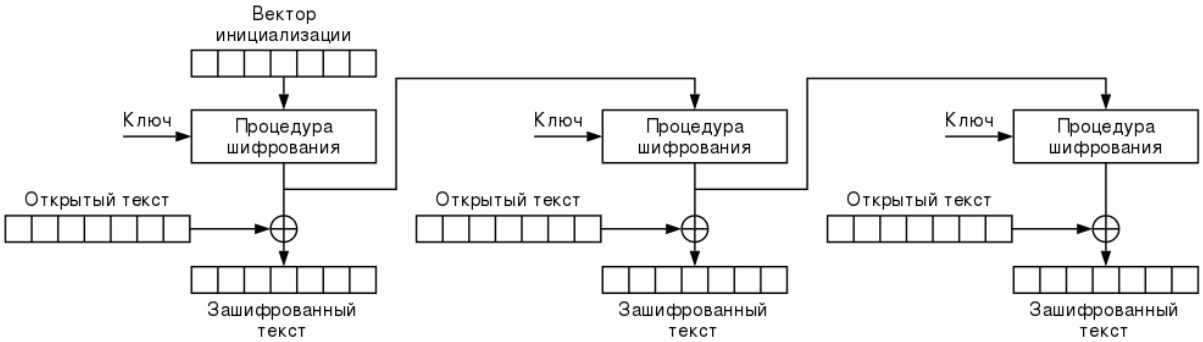

Рисунок 8 Шифрование в режиме OFB

Расшифрование: $O_o = IV,\ C_0 = E_k(O_{i-1}),\ P_i = C_i \oplus O_i$

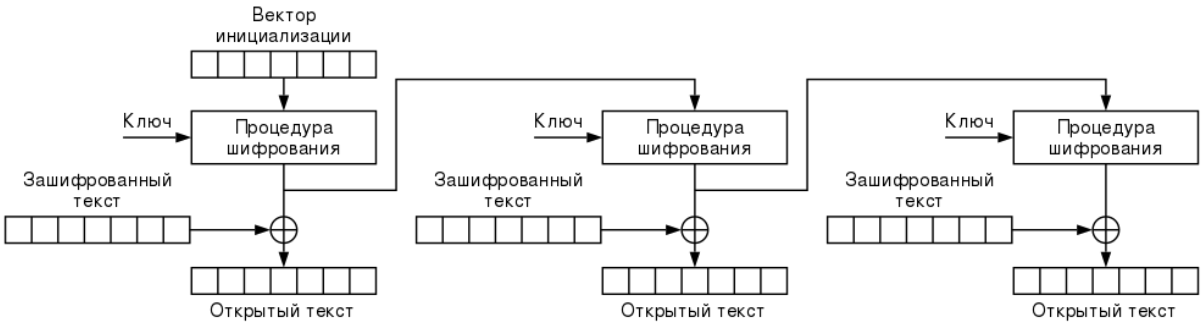

Рисунок 9 Расшифрование в режиме OFB

CTR — режим счётчика

Шифрование: $C_i = P_i \oplus E_k(Ctr_i)$, где Ctr_i — значение счётчика для i-го блока.

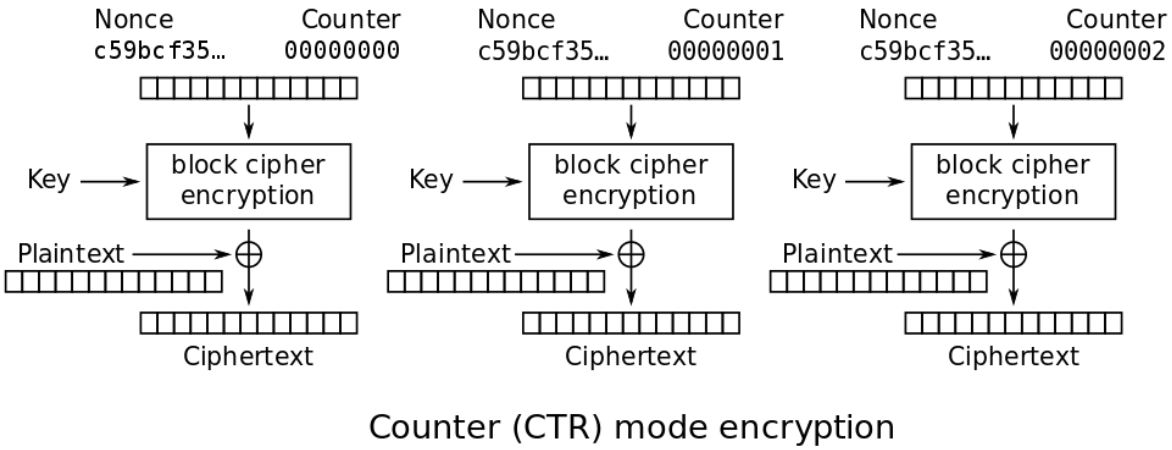

Рисунок 10 Шифрование в режиме CTR

Расшифрование: $P_i = C_i \oplus E_k(Ctr_i)$

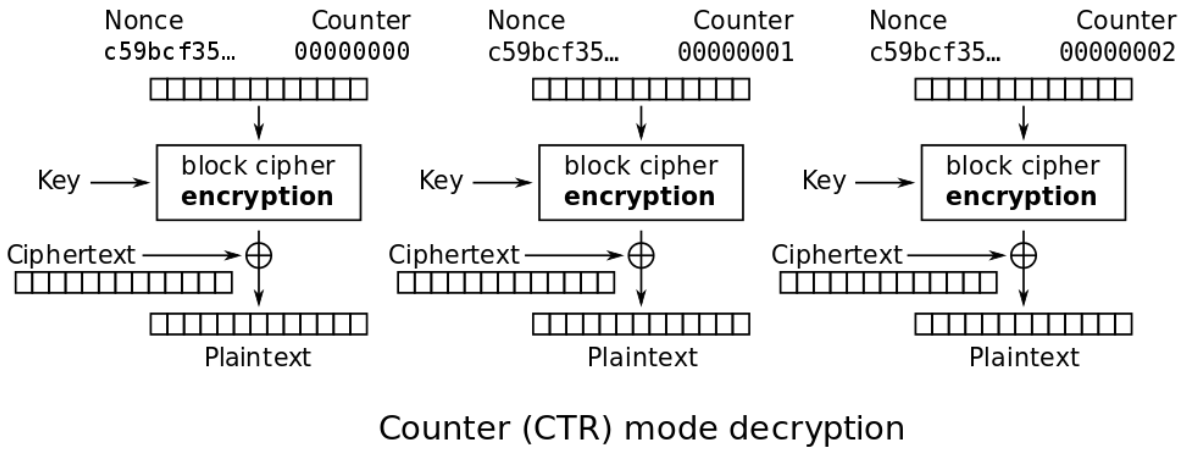

Рисунок 11 Расшифрование в режиме CTR

RD

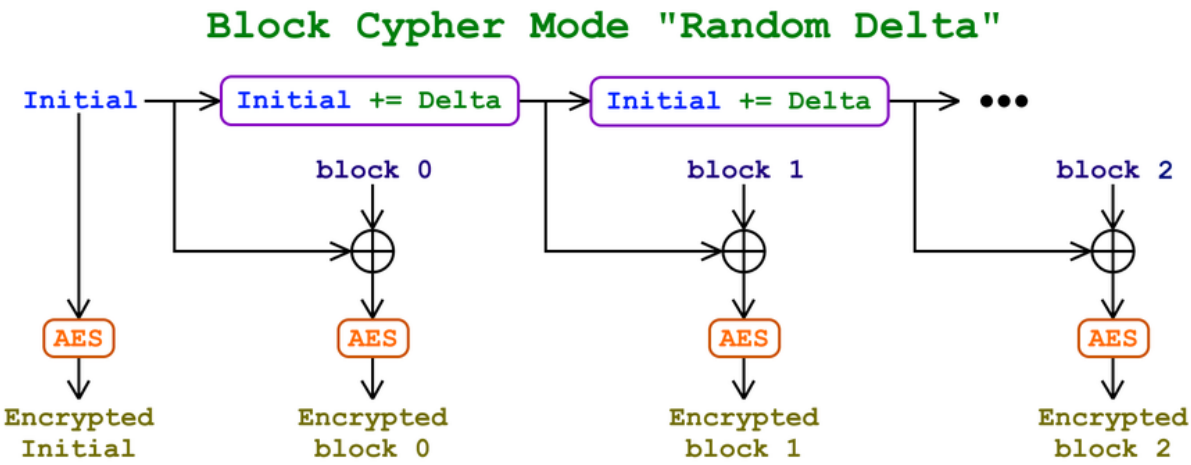

Рисунок 12 Шифрование в режиме RD

RDH

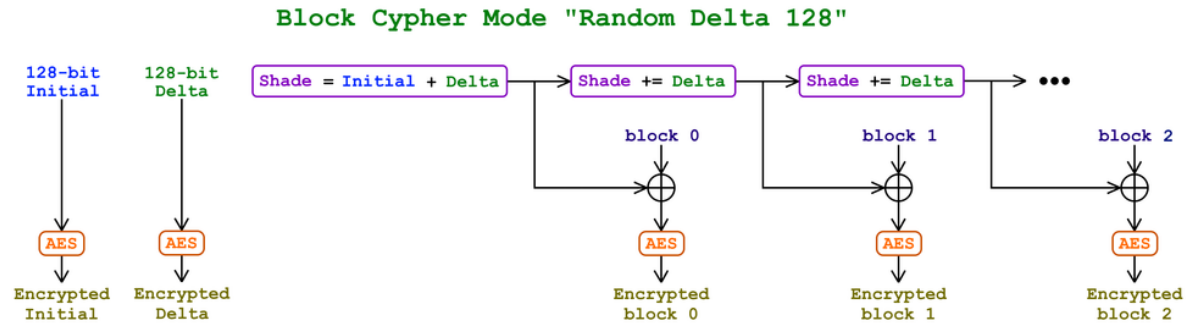

Рисунок 13 Шифрование в режиме RDH

Дополнение (padding) PKCS7

Дополнение в криптографии — добавление ничего не значащих данных к зашифровываемой информации, нацеленное на повышение криптостойкости.

Дополнение в целых байтах. Значение каждого байта равно числу добавленных байтов, то есть добавляется N байт со значением N. Число добавленных байтов зависит от границы блока, до которого необходимо расширить сообщение. Дополнение будет одним из:

```
01
02 02
03 03 03
04 04 04 04
05 05 05 05 05
etc.
```

Рисунок 14 Дополнения в PKCS7

Данный метод дополнения (также как и два предыдущих) хорошо определён, только если N меньше, чем 256.

Генерация раундовых ключей

На вход поступает DEAL ключ [9][10] размером длиной 128, 192 или 256 бит, который делится на s = 2, 3 или 4 ключей по 64 бита: $K_1 \dots K_s$.

Расписание ключей единообразно для всех трёх длин исходного ключа. Сначала расширяем s ключей до r ключей, создавая отсутствующие ключи операцией XOR с новой константой для каждого нового ключа. Константа используется чтобы исключить слабые ключи. К полученным ключам применим DES в режиме CBC с фиксированным ключом и нулевым начальным значением. Из полученных блоков шифр-текста и формируются подключи RK_i. Для шифрования алгоритмом DES используется фиксированный ключ: $K = 0x1234567890abcdef$ (шестнадцатеричное число).

В DEAL-128 подключи генерируются следующим образом:

$$RK_1 = E_K(K_1),$$

$$RK_2 = E_K(K_2 \oplus RK_1),$$

$$RK_3 = E_K(K_1 \oplus \langle 1 \rangle \oplus RK_2),$$

$$RK_4 = E_K(K_2 \oplus \langle 2 \rangle \oplus RK_3),$$

$$RK_5 = E_K(K_1 \oplus \langle 4 \rangle \oplus RK_4),$$

$$RK_6 = E_K(K_2 \oplus \langle 8 \rangle \oplus RK_5),$$

где $\langle 1 \rangle$ — 64-битное целое число, в котором i — 1-й бит (нумерация с 0) установлен в «1», а остальные — в «0». Например, $\langle 1 \rangle$ может быть представлено как шестнадцатеричное «$0x8000000000000000$».

В DEAL-192 подключи генерируются следующим образом:

$$RK_1 = E_K(K_1),$$

$$RK_2 = E_K(K_2 \oplus RK_1),$$

$$RK_3 = E_K(K_3 \oplus RK_2),$$

$$RK_4 = E_K(\oplus \langle 1 \rangle \oplus RK_3),$$

$$RK_5 = E_K(K_2 \oplus \langle 2 \rangle \oplus RK_4),$$

$$RK_6 = E_K(K_3 \oplus \langle 4 \rangle \oplus RK_5).$$

В DEAL-256 подключи генерируются следующим образом:

$$RK_1 = E_K(K_1),$$

$$RK_2 = E_K(K_2 \oplus RK_1),$$

$$RK_3 = E_K(K_3 \oplus RK_2),$$

$$RK_4 = E_K(K_4 \oplus RK_3),$$

$$RK_5 = E_K(K_1 \oplus \langle 1 \rangle \oplus RK_4),$$

$$RK_6 = E_K(K_2 \oplus \langle 2 \rangle \oplus RK_5),$$

$$RK_7 = E_K(K_3 \oplus \langle 4 \rangle \oplus RK_6),$$

$$RK_8 = E_K(K_4 \oplus \langle 8 \rangle \oplus RK_7).$$

Шифрование и дешифрование сообщения

Пусть $C = E_B(A)$ означает зашифрованное DES значение 64-х битного A на ключе B, и пусть $Y = EA_Z(X)$ означает зашифрование DEAL 128-и битного X на ключе Z. Открытый текст P разделяется на блоки P_i по 128 бит каждый, $P = P_1, P_2, \dots, P_n$. Расписание ключей принимает ключ K и возвращает r ключей DES RK_i, где $i = 1, \dots, r$, как описано ниже. Обозначим X^L и X^R левую и правую части X соответственно. Шифр-текст вычисляется следующим образом. Положим $X_0^L = P_i^L, X_0^R = P_i^R$, и вычислим для $j = 1, \dots, r$.

$$X_j^L = E_{RK_i}(X_{j-1}^L) \oplus X_{j-1}^R,$$

$$X_j^R = X_{j-1}^L.$$

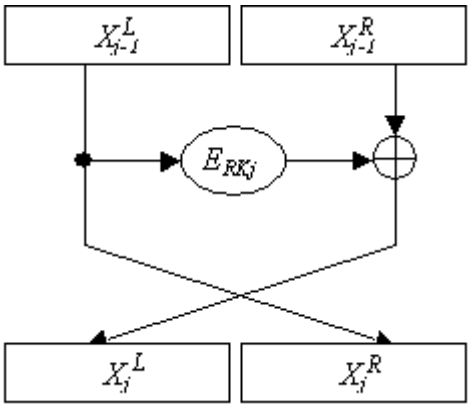

Рисунок 15 Один цикл DEAL

В последнем цикле DEAL половины блока местами меняются.

Практическая часть

Описание работы сервера

Сервер представляет собой консольное приложение, написанное на языке программирования C#. Клиент-серверное взаимодействие написано с помощью опенсорсного фреймворка для удаленного вызова процедур gRPC. Сервер работает на локальном IP-адресе с определенным портом, свободным системой. API включает в себя функции:

1. отправка ключа на сервер;
2. получение ключа с сервера;
3. отправление сообщения/файла на сервер;
4. стриминговое получение очереди сообщений/файлов с сервера;
5. авторизация/выход с сервера клиента.

При подключении клиента к серверу он заносится в список активных пользователей, также при отключении удаляется. Далее сервер ждет запросов от клиента, включающих в себя отправку и получении информации.

Все файлы и сообщения, полученные от клиентов, хранятся и обрабатываются на сервере в зашифрованном виде, чтобы предотвратить кражу информации во время передачи данных. Ключ симметричного алгоритма, которым шифруются файлы и сообщения, также хранится на сервере в зашифрованном ассиметричным алгоритмом виде. Для корректной работы клиентов, они должны иметь правильные ключи дешифрования, иначе симметричный ключ не будет корректно расшифрован, что приведет к неверному прочтению получаемых сообщений и файлов.

Описание работы клиента

Разработки пользовательского приложения велась на языке программирования C# с помощью подсистемы создания графических интерфейсов WPF. Представляет собой многопользовательский чат для общения и передачи файлов.

Стартовый интерфейс представлен на рисунке 16.

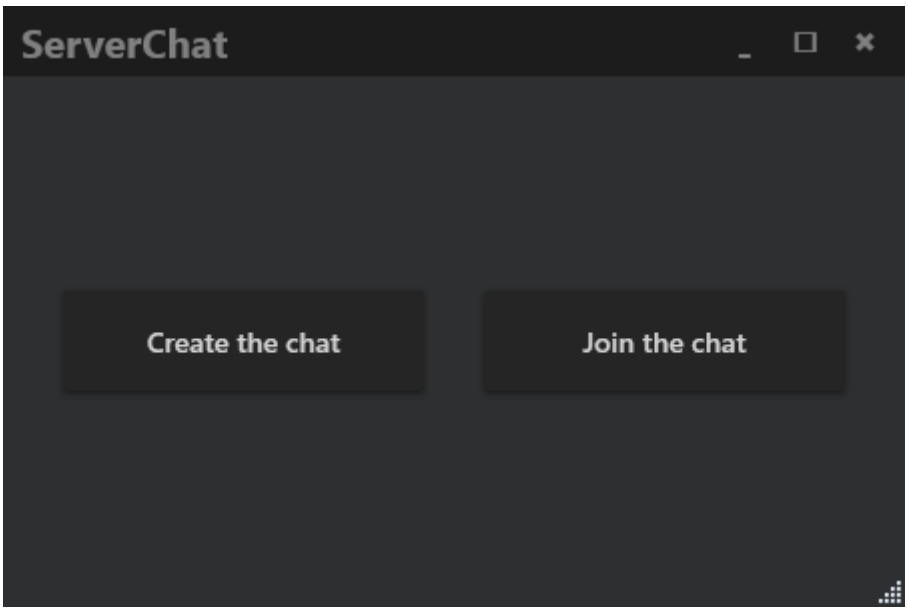

Рисунок 16 Стартовый интерфейс приложения

При старте приложения пользователь может либо создать чат, либо присоединиться к нему. Окна администратора и пользователя представлены на рисунках 17 и 18 соответственно.

Для подключения к серверу клиент должен ввести своё имя в поле «Nickname». Если сервер работает и такого имени нет среди авторизированных пользователей, происходит успешное подключение. Далее администратор должен сгенерировать симметричный ключ шифрования и ключи для ассиметричного шифрования. Генерация происходит асинхронно, чтобы не блокировать основной поток. Для удобства пользователя все ключи имеют числовое представление. Для начала чата необходимо отправить симметричный ключ на сервер, при успешной отправке запускается

асинхронный стриминг получения сообщений/файлов участников чата с сервера.

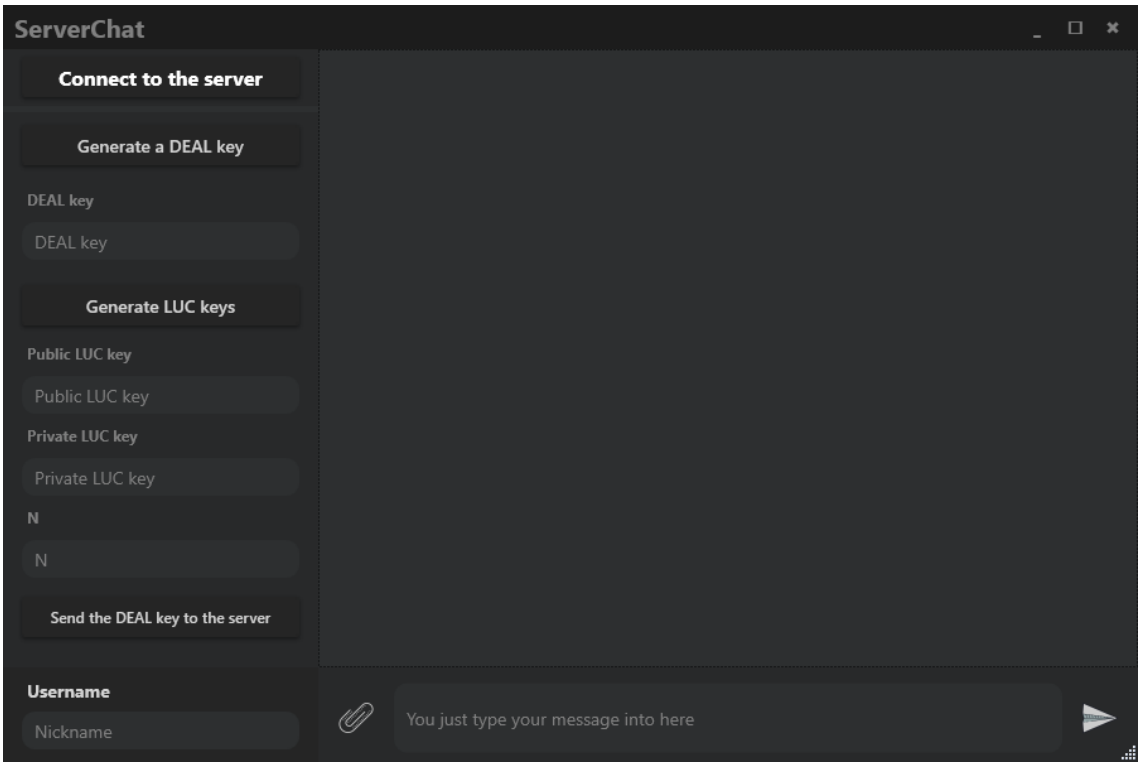

Рисунок 17 Окно для пользователя, создающего чат

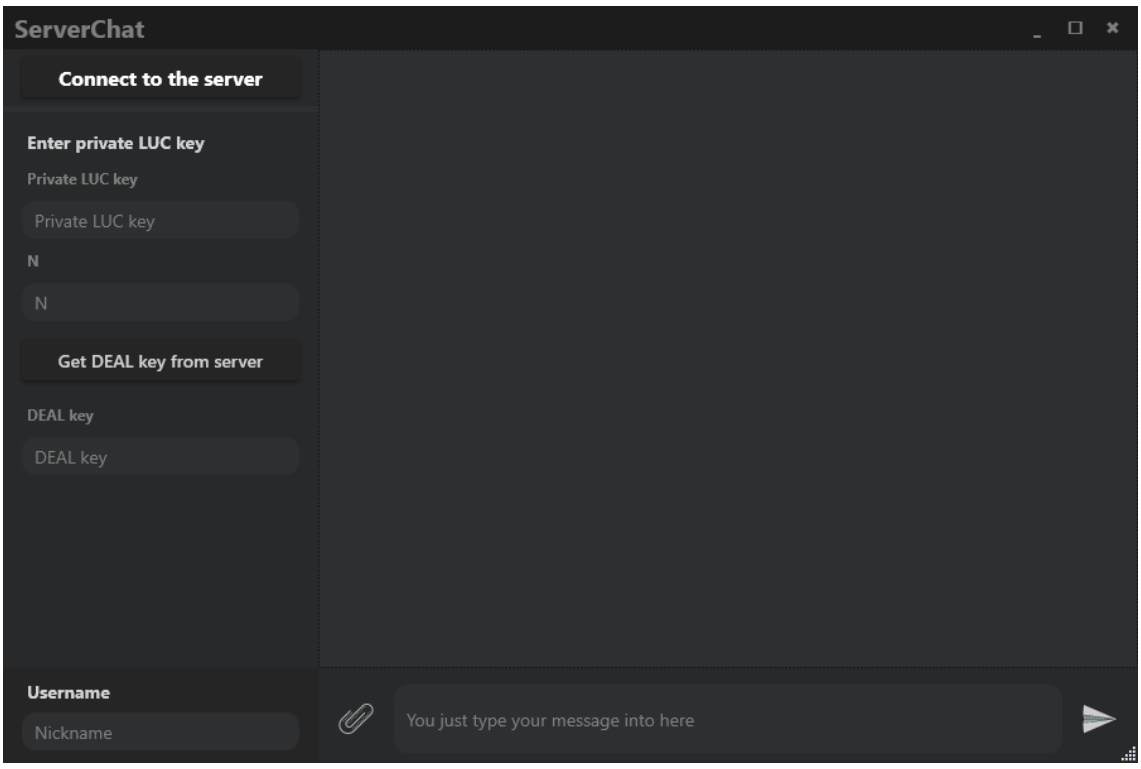

Рисунок 18 Окно пользователя, присоединяющегося к чату

Если клиент хочет подключиться к уже существующему чату, ему также необходимо ввести имя и авторизироваться. Для получения симметричного ключа, ему необходимо вручную ввести приватный ключ и N для его расшифровки. В случае успешного получения ключа также запускается асинхронный стриминг получения сообщений/файлов. Любой клиент может отправлять сообщения/файлы асинхронно, а также их скачивать. При отправке файла нужно нажать на скрепку, далее открывается проводник, где можно выбрать файл, который также асинхронно отправится на сервер, не блокируя основной поток, поэтому в случае, если файл достаточно большой, UI будет активен. Чтобы сохранить файл, нужно нажать на скрепку около названия файла, после чего открывается проводник, где можно выбрать папку для сохранения.

Шифрование и дешифрование сообщений/файлов происходит на стороне клиента, то есть они приходят с сервера и отправляются туда только в зашифрованном виде, что обеспечивает приватность и безопасность общения. Пример общения трёх пользователей приведен на рисунках 19, 20 и 21.

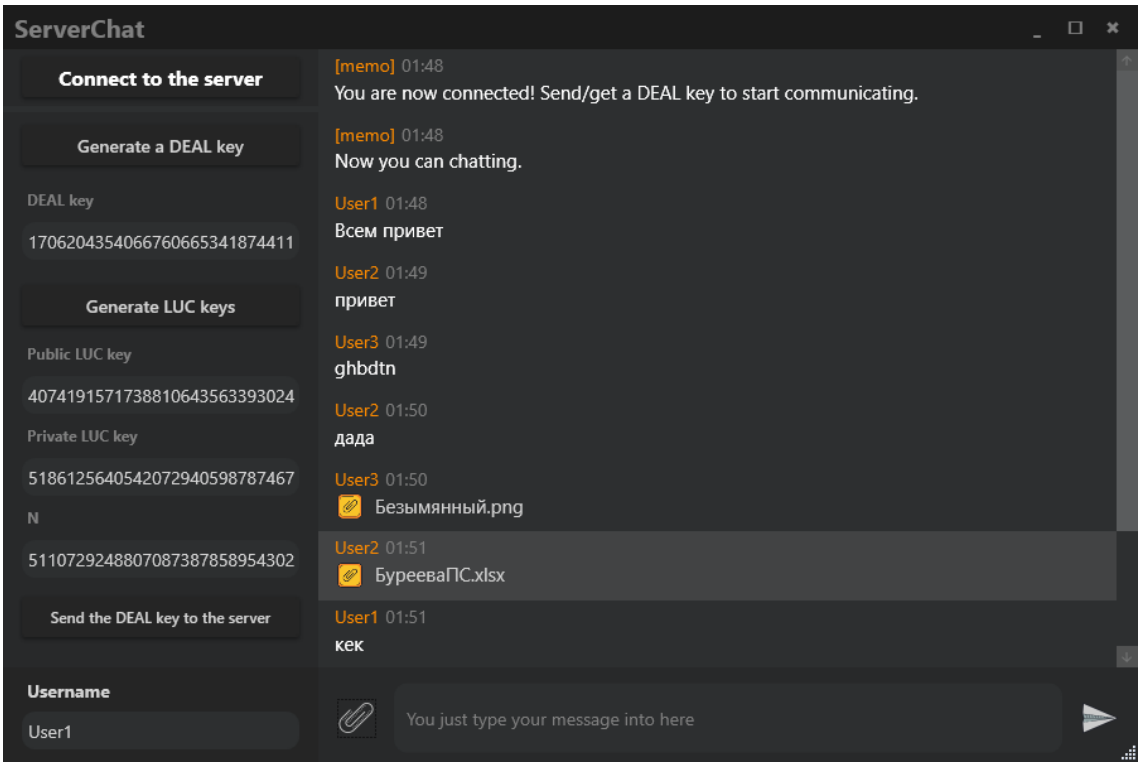

Рисунок 19 Пример общения трёх пользователей

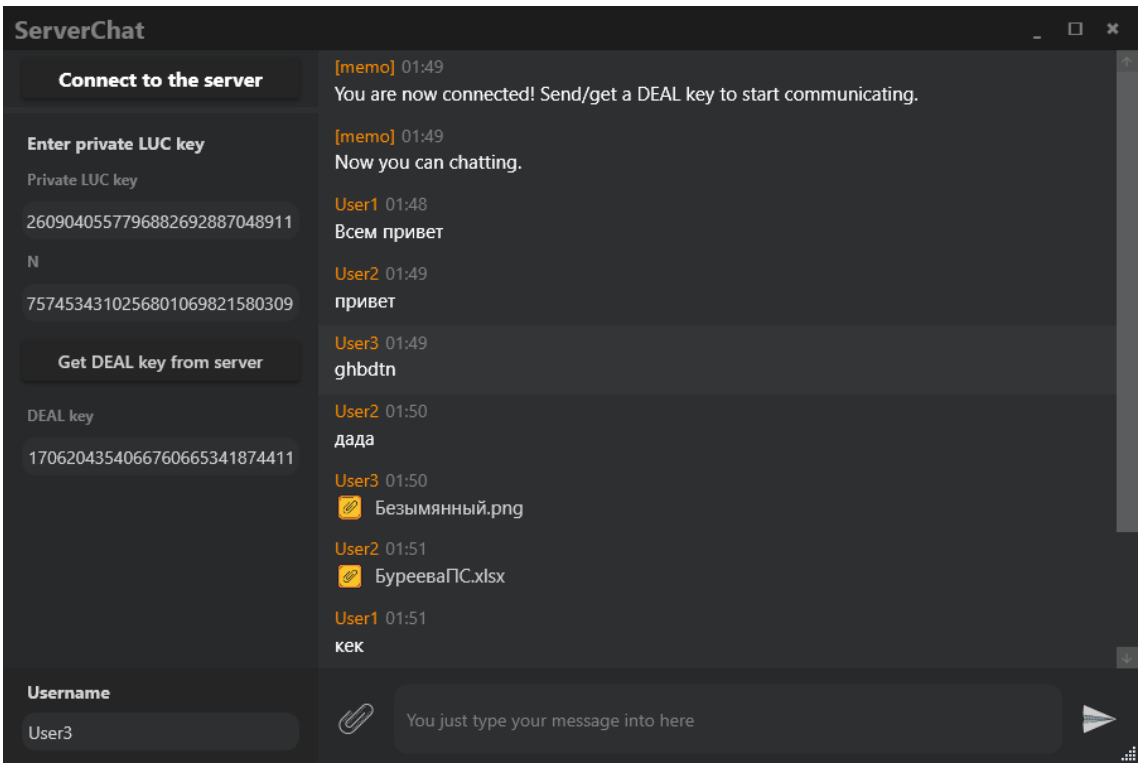

Рисунок 20 Пример общения трёх пользователей

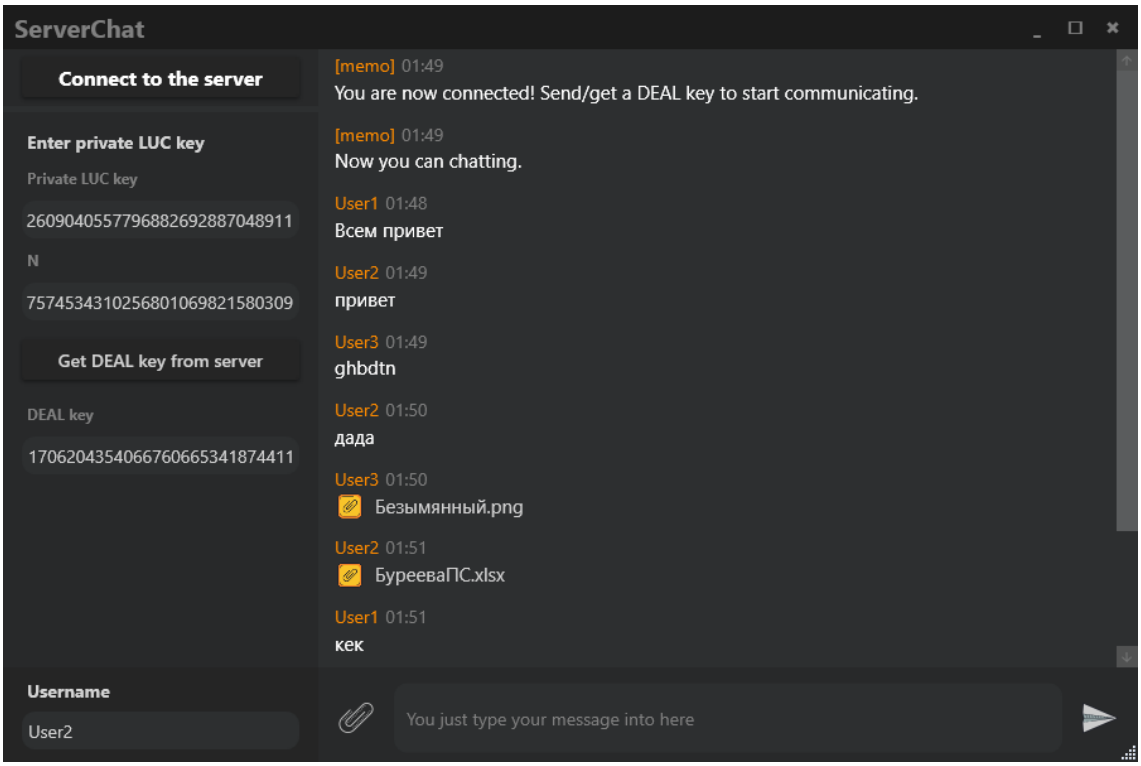

Рисунок 21 Пример общения трёх пользователей

Для выхода с сервера и из чата нужно закрыть окно приложения.

Вывод

В процессе работы были изучены и реализованы на языке программирования C# алгоритмы симметричного шифрования DEAL, DES и ассиметричного шифрования LUC. Для демонстрации работы этих алгоритмов было разработано клиент-серверное приложение многопользовательского чата, состоящее из двух отдельных приложений: сервера и клиента. Сервер служит для обработки действий клиентов и обменом данных. Клиентское приложение на WPF позволяет пользователю генерировать и получать ключи шифрования, а также общаться с другими клиентами, производить обмен сообщениями и файлами.

Все данные, которыми обмениваются пользователи, защищены алгоритмом шифрования, что обеспечивает надежность программы, а также сохранность данных.

Список использованных источников

1. Шнайер Б. Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си. М.: Триумф, 2003. 806 с.
2. Peter Smith, LUC Public-Key Encryption: Dr. Dobb's journal Jan 1993 pp.44-49.
3. LUC // Википедия. [2022]. Дата обновления: 05.09.2022. URL: https://ru.wikipedia.org/?curid=5280426&oldid=125301039.
4. Lhoussain El Fadil. "A Public-Key Cryptosystem Based on Lucas Sequences".
5. Peter Smith, LUC Public-Key Encryption: Dr. Dobb's journal Jan 1993 pp.44-49.
6. Виноградов И. М. Основы теории чисел. — Москва: ГИТТЛ, 1952. — С. 180. — ISBN 5-93972-252-0.
7. Н. Смарт, С.А. Кулешова, Мир Программирования: Криптография. Москва, 2006 г. 529 с.
8. DEAL // Википедия. [2021]. Дата обновления: 20.11.2021. URL: https://ru.wikipedia.org/?curid=2144048&oldid=117992516.
9. Lars R.Knudsen. "DEAL - A 128-bit Block Cipher", February 21, 1998.
10. J. Kelsey, B. Schneier. "Key-Schedule Cryptanalysis of DEAL".
11. Мухачев В.А., Хорошко В.А. Методы практической криптографии. – К.: ООО «Полиграф-Консалтинг», 2005. – 215 с.
12. [Электронный ресурс]: Lucas Sequences in Cryptography. — Режим доступа: http://www.weidai.com/lucas.htm.

Приложение

SymmetricalAlgorithm

IEncryptionTransformation.cs

```
namespace SymmetricalAlgorithm
{
    public interface IEncryptionTransformation
    {
        public byte[] Transform(byte[] block, byte[] roundKey);
    }
}
```

IRoundKeyGenerator.cs

```
namespace SymmetricalAlgorithm
{
    public interface IRoundKeyGenerator
    {
        public byte[][] GenerateRoundKeys(byte[] key);
    }
}
```

ISymmetricalAlgorithm.cs

```
namespace SymmetricalAlgorithm
{
    public interface ISymmetricalAlgorithm
    {
        public int BlockSize { get; }
        public byte[] Encrypt(byte[] block, byte[][] roundKeys);
        public byte[] Decrypt(byte[] block, byte[][] roundKeys);
        public byte[][] GenerateRoundKeys(byte[] key);
    }
}
```

DES

DES.cs

```
using System;
using static Cryptography.Extensions.ByteArrayExtensions;
using SymmetricalAlgorithm;

namespace DES
{
    public class DES : ISymmetricalAlgorithm
    {
        private const int BlockSizeConst = 8;

        private readonly ISymmetricalAlgorithm _feistelNetwork =
            new    FeistelNetwork(new    DESRoundKeyGenerator(),    new
DESFeistelFunction(), BlockSizeConst, 16);

        public int BlockSize => BlockSizeConst;
```

```
        public byte[] Encrypt(byte[] block, byte[][] roundKeys)
        {
            if (block.Length != BlockSizeConst)
            {
                throw new ArgumentException($"Block length must be equal
to {BlockSizeConst}.");
            }

            block = Permutation(block, Tables.InitialPermutation);
            block = _feistelNetwork.Encrypt(block, roundKeys);

            return Permutation(block, Tables.FinalPermutation);
        }

        public byte[] Decrypt(byte[] block, byte[][] roundKeys)
        {
            if (block.Length != BlockSizeConst)
            {
                throw new ArgumentException($"Block length must be equal
to {BlockSizeConst}.");
            }

            block = Permutation(block, Tables.InitialPermutation);
            block = _feistelNetwork.Decrypt(block, roundKeys);

            return Permutation(block, Tables.FinalPermutation);
        }

        public byte[][] GenerateRoundKeys(byte[] key)
        {
            return _feistelNetwork.GenerateRoundKeys(key);
        }

        public byte[] GenerateIV()
        {
            return GenerateRandomByteArray(BlockSizeConst);
        }
    }
}
```

DESFeistelFunction.cs

```
using SymmetricalAlgorithm;
using static Cryptography.Extensions.ByteArrayExtensions;

namespace DES
{
    public class DESFeistelFunction : IEncryptionTransformation
    {
        public byte[] Transform(byte[] block, byte[] roundKey)
        {
            var expandingPermutation = Permutation(block,
Tables.ExpandingPermutation);
            var xor = expandingPermutation.Xor(roundKey);
            var sBlock = PermutationSBlock(xor, Tables.SBlocks);

            return Permutation(sBlock, Tables.PBlock);
```

```
        }
    }
}
```

DESRoundKeyGenerator.cs

```
using System;
using System.Collections;
using SymmetricalAlgorithm;
using static Cryptography.Extensions.BitArrayExtensions;
using static Cryptography.Extensions.ByteArrayExtensions;


namespace DES
{
    public class DESRoundKeyGenerator : IRoundKeyGenerator
    {
        public byte[][] GenerateRoundKeys(byte[] key)
        {
            if (key.Length != 8)
                throw    new    ArgumentOutOfRangeException(nameof(key),
key.Length,
                    "Invalid key. The allowed key size is 8 byte.");
            var roundKeys = new byte[16][];
            var keybits = new BitArray(key);
            var        currentC        =        BitsPermutation(keybits,
Tables.KeyPermutationC);
            var        currentD        =        BitsPermutation(keybits,
Tables.KeyPermutationD);

            for (var i = 0; i < 16; ++i)
            {
                currentC = currentC.LeftShift(Tables.KeyShift[i]);
                currentD = currentD.LeftShift(Tables.KeyShift[i]);
                var currentKey = currentC.BitsConcat(currentD);
                roundKeys[i]                                                =
BitArrayToByteArray(BitsPermutation(currentKey,
Tables.KeyCompressionPermutation));
            }

            return roundKeys;
        }
    }
}
```

FeistelNetwork.cs

```
using System;
using System.Linq;
using Cryptography.Extensions;
using SymmetricalAlgorithm;

namespace DES
{
    public class FeistelNetwork : ISymmetricalAlgorithm
    {
```

```
        private readonly IRoundKeyGenerator _roundKeysGenerator;
        private readonly IEncryptionTransformation _feistelFunction;
        private readonly int _blockSize;
        private readonly int _numberOfRounds;
        public  FeistelNetwork(IRoundKeyGenerator  roundKeysGenerator,
IEncryptionTransformation feistelFunction,
            int blockSize, int numberOfRounds)
        {
            _roundKeysGenerator = roundKeysGenerator;
            _feistelFunction = feistelFunction;
            _blockSize = blockSize;
            _numberOfRounds = numberOfRounds;
        }

        public int BlockSize => _blockSize;


        public byte[] Encrypt(byte[] block, byte[][] roundKeys)
        {
            if (block.Length != _blockSize)
            {
                throw new ArgumentException($"Block length must be equal
to {_blockSize}.");
            }

            var left = block.Take(_blockSize / 2).ToArray();
            var right = block.Skip(_blockSize / 2).ToArray();
            for (var i = 0; i < _numberOfRounds; i++)
            {
                var tmp = right;
                right   =   left.Xor(_feistelFunction.Transform(right,
roundKeys[i]));
                left = tmp;
            }

            return left.Concat(right).ToArray();
        }

        public byte[] Decrypt(byte[] block, byte[][] roundKeys)
        {
            if (block.Length != _blockSize)
            {
                throw new ArgumentException($"Block length must be equal
to {_blockSize}.");
            }

            var left = block.Take(_blockSize / 2).ToArray();
            var right = block.Skip(_blockSize / 2).ToArray();
            for (var i = _numberOfRounds - 1; i >= 0; i--)
            {
                var tmp = left;
                left    =    right.Xor(_feistelFunction.Transform(left,
roundKeys[i]));
                right = tmp;
            }

            return left.Concat(right).ToArray();
        }
```

```
        public byte[][] GenerateRoundKeys(byte[] key)
        {
            return _roundKeysGenerator.GenerateRoundKeys(key);
        }
    }
}
```

Tables.cs

```
namespace DES
{
    public static class Tables
    {
        public static readonly byte[] InitialPermutation =
        {
            58, 50, 42, 34, 26, 18, 10, 2, 60, 52, 44, 36, 28, 20, 12,
4,
            62, 54, 46, 38, 30, 22, 14, 6, 64, 56, 48, 40, 32, 24, 16,
8,
            57, 49, 41, 33, 25, 17, 9, 1, 59, 51, 43, 35, 27, 19, 11,
3,
            61, 53, 45, 37, 29, 21, 13, 5, 63, 55, 47, 39, 31, 23, 15,
7
        };

        public static readonly byte[] KeyPermutationC =
        {
            57, 49, 41, 33, 25, 17, 9, 1, 58, 50, 42, 34, 26, 18,
            10, 2, 59, 51, 43, 35, 27, 19, 11, 3, 60, 52, 44, 36
        };

        public static readonly byte[] KeyPermutationD =
        {
            63, 55, 47, 39, 31, 23, 15, 7, 62, 54, 46, 38, 30, 22,
            14, 6, 61, 53, 45, 37, 29, 21, 13, 5, 28, 20, 12, 4
        };

        public static readonly byte[] KeyShift =
        {
            1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1
        };

        public static readonly byte[] KeyCompressionPermutation =
        {
            14, 17, 11, 24, 1, 5, 3, 28, 15, 6, 21, 10, 23, 19, 12, 4,
            26, 8, 16, 7, 27, 20, 13, 2, 41, 52, 31, 37, 47, 55, 30, 40,
            51, 45, 33, 48, 44, 49, 39, 56, 34, 53, 46, 42, 50, 36, 29,
32
        };

        public static readonly byte[] ExpandingPermutation =
        {
            32, 1, 2, 3, 4, 5, 4, 5, 6, 7, 8, 9,
            8, 9, 10, 11, 12, 13, 12, 13, 14, 15, 16, 17,
            16, 17, 18, 19, 20, 21, 20, 21, 22, 23, 24, 25,
            24, 25, 26, 27, 28, 29, 28, 29, 30, 31, 32, 1
        };
```

```
public static readonly byte[,,] SBlocks =
{
    {
        {14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7},
        {0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8},
        {4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0},
        {15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13}
    },
    {
        {15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10},
        {3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5},
        {0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15},
        {13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9}
    },
    {
        {10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8},
        {13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1},
        {13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7},
        {1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 13}
    },
    {
        {7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15},
        {13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9},
        {10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4},
        {3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 6, 11, 12, 7, 2, 14}
    },
    {
        {2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9},
        {14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6},
        {4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14},
        {11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3}
    },
    {
        {12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11},
        {10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8},
        {9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6},
        {4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13}
    },
    {
        {4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1},
        {13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6},
        {1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2},
        {6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12}
    },
    {
        {13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7},
        {1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2},
        {7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8},
        {2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11}
    }
};

public static readonly byte[] PBlock =
{
    16, 7, 20, 21, 29, 12, 28, 17,
    1, 15, 23, 26, 5, 18, 31, 10,
    2, 8, 24, 14, 32, 27, 3, 9,
```

```
            19, 13, 30, 6, 22, 11, 4, 25
        };

        public static readonly byte[] FinalPermutation =
        {
            40, 8, 48, 16, 56, 24, 64, 32, 39, 7, 47, 15, 55, 23, 63,
31,
            38, 6, 46, 14, 54, 22, 62, 30, 37, 5, 45, 13, 53, 21, 61,
29,
            36, 4, 44, 12, 52, 20, 60, 28, 35, 3, 43, 11, 51, 19, 59,
27,
            34, 2, 42, 10, 50, 18, 58, 26, 33, 1, 41, 9, 49, 17, 57, 25
        };
    }
}
```

DEAL

DEAL.cs

```
using System;
using System.Linq;
using DES;
using static Cryptography.Extensions.ByteArrayExtensions;
using SymmetricalAlgorithm;

namespace DEAL
{
    public class DEAL : ISymmetricalAlgorithm
    {
        private static readonly DES.DES Des = new DES.DES();
        private const int BlockSizeConst = 16;
        private readonly ISymmetricalAlgorithm _feistelNetwork;
        private byte[] _desIv;

        public int BlockSize => BlockSizeConst;

        public DEAL(byte[] key, byte[] desIv)
        {
            var numberOfRounds = key.Length == 32 ? 8 : 6;
            _desIv = desIv;
            _feistelNetwork =
                new FeistelNetwork(
                    new DEALRoundKeysGenerator(Des, _desIv),
                    new DEALFeistelFunction(Des, _desIv),
                    BlockSizeConst,
                    numberOfRounds);
        }

        public byte[] Encrypt(byte[] block, byte[][] roundKeys)
        {
            if (block.Length != BlockSizeConst)
            {
                throw new ArgumentException($"Block length must be equal
to {BlockSizeConst}.");
            }

            var result = new byte[BlockSizeConst];
```

```
            for (var i = 0; i < result.Length / BlockSizeConst; i++)
            {
                var currentBlock = new byte[BlockSizeConst];
                Array.Copy(block, i * BlockSizeConst, currentBlock, 0,
BlockSizeConst);
                var (left, right) = (currentBlock.Take(BlockSizeConst /
2).ToArray(),
                    currentBlock.Skip(BlockSizeConst / 2).ToArray());

                currentBlock                                          =
_feistelNetwork.Encrypt(right.Concat(left).ToArray(), roundKeys);

                (left,  right)  =  (currentBlock.Take(BlockSizeConst  /
2).ToArray(),
                    currentBlock.Skip(BlockSizeConst / 2).ToArray());
                Array.Copy(right.Concat(left).ToArray(), 0, result, i *
BlockSizeConst, BlockSizeConst);
            }

            return result;
        }

        public byte[] Decrypt(byte[] block, byte[][] roundKeys)
        {
            if (block.Length != BlockSizeConst)
            {
                throw new ArgumentException($"Block length must be equal
to {BlockSizeConst}.");
            }

            var result = new byte[BlockSizeConst];

            for (var i = 0; i < result.Length / BlockSizeConst; i++)
            {
                var currentBlock = new byte[BlockSizeConst];
                Array.Copy(block, i * BlockSizeConst, currentBlock, 0,
BlockSizeConst);
                var (left, right) = (currentBlock.Take(BlockSizeConst /
2).ToArray(),
                    currentBlock.Skip(BlockSizeConst / 2).ToArray());

                currentBlock                                          =
_feistelNetwork.Decrypt(right.Concat(left).ToArray(), roundKeys);

                (left,  right)  =  (currentBlock.Take(BlockSizeConst  /
2).ToArray(),
                    currentBlock.Skip(BlockSizeConst / 2).ToArray());
                Array.Copy(right.Concat(left).ToArray(), 0, result, i *
BlockSizeConst, BlockSizeConst);
            }

            return result;
        }

        public byte[][] GenerateRoundKeys(byte[] key)
        {
            return _feistelNetwork.GenerateRoundKeys(key);
```

```
        }

        public byte[] GenerateIV()
        {
            return GenerateRandomByteArray(BlockSize);
        }
    }
}
```

DEALFeistelFunction.cs

```
using CipherContext;
using SymmetricalAlgorithm;

namespace DEAL
{
    public class DEALFeistelFunction : IEncryptionTransformation
    {
        private readonly DES.DES _des;
        private readonly byte[] _iv;

        public DEALFeistelFunction(DES.DES des, byte[] iv)
        {
            _des = des;
            _iv = iv;
        }

        public byte[] Transform(byte[] block, byte[] roundKey)
        {
            var encoder = new CipherContext.CipherContext(_des,
roundKey, _iv)
            {
                EncryptionMode                                          =
CipherContext.CipherContext.EncryptionModeList.CBC
            };

            return encoder.Encrypt(block, encoder.GenerateRoundKeys());
        }
    }
}
```

DEALRoundKeysGenerator.cs

```
using System;
using System.Linq;
using CipherContext;
using Cryptography.Extensions;
using SymmetricalAlgorithm;

namespace DEAL
{
    public class DEALRoundKeysGenerator : IRoundKeyGenerator
    {
        private readonly DES.DES _des;
        private const long ConstKey = 0x1234567890abcdef;
        private readonly byte[][] _bit64OriginalString;
        private byte[][] _roundKeys;
```

```
        private readonly byte[] _iv;

        public DEALRoundKeysGenerator(DES.DES des, byte[] iv)
        {
            _bit64OriginalString = new ulong[]
            {
                0x8000000000000000,
                0x0800000000000000,
                0x0008000000000000,
                0x0000000800000000
            }.Select(BitConverter.GetBytes).ToArray();
            _des = des;
            _iv = iv;
        }

        public byte[][] GenerateRoundKeys(byte[] key)
        {
            var encoder = new CipherContext.CipherContext(_des,
BitConverter.GetBytes(ConstKey), _iv)
            {
                EncryptionMode                                          =
CipherContext.CipherContext.EncryptionModeList.CBC
            };
            var desRoundKeys = encoder.GenerateRoundKeys();
            switch (key.Length)
            {
                case 16:
                {
                    _roundKeys = new byte[6][];

                    var k = new[]
                    {
                        key.Take(key.Length / 2).ToArray(),
                        key.Skip(key.Length / 2).ToArray()
                    };

                    _roundKeys[0]       =       encoder.Encrypt(k[0],
desRoundKeys);
                    _roundKeys[1]                                       =
encoder.Encrypt(k[1].Xor(_roundKeys[0]), desRoundKeys);

                    for (var i = 0; i < 4; i++)
                    {
                        _roundKeys[i + 2] = encoder.Encrypt(k[i % 2]
                            .Xor(_bit64OriginalString[i])
                            .Xor(_roundKeys[i + 1]), desRoundKeys);
                    }

                    break;
                }

                case 24:
                {
                    _roundKeys = new byte[6][];

                    var k = new[]
                    {
                        key.Take(key.Length / 3).ToArray(),
```

```
                        key.Skip(key.Length / 3).Take(key.Length / 3).ToArray(),
                        key.Skip(key.Length / 3 * 2).ToArray()
                    };

                    _roundKeys[0] = encoder.Encrypt(k[0], desRoundKeys);

                    for (var i = 0; i < 2; i++)
                    {
                        _roundKeys[i + 1] = encoder.Encrypt(k[i % 3].Xor(_roundKeys[i]), desRoundKeys);
                    }

                    for (var i = 0; i < 3; i++)
                    {
                        _roundKeys[i + 3] = encoder.Encrypt(k[i % 3]
                            .Xor(_bit64OriginalString[i])
                            .Xor(_roundKeys[i + 1]), desRoundKeys);
                    }

                    break;
                }
                case 32:
                {
                    _roundKeys = new byte[8][];

                    var k = new[]
                    {
                        key.Take(key.Length / 4).ToArray(),
                        key.Skip(key.Length / 4).Take(key.Length / 4).ToArray(),
                        key.Skip(key.Length / 4 * 2).Take(key.Length / 4).ToArray(),
                        key.Skip(key.Length / 4 * 3).ToArray()
                    };

                    _roundKeys[0] = encoder.Encrypt(k[0], desRoundKeys);

                    for (var i = 0; i < 3; i++)
                    {
                        _roundKeys[i + 1] = encoder.Encrypt(k[i % 4].Xor(_roundKeys[i]), desRoundKeys);
                    }

                    for (var i = 0; i < 4; i++)
                    {
                        _roundKeys[i + 4] = encoder.Encrypt(k[i % 4]
                            .Xor(_bit64OriginalString[i])
                            .Xor(_roundKeys[i + 1]), desRoundKeys);
                    }

                    break;
                }
                default:
                    throw new ArgumentOutOfRangeException(nameof(key), key.Length,
```

```
                        "Invalid key. The allowed key size is 16/24/32
byte");
            }

            return _roundKeys;
        }
    }
}
```

LUC

KeyGenerator.cs

```
using System.Numerics;
using System.Threading;
using System.Threading.Tasks;
using static Cryptography.Extensions.BigIntegerExtensions;

namespace LUC
{
    public class KeyGenerator
    {
        private PrimeNumbers _primeNumbers;
        private readonly BigInteger _message;
        public LucKey PublicKey { get; set; }

        public LucKey PrivateKey { get; set; }

        public KeyGenerator(BigInteger message)
        {
            _message = message;
        }

        public void GenerateKey()
        {
            var                prime                =                new
PrimeNumbersGenerator(_message.GetByteCount(),
PrimeNumberTest.MillerRabin, 0.75);
            _primeNumbers = prime.GeneratePrime();

            BigInteger e = GetE();
            BigInteger D = _message * _message - 4;
            BigInteger   S   =   Lcm(_primeNumbers.P   -   Legendre(D,
_primeNumbers.P), _primeNumbers.Q - Legendre(D, _primeNumbers.Q));
            BigInteger d = MultiplicativeInverseModulo(e, S);

            PublicKey = new LucKey(e, _primeNumbers.N);
            PrivateKey = new LucKey(d, _primeNumbers.N);
        }

        private BigInteger GetE()
        {
            BigInteger number = (_primeNumbers.P - 1) * (_primeNumbers.Q
- 1) *
                                (_primeNumbers.P + 1) * (_primeNumbers.Q
+ 1);
```

```
            BigInteger e;
            do
            {
                e = GenerateRandomBigInteger(2, _primeNumbers.N);
            } while (BigInteger.GreatestCommonDivisor(e, number) != 1);

            return e;
        }

        public Task GenerateKeyAsync(CancellationToken token = default)
        {
            token.ThrowIfCancellationRequested();
            return Task.Run(() => GenerateKey(), token);
        }
    }
}
```

LUC.cs

```
using System.Numerics;
using System.Threading;
using System.Threading.Tasks;
using static Cryptography.Extensions.BigIntegerExtensions;

namespace LUC
{
    public class LUC
    {
        public BigInteger Encrypt(BigInteger message, LucKey publicKey)
            => LucasSequencesMod(message, publicKey.Key, publicKey.N);

        public BigInteger Decrypt(BigInteger message, LucKey privateKey)
            =>     LucasSequencesMod(message,           privateKey.Key,
privateKey.N);

        public Task<BigInteger> EncryptAsync(BigInteger message, LucKey
publicKey, CancellationToken token = default)
        {
            token.ThrowIfCancellationRequested();
            return Task.Run(() => Encrypt(message, publicKey), token);
        }
        public Task<BigInteger> DecryptAsync(BigInteger message, LucKey
privateKe, CancellationToken token = default)
        {
            token.ThrowIfCancellationRequested();
            return Task.Run(() => Decrypt(message, privateKe), token);
        }
    }
}
```

LucKey.cs

```
using System.Numerics;

namespace LUC
{
    public struct LucKey
    {
```

```
        public BigInteger Key { get;}

        public BigInteger N { get;}

        public LucKey(BigInteger key, BigInteger n)
        {
            Key = key;
            N = n;
        }
    }
}
```

PrimalityTests.cs

```
using System;
using System.Numerics;
using static Cryptography.Extensions.BigIntegerExtensions;

namespace LUC
{
    public static class PrimalityTests
    {
        public static bool FermatPrimalityTest(BigInteger number, double
probability)
        {
            if (number == (BigInteger)1) return false;
            for (var k = 1; 1 - Math.Pow(2, -k) <= probability; k++)
            {
                BigInteger a = GenerateRandomBigInteger(1, number);
                if   (BigInteger.ModPow(a,   number   -   BigInteger.One,
number) != 1)
                    return false;
            }

            return true;
        }
        public   static   bool   SolovayStrassenPrimalityTest(BigInteger
number, double probability)
        {
            if (number == BigInteger.One) return false;
            for (var k = 1; 1 - Math.Pow(2, -k) <= probability; k++)
            {
                BigInteger a = GenerateRandomBigInteger(1, number);
                if   (BigInteger.GreatestCommonDivisor(a,   number)   >
BigInteger.One)
                    return false;
                if (BigInteger.ModPow(a, (number - BigInteger.One) / 2,
number) != (BigInteger) Jacobi(a, number))
                    return false;
            }

            return true;
        }
        public static bool MillerRabinPrimalityTest(BigInteger number,
double probability)
        {
            if (number == (BigInteger)3) return true;
            if (number == (BigInteger)2) return true;
```

```
            if (number < (BigInteger)2) return false;

            var a = number - BigInteger.One;
            int s = 0;
            while (BigInteger.GreatestCommonDivisor(a, 2) == 0)
            {
                a /= 2;
                s++;
            }

            for (int k = 1; 1 - Math.Pow(4, -k) <= probability; k++)
            {
                var b = GenerateRandomBigInteger(1, a);
                var x = BigInteger.ModPow(b, a, number);

                if (BigInteger.Compare(x, 1) == 0 ||
BigInteger.Compare(x, a) == 0)
                    continue;

                for (var i = 0; i < s - 1; i++)
                {
                    x = BigInteger.ModPow(x, 2, number);
                    if (x == BigInteger.One)
                        return false;
                    if (x == a)
                        break;
                }
                if (x != a)
                    return false;
            }

            return true;
        }
    }
}
```

PrimeNumbers.cs

```
using System.Numerics;

namespace LUC
{
    public struct PrimeNumbers
    {
        public BigInteger P { get;}

        public BigInteger Q { get;}

        public BigInteger N { get;}

        public PrimeNumbers(BigInteger p, BigInteger q)
        {
            P = p;
            Q = q;
            N = p * q;
        }

    }
```

```
}
```

PrimeNumbersGenerator.cs

```
using System.Numerics;
using System.Security.Cryptography;

namespace LUC
{
    public class PrimeNumbersGenerator
    {
        private readonly long _length;
        private readonly PrimeNumberTest _test;
        private readonly double _probability;

        public PrimeNumbersGenerator(long length, PrimeNumberTest test,
double probability)
        {
            _length = length;
            _test = test;
            _probability = probability;
        }
        public PrimeNumbers GeneratePrime()
        {
            BigInteger N, p, q;
            do
            {
                p = RandomPrimeNumberGenerator();
                q = RandomPrimeNumberGenerator();
                N = p * q;

            } while (BigInteger.Compare(N.GetBitLength(), _length) < 0);

            return new PrimeNumbers(p, q);
        }
        private BigInteger RandomPrimeNumberGenerator()
        {
            var random = RandomNumberGenerator.Create();
            var buffer = new byte[_length / 2 + 1];
            while (true)
            {
                random.GetBytes(buffer);
                buffer[^1] &= 0b01111111;
                var primeNumber = new BigInteger(buffer);

                if ((BigInteger)2 > primeNumber) continue;

                switch (_test)
                {
                    case PrimeNumberTest.Fermat:
                        if
(PrimalityTests.FermatPrimalityTest(primeNumber, _probability))
                            return primeNumber;
                        break;
                    case PrimeNumberTest.SolovayStrassen:
                        if
(PrimalityTests.SolovayStrassenPrimalityTest(primeNumber,
_probability))
```

```
                            return primeNumber;
                        break;
                    case PrimeNumberTest.MillerRabin:
                        if
(PrimalityTests.MillerRabinPrimalityTest(primeNumber, _probability))
                            return primeNumber;
                        break;
                }
            }
        }
    }
}
```

PrimeNumbersTest.cs

```
namespace LUC
{
    public enum PrimeNumberTest
    {
        SolovayStrassen,
        Fermat,
        MillerRabin
    }
}
```

Cryptography.Extensions

ByteArrayExtensions.cs

```
using System;
using System.Collections;
using System.Security.Cryptography;

namespace Cryptography.Extensions
{
    public static class ByteArrayExtensions
    {
        public   static   byte[]   Permutation(byte[]   block,   byte[]
permutationTable)
        {
            var bitsblock = new BitArray(block);

            var changed = new BitArray(permutationTable.Length);
            for (var i = 0; i < permutationTable.Length; i++)
            {
                changed[i] = bitsblock[permutationTable[i] - 1];
            }

            return BitArrayToByteArray(changed);
        }

        public static byte[] BitArrayToByteArray(BitArray bits)
        {
            var bytes = new byte[(bits.Length - 1) / 8 + 1];
            bits.CopyTo(bytes, 0);
            return bytes;
        }
```

```
        public static byte[] PermutationSBlock(byte[] block, byte[,]
permutationTable)
        {
            var number = BitConverter.ToUInt32(block, 0);
            ulong result = 0;
            for (var i = 0; i < 8; i++)
            {
                var B = (number >> (i * 6)) & ((uint) 1 << 6) - 1;
                var a = ((B >> 5) << 1) | (B & 1);
                var b = (B >> 1) & 0b1111;

                B = permutationTable[i, a, b];
                result |= B << i * 4;
            }

            return BitConverter.GetBytes(result);
        }

        public static byte[] Xor(this byte[] left, byte[] right)
        {
            if (left is null)
            {
                throw new ArgumentNullException(nameof(left));
            }

            if (right is null)
            {
                throw new ArgumentNullException(nameof(right));
            }

            if (left.Length != right.Length)
            {
                throw  new  ArgumentException("Arrays  lengths  must  be
equal.");
            }

            for (var i = 0; i < left.Length; i++)
            {
                left[i] = (byte) (left[i] ^ right[i]);
            }

            return left;
        }

        public static byte[] PaddingPKCs7(byte[] block, int blockSize)
        {
            if (block.Length == 8 && blockSize == 8) return block;
            var  addition  =  (byte)  (blockSize  -  block.Length  %
blockSize);
            var paddedBlock = new byte[block.Length + addition];
            Array.Copy(block, paddedBlock, block.Length);
            Array.Fill(paddedBlock, addition, block.Length, addition);

            return paddedBlock;
        }
```

```
        public static byte[] ByteArrayAdditionByModulo2PowN(byte[]
left, byte[] right)
        {
            BitArray num1, num2;

            if (left.Length > right.Length)
            {
                num1 = new BitArray(left);
                num2 = new BitArray(right);
            }
            else
            {
                num1 = new BitArray(right);
                num2 = new BitArray(left);
            }

            var max = num1.Count;
            var min = num2.Count;
            var result = new BitArray(num1);
            var tmp = false;
            for (var i = max / 8; i > (max - min) / 8; i--)
            {
                for (var j = 8; j > 0; j--)
                {
                    var w = i * 8 - j;
                    var t = (i - (max - min) / 8) * 8 - j;
                    if (num1[w] == num2[t] && num2[t] == tmp)
                    {
                        result[i * 8 - 1] = tmp;
                    }
                    else if (!(num1[w] ^ num2[t] ^ tmp))
                    {
                        result[w] = false;
                        tmp = true;
                    }
                    else if (num1[w] ^ num2[t] ^ tmp)
                    {
                        result[w] = true;
                        tmp = false;
                    }
                }
            }

            if (tmp)
            {
                for (var i = (max - min) / 8; i > 0; i--)
                {
                    for (var j = 8; j > 0; j--)
                    {
                        var w = i * 8 - j;
                        if (num1[w])
                        {
                            result[w] = false;
                        }
                        else
                        {
                            result[w] = true;
                            return BitArrayToByteArray(result);
```

```
                    }
                }
            }
        }

        return BitArrayToByteArray(result);
    }

    public static byte[] GenerateRandomByteArray(int size)
    {
        var array = new byte[size];
        var random = RandomNumberGenerator.Create();
        random.GetBytes(array);
        //делаем число неотр
        array[^1] &= 0b01111111;
        return array;
    }
}
}
```

BitArrayExtensions.cs

```
using System.Collections;

namespace Cryptography.Extensions
{
    public static class BitArrayExtensions
    {
        public static BitArray BitsPermutation(BitArray block, byte[]
permutationTable)
        {
            var changed = new BitArray(permutationTable.Length);
            for (var i = 0; i < permutationTable.Length; i++)
            {
                changed[i] = block[permutationTable[i] - 1];
            }

            return changed;
        }

        public static BitArray BitsConcat(this BitArray left, BitArray
right)
        {
            var changed = new BitArray(left.Count + right.Count);
            for (var i = 0; i < left.Count; i++)
            {
                changed[i] = left[i];
            }

            for (var i = 0; i < right.Count; i++)
            {
                changed[i + left.Count] = right[i];
            }

            return changed;
        }
    }
}
```

```
using System;
using System.Linq;
using System.Numerics;
using System.Security.Cryptography;

namespace Cryptography.Extensions
{
    public static class BigIntegerExtensions
    {
        public static BigInteger Legendre(BigInteger a, BigInteger p)
        {
            if (p < (BigInteger)2)
                throw new ArgumentOutOfRangeException(nameof(p), "P
must be >= 2");

            if (a == BigInteger.Zero || a == BigInteger.One)
                return a;

            BigInteger result;
            if (BigInteger.Remainder(a, 2) == BigInteger.Zero)
            {
                result = Legendre(a / 2, p);
                if (((p * p - BigInteger.One) & 8) != BigInteger.Zero)
                    result = BigInteger.Negate(result);
            }
            else
            {
                result = Legendre(p % a, a);
                if (((a - BigInteger.One) * (p - 1) & 4) !=
BigInteger.Zero)
                    result = BigInteger.Negate(result);
            }

            return result;
        }
        public static BigInteger Lcm(BigInteger a, BigInteger b)
            => (a * b) / BigInteger.GreatestCommonDivisor(a, b);

        public static BigInteger MultiplicativeInverseModulo(BigInteger
a, BigInteger m)
        {
            BigInteger g = Gcd(a, m, out BigInteger x, out _);
            if (g != BigInteger.One) return BigInteger.Zero;

            return (x % m + m) % m;
        }

        private static BigInteger Gcd(BigInteger a, BigInteger m, out
BigInteger x, out BigInteger y)
        {
            if (a == BigInteger.Zero)
            {
                x = BigInteger.Zero;
                y = BigInteger.One;
```

```
                return m;
            }

            BigInteger d = Gcd(m % a, a, out BigInteger x1, out
BigInteger y1);

            x = y1 - (m / a) * x1;
            y = x1;

            return d;
        }

        public static BigInteger GenerateRandomBigInteger(BigInteger
left, BigInteger right)
        {
            var random = RandomNumberGenerator.Create();
            var buffer = new byte[right.GetByteCount()];
            BigInteger randomNumber;
            do
            {
                random.GetBytes(buffer);
                randomNumber = new BigInteger(buffer);
            } while ((left > randomNumber) || (randomNumber > right));

            return randomNumber;
        }

        public static int Jacobi(BigInteger a, BigInteger b)
        {
            if (BigInteger.GreatestCommonDivisor(a, b) != 1) return 0;
            int r = 1;
            if (a < BigInteger.Zero)
            {
                a = BigInteger.Negate(a);
                if (BigInteger.Remainder(b, 4) == 3)
                {
                    r = -r;
                }
            }

            do
            {
                int t = 0;
                while (BigInteger.Remainder(a, 2) == 0)
                {
                    t++;
                    a /= 2;
                }

                if (t % 2 != 0)
                {
                    if    (BigInteger.Remainder(b,    8)    ==    3    ||
BigInteger.Remainder(b, 8) == 5)
                        r = -r;
                }

                if    (BigInteger.Remainder(a,    4)    ==    3    &&
BigInteger.Remainder(b, 4) == 3)
```

```
                r = -r;

            BigInteger c = a;
            a = BigInteger.Remainder(b, c);
            b = c;

        } while (a != (BigInteger) 0);

        return r;
    }

    public static BigInteger LucasSequencesMod(BigInteger P,
BigInteger n, BigInteger mod)
    {
        BigInteger prev = P;
        BigInteger current = (P * P - 2) % n;
        var size = n.GetBitLength() - 2;

        var                    nInBinarySystem                    =
string.Concat(n.ToByteArray().Select(b      =>      Convert.ToString(b,
2).PadLeft(8, '0')).Reverse());
        for (var i = size; i > 0; i--)
        {
            if (nInBinarySystem[(int)(nInBinarySystem.Length - i)]
== 1)
            {
                prev = (prev * current - P) % mod;
                current = (current * current - 2) % mod;
            }
            else if (nInBinarySystem[(int)(nInBinarySystem.Length -
i)] == 0)
            {
                current = (prev * current - P) % mod;
                prev = (prev * prev - 2) % mod;
            }
        }

        return prev;
    }
}
}
```