Министерство науки и высшего образования РФ

Федеральное государственное автономное

образовательное учреждение высшего образования

**«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

Институт космических и информационных технологий

институт

Программная инженерия

кафедра

**ОТЧЕТ О ПРАКТИЧЕСКОЙ** **РАБОТЕ №1**

Простые симметричные шифры

тема

Преподаватель Р. С. Шиманович

подпись, дата инициалы, фамилия

Студент КИ23-16/1Б, 032322546 Е. А. Гуртякин

номер группы, зачетной книжки подпись, дата инициалы, фамилия

Красноярск 2025

**СОДЕРЖАНИЕ**

[Задание 3](#_Toc210131248)

[Ход выполнения 4](#_Toc210131249)

[Описание алгоритма шифрования 4](#_Toc210131250)

[Программа, реализующая алгоритм 9](#_Toc210131251)

[Модификация алгоритма 13](#_Toc210131252)

[Программа, реализующая модифицированный алгоритм 17](#_Toc210131253)

[Вывод 23](#_Toc210131254)

[ПРИЛОЖЕНИЕ А 24](#_Toc210131255)

[ПРИЛОЖЕНИЕ Б 27](#_Toc210131256)

# Задание

Согласно Вашему персональному варианту (см. табл. 2) или индивидуальному заданию преподавателя разработайте и составьте в виде блок-схемы алгоритмы шифрования и расшифровывания текста. Убедитесь в правильности составления алгоритмов и затем на языке программирования составьте программу, которая реализует данные алгоритмы. На ряде контрольных примеров (не менее 10) открытого текста, состоящего из различного количества символов, проверьте правильность работы алгоритмов шифрования и расшифровывания. Самостоятельно придумайте способ модификации шифра с целью повышения его криптостойкости. Для этого используйте блоки подстановочных/перестановочных операций. Внесите изменения в исходный алгоритм и программу. Проверьте работоспособность алгоритма на тестовых примерах. Покажите, что предложенный Вами способ модификации действительно повышает криптостокость. Разработанная Вами программа должна содержать графический интерфейс пользователя. Выбранный вариант - №1 (Шифр на основе магических квадратов).

# Ход выполнения

# Описание алгоритма шифрования

На рисунке 1 продемонстрирована блок-схема шифрования текста.

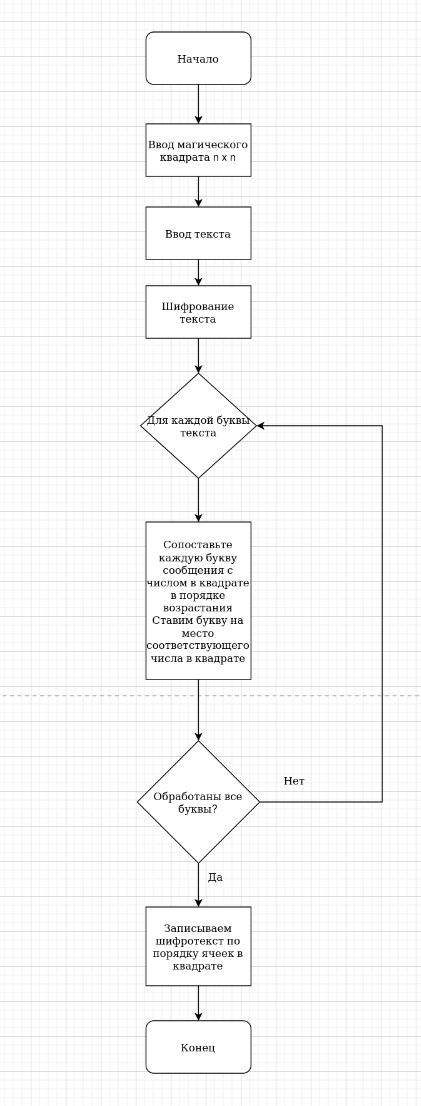


Рисунок 1 – Алгоритм шифрования текста

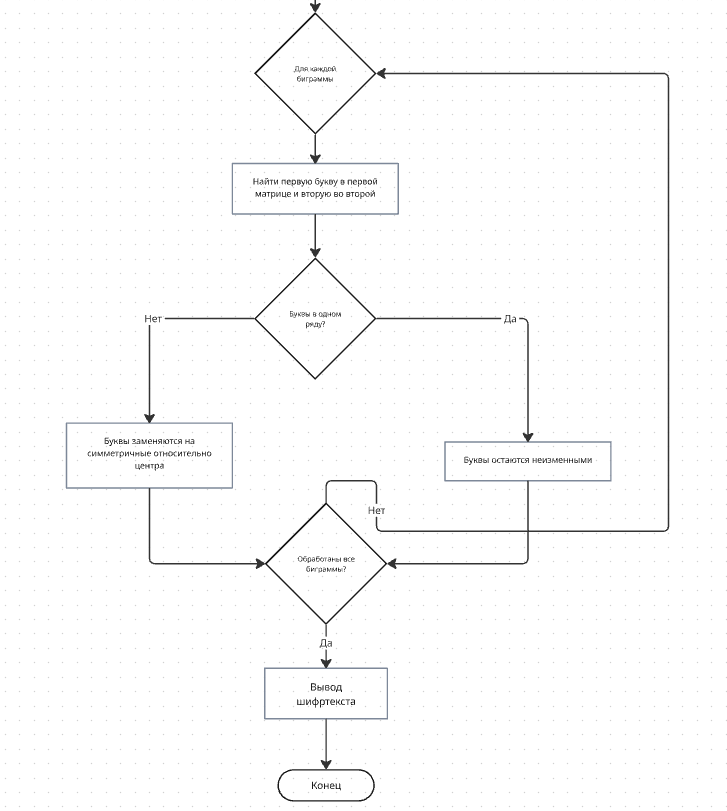


Рисунок 2 – Алгоритм шифрования текста (часть 2)

На рисунках 3 и 4 продемонстрирована блок-схема расшифровки текста.

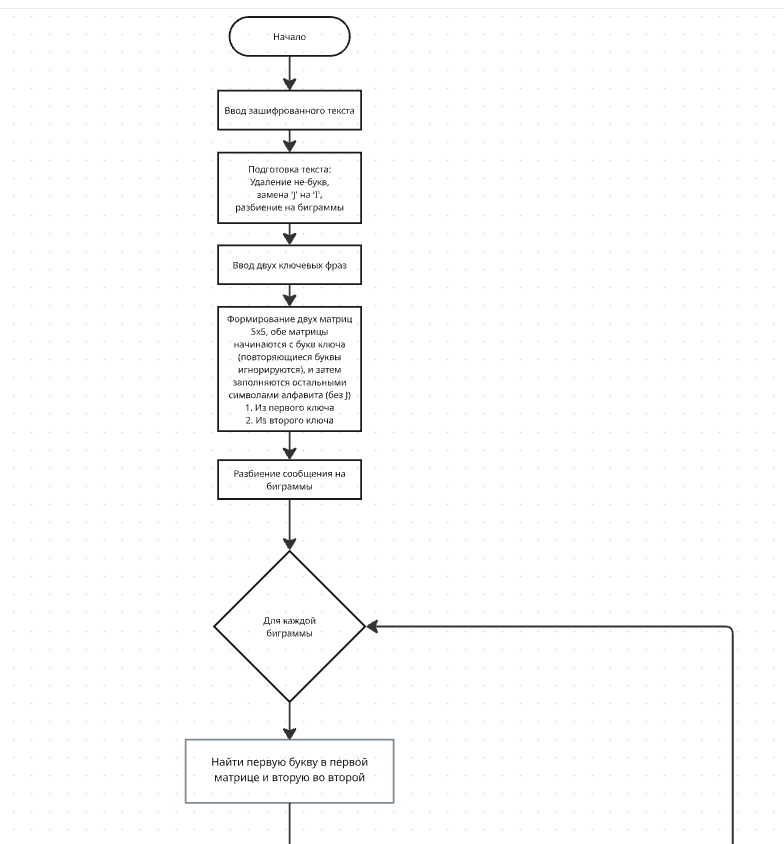


Рисунок 3 – Блок-схема расшифровки текста (часть 1)

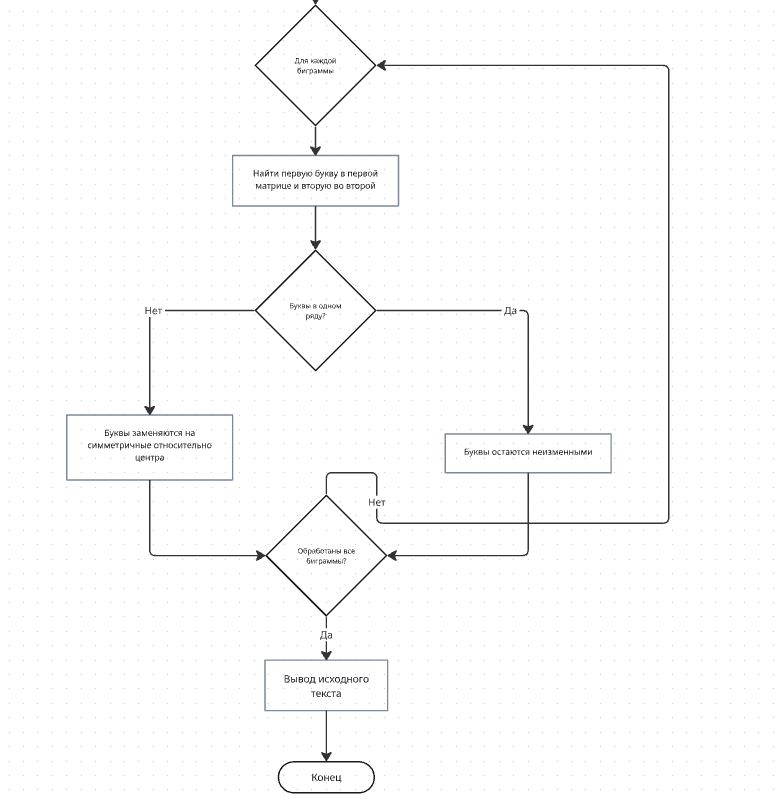


Рисунок 4 – Блок-схема расшифровки текста (часть 2)

# Программа, реализующая алгоритм

Для реализации шифра двойного квадрата была написана программа на Typescript, представленная в приложении А. Результаты работы программы демонстрируются на рисунках 5-10.

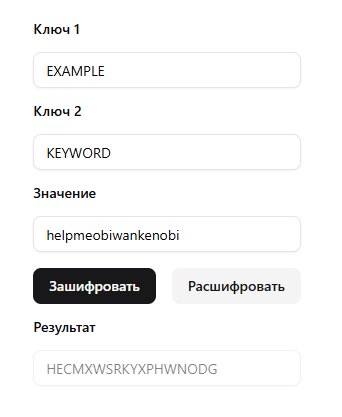


Рисунок 5 – Шифрование сообщения

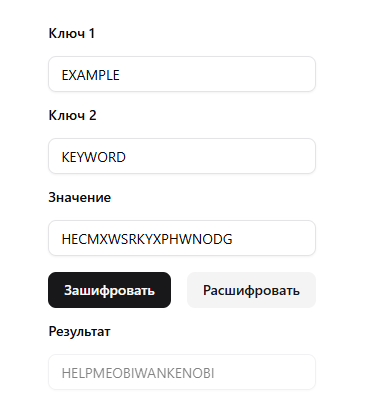


Рисунок 6 – Расшифровка сообщения

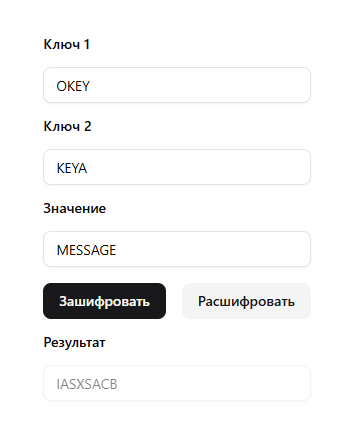


Рисунок 7 – Шифрование сообщения

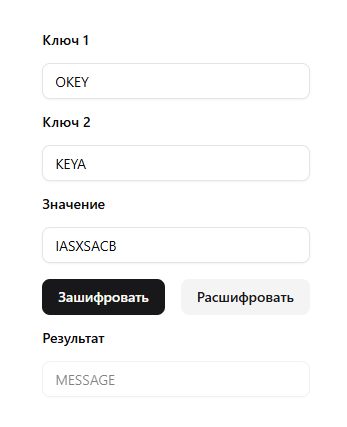


Рисунок 8 – Расшифровка сообщения

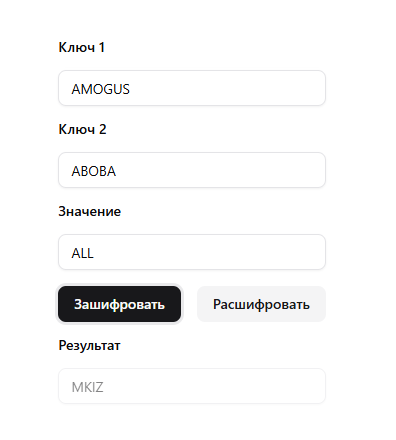


Рисунок 9 – Шифрование сообщения

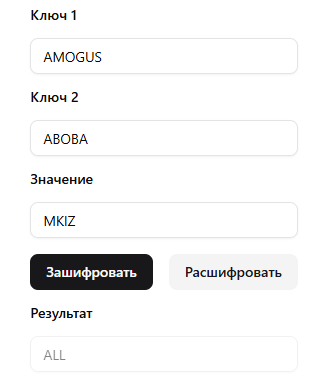


Рисунок 10 – Расшифровка сообщения

# Модификация алгоритма

Предлагаемая модификация кардинально меняет подход к формированию шифровальных матриц. В отличие от исходного алгоритма, где две матрицы остаются статичными на протяжении всего процесса шифрования, в модифицированной версии матрицы динамически изменяются для каждой обрабатываемой биграммы текста. Это достигается за счет введения механизма генерации уникальных seed-значений для каждой позиции в тексте, которые зависят от трех параметров: текущей позиции биграммы, предыдущих уже обработанных символов и базового ключа.

В классическом алгоритме основная уязвимость проистекает из статичности шифровальных матриц. Поскольку соответствие между символами открытого текста и шифротекста остается неизменным на протяжении всего сообщения, криптоаналитик может применять частотный анализ, основанный на статистических закономерностях языка. Один и тот же символ открытого текста всегда будет преобразовываться в один и тот же символ шифротекста в идентичном контексте, что позволяет строить корреляционные модели.

В модифицированном алгоритме эта фундаментальная слабость полностью устраняется. Благодаря динамически меняющимся сеткам, один и тот же символ открытого текста, встречающийся в разных позициях сообщения, будет шифроваться в различные символы шифротекста. Более того, даже идентичные биграммы, появляющиеся в разных частях сообщения, получат совершенно различные шифровальные представления. Это нарушает базовое предположение частотного анализа о сохранении статистических характеристик и делает традиционные методы криптоанализа неэффективными.

Криптографическая стойкость алгоритма напрямую связана с размером пространства ключей. В классическом алгоритме пространство ключей ограничено числом возможных перестановок двух полибианских сеток, что составляет примерно 2 × 25! ≈ 3.1 × 10²⁵ возможных комбинаций.

В модифицированном алгоритме пространство ключей приобретает экспоненциальную зависимость от длины сообщения. Для каждой из m/2 биграмм в сообщении длиной m символов генерируется уникальная пара сеток на основе seed-значений из пространства размером примерно 1000 вариантов (благодаря модульной арифметике в функции генерации seed). Таким образом, общее пространство ключей возрастает до .

Для типичного сообщения из 20 символов (10 биграмм) это дает примерно 3.1 × 10²⁵ × 10³⁰ = 3.1 × 10⁵⁵ возможных комбинаций, что на 30 порядков больше классического варианта.

На рисунках 11 и 12 представлена блок-схема для шифрования сообщения, с выделением изменившейся части алгоритма.

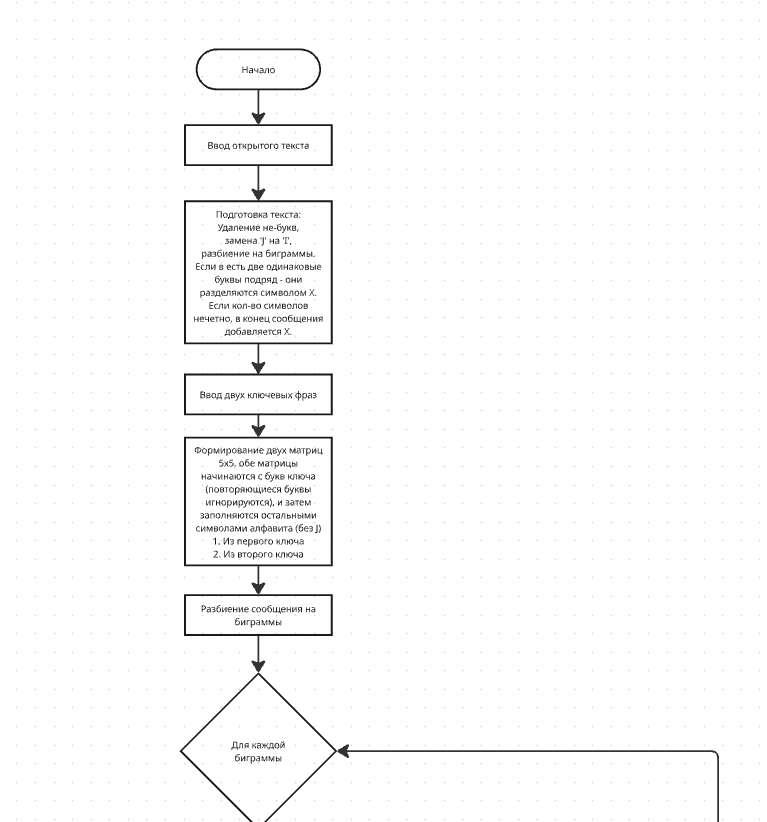


Рисунок 11 – Блок-схема шифрования (часть 1)

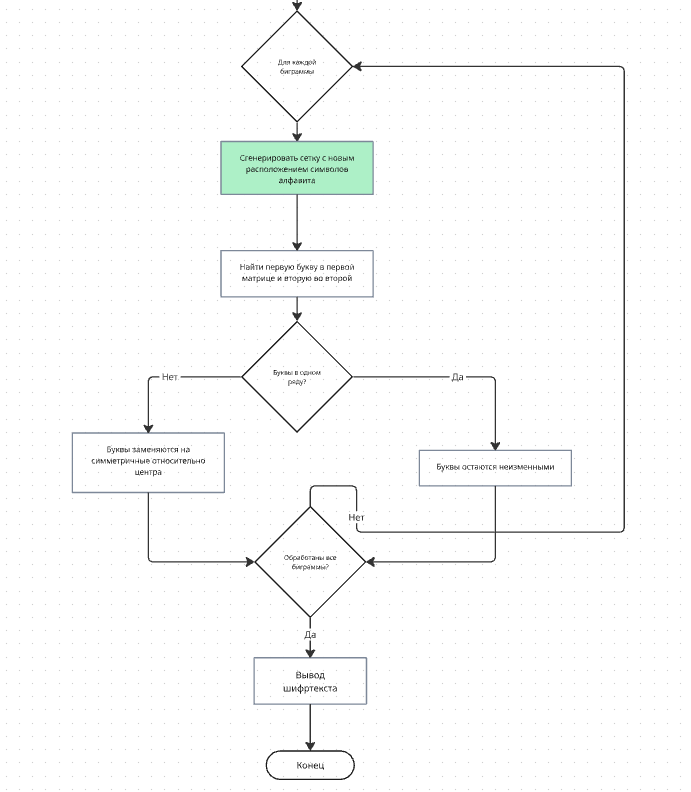


Рисунок 12 – Блок-схема шифрования (часть 2)

На рисунках 13 и 14 демонстрируется блок-схема расшифровки сообщений с выделением изменившейся части алгоритма

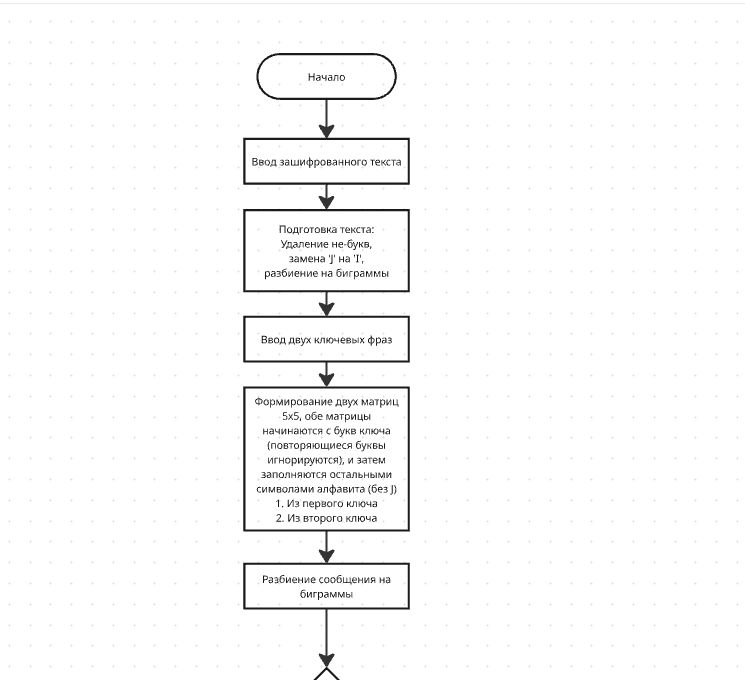


Рисунок 13 – Блок-схема расшифровки (часть 1)

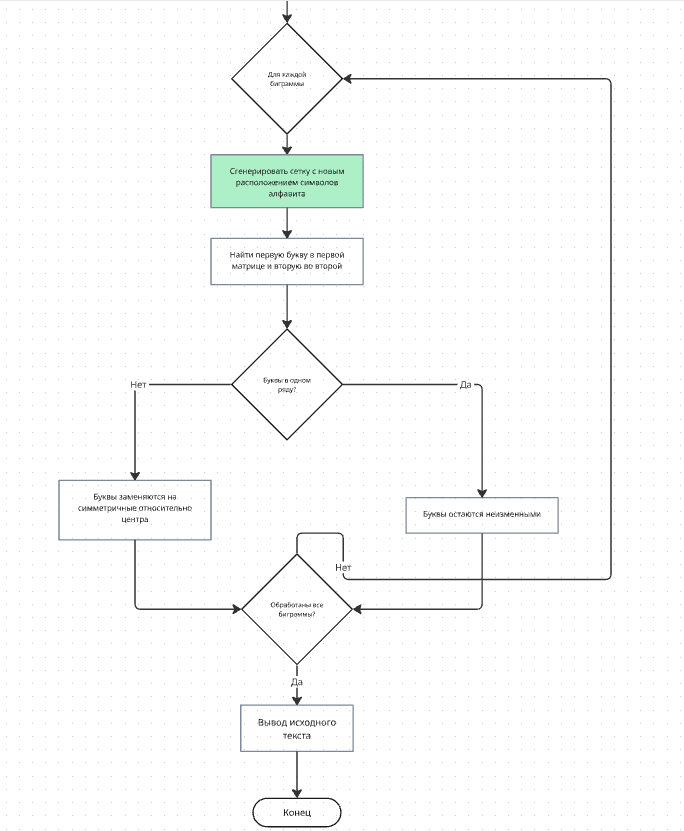


Рисунок 14 – Блок-схема расшифровки (часть 2)

# Программа, реализующая модифицированный алгоритм

Для реализации алгоритма была написана программа на языке Typescript, представленная в Приложении Б. Результаты работы программы представлены на рисунках 15-20.

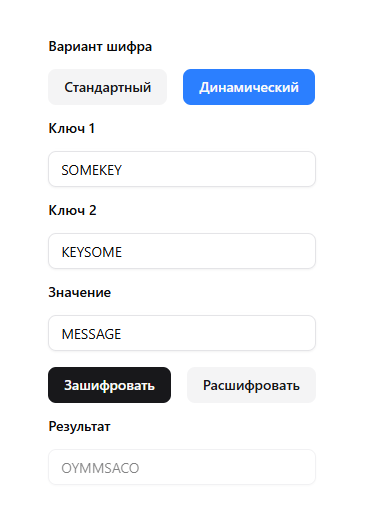


Рисунок 15 – Шифрование сообщения

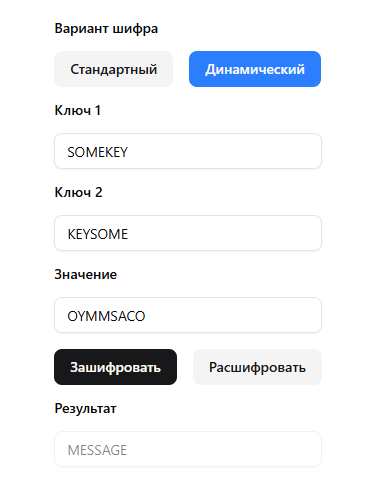


Рисунок 16 – Расшифровка сообщения

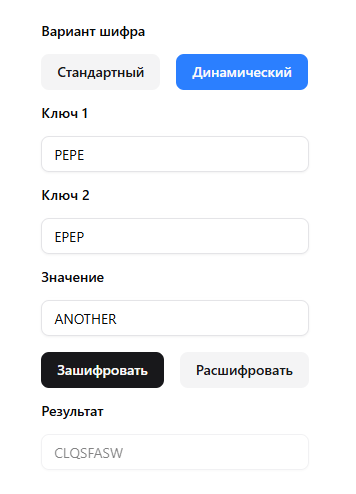


Рисунок 17 – Шифрование сообщения

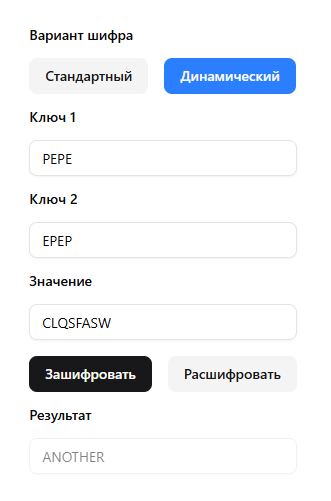


Рисунок 18 – Расшифровка сообщения

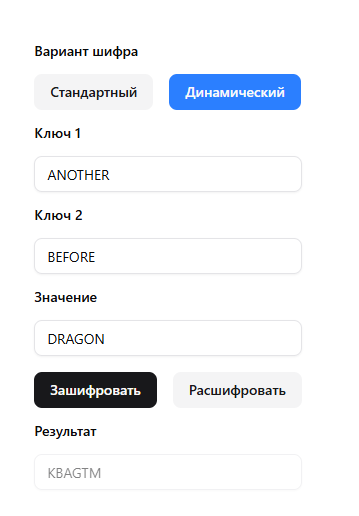


Рисунок 19 – Шифрование сообщения

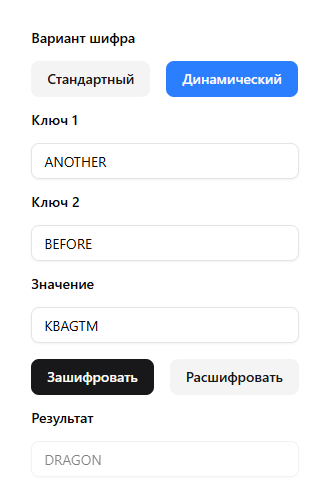


Рисунок 20 – Расшифровка сообщения

Вывод

В ходе выполнения практической работы были получены знания о симметричных шифрах, о криптостойкости алгоритмов шифрования, а также практические навыки реализации симметричных алгоритмов шифрования.

ПРИЛОЖЕНИЕ А

export class TwoSquareCipher {

    private key1: string;

    private key2: string;

    private alphabet: string = 'ABCDEFGHIKLMNOPQRSTUVWXYZ';

    private gridSize: number = 5;

    constructor(key1: string = 'EXAMPLE', key2: string = 'KEYWORD') {

        this.key1 = this.prepareKey(key1);

        this.key2 = this.prepareKey(key2);

    }

    private prepareKey(key: string): string {

        let prepared = key.toUpperCase().replace(/J/g, 'I');

        prepared = prepared.replace(/[^A-Z]/g, '');

        let uniqueChars = '';

        for (const char of prepared) {

            if (!uniqueChars.includes(char)) {

                uniqueChars += char;

            }

        }

        return uniqueChars;

    }

    private generateGrid(key: string): string[][] {

        const grid: string[][] = [];

        let usedLetters = new Set<string>();

        let currentLetters = '';

        for (const char of key) {

            if (!usedLetters.has(char)) {

                currentLetters += char;

                usedLetters.add(char);

            }

        }

        for (const char of this.alphabet) {

            if (!usedLetters.has(char)) {

                currentLetters += char;

                usedLetters.add(char);

            }

        }

        for (let i = 0; i < this.gridSize; i++) {

            grid.push([]);

            for (let j = 0; j < this.gridSize; j++) {

                const index = i \* this.gridSize + j;

                grid[i].push(currentLetters[index]);

            }

        }

        return grid;

    }

    private prepareText(text: string): string {

        let prepared = text.toUpperCase().replace(/J/g, 'I');

        prepared = prepared.replace(/[^A-Z]/g, '');

        let result = '';

        let i = 0;

        while (i < prepared.length) {

            if (i === prepared.length - 1) {

                result += prepared[i] + 'X';

                i++;

            } else if (prepared[i] === prepared[i + 1]) {

                result += prepared[i] + 'X';

                i++;

            } else {

                result += prepared[i] + prepared[i + 1];

                i += 2;

            }

        }

        if (result.length % 2 !== 0) {

            result += 'X';

        }

        return result;

    }

    private findPosition(grid: string[][], char: string): { row: number; col: number } {

        for (let row = 0; row < this.gridSize; row++) {

            for (let col = 0; col < this.gridSize; col++) {

                if (grid[row][col] === char) {

                    return { row: row, col };

                }

            }

        }

        throw new Error(`Character ${char} not found in grid`);

    }

    public encrypt(plaintext: string): string {

        const preparedText = this.prepareText(plaintext);

        const grid1 = this.generateGrid(this.key1);

        const grid2 = this.generateGrid(this.key2);

        let ciphertext = '';

        for (let i = 0; i < preparedText.length; i += 2) {

            const char1 = preparedText[i];

            const char2 = preparedText[i + 1];

            const pos1 = this.findPosition(grid1, char1);

            const pos2 = this.findPosition(grid2, char2);

            const encryptedChar1 = grid1[pos1.row][pos2.col];

            const encryptedChar2 = grid2[pos2.row][pos1.col];

            ciphertext += encryptedChar1 + encryptedChar2;

        }

        return ciphertext;

    }

    public decrypt(ciphertext: string): string {

        const cleanCiphertext = ciphertext.toUpperCase().replace(/[^A-Z]/g, '');

        if (cleanCiphertext.length % 2 !== 0) {

            throw new Error('Ciphertext must have even length');

        }

        const grid1 = this.generateGrid(this.key1);

        const grid2 = this.generateGrid(this.key2);

        let plaintext = '';

        for (let i = 0; i < cleanCiphertext.length; i += 2) {

            const char1 = cleanCiphertext[i];

            const char2 = cleanCiphertext[i + 1];

            const pos1 = this.findPosition(grid1, char1);

            const pos2 = this.findPosition(grid2, char2);

            const decryptedChar1 = grid1[pos1.row][pos2.col];

            const decryptedChar2 = grid2[pos2.row][pos1.col];

            plaintext += decryptedChar1 + decryptedChar2;

        }

        return plaintext.replace(/X$/, '');

    }

    public displayGrids(): void {

        const grid1 = this.generateGrid(this.key1);

        const grid2 = this.generateGrid(this.key2);

        console.log('Grid 1:');

        for (const row of grid1) {

            console.log(row.join(' '));

        }

        console.log('\nGrid 2:');

        for (const row of grid2) {

            console.log(row.join(' '));

        }

    }

    public setKeys(key1: string, key2: string): void {

        this.key1 = this.prepareKey(key1);

        this.key2 = this.prepareKey(key2);

    }

}

ПРИЛОЖЕНИЕ Б

export class DynamicGridTwoSquareCipher {

    private key1: string;

    private key2: string;

    private alphabet: string = 'ABCDEFGHIKLMNOPQRSTUVWXYZ';

    private gridSize: number = 5;

    constructor(key1: string = 'EXAMPLE', key2: string = 'KEYWORD') {

        this.key1 = this.prepareKey(key1);

        this.key2 = this.prepareKey(key2);

    }

    private prepareKey(key: string): string {

        let prepared = key.toUpperCase().replace(/J/g, 'I');

        prepared = prepared.replace(/[^A-Z]/g, '');

        let uniqueChars = '';

        for (const char of prepared) {

            if (!uniqueChars.includes(char)) {

                uniqueChars += char;

            }

        }

        return uniqueChars;

    }

    private generateDynamicGrid(baseKey: string, seed: number): string[][] {

        const grid: string[][] = [];

        let dynamicAlphabet = this.alphabet;

        for (let i = 0; i < seed % 10; i++) {

            dynamicAlphabet = this.rotateString(dynamicAlphabet, (seed + i) % 25);

        }

        let usedLetters = new Set<string>();

        let currentLetters = '';

        for (const char of baseKey) {

            if (!usedLetters.has(char)) {

                currentLetters += char;

                usedLetters.add(char);

            }

        }

        for (const char of dynamicAlphabet) {

            if (!usedLetters.has(char)) {

                currentLetters += char;

                usedLetters.add(char);

            }

        }

        for (let i = 0; i < this.gridSize; i++) {

            grid.push([]);

            for (let j = 0; j < this.gridSize; j++) {

                const index = i \* this.gridSize + j;

                grid[i].push(currentLetters[index]);

            }

        }

        return grid;

    }

    private rotateString(str: string, shift: number): string {

        shift = shift % str.length;

        return str.slice(shift) + str.slice(0, shift);

    }

    private generateSeed(position: number, prevChars: string, baseKey: string): number {

        let seed = 0;

        for (let i = 0; i < prevChars.length; i++) {

            seed = (seed \* 31 + prevChars.charCodeAt(i)) % 1000;

        }

        seed = (seed + position \* 17) % 1000;

        for (let i = 0; i < baseKey.length; i++) {

            seed = (seed + baseKey.charCodeAt(i) \* (i + 1)) % 1000;

        }

        return seed;

    }

    private prepareText(text: string): string {

        let prepared = text.toUpperCase().replace(/J/g, 'I');

        prepared = prepared.replace(/[^A-Z]/g, '');

        let result = '';

        let i = 0;

        while (i < prepared.length) {

            if (i === prepared.length - 1) {

                result += prepared[i] + 'X';

                i++;

            } else if (prepared[i] === prepared[i + 1]) {

                result += prepared[i] + 'X';

                i++;

            } else {

                result += prepared[i] + prepared[i + 1];

                i += 2;

            }

        }

        if (result.length % 2 !== 0) {

            result += 'X';

        }

        return result;

    }

    private findPosition(grid: string[][], char: string): { row: number; col: number } {

        for (let row = 0; row < this.gridSize; row++) {

            for (let col = 0; col < this.gridSize; col++) {

                if (grid[row][col] === char) {

                    return { row: row, col };

                }

            }

        }

        throw new Error(`Character ${char} not found in grid`);

    }

    public encrypt(plaintext: string): string {

        const preparedText = this.prepareText(plaintext);

        let ciphertext = '';

        let previousChars = '';

        for (let i = 0; i < preparedText.length; i += 2) {

            const char1 = preparedText[i];

            const char2 = preparedText[i + 1];

            const seed1 = this.generateSeed(i, previousChars, this.key1);

            const seed2 = this.generateSeed(i + 1, previousChars, this.key2);

            const grid1 = this.generateDynamicGrid(this.key1, seed1);

            const grid2 = this.generateDynamicGrid(this.key2, seed2);

            const pos1 = this.findPosition(grid1, char1);

            const pos2 = this.findPosition(grid2, char2);

            const encryptedChar1 = grid1[pos1.row][pos2.col];

            const encryptedChar2 = grid2[pos2.row][pos1.col];

            ciphertext += encryptedChar1 + encryptedChar2;

            previousChars += char1 + char2;

        }

        return ciphertext;

    }

    public decrypt(ciphertext: string): string {

        const cleanCiphertext = ciphertext.toUpperCase().replace(/[^A-Z]/g, '');

        if (cleanCiphertext.length % 2 !== 0) {

            throw new Error('Ciphertext must have even length');

        }

        let plaintext = '';

        let previousChars = '';

        for (let i = 0; i < cleanCiphertext.length; i += 2) {

            const char1 = cleanCiphertext[i];

            const char2 = cleanCiphertext[i + 1];

            const seed1 = this.generateSeed(i, previousChars, this.key1);

            const seed2 = this.generateSeed(i + 1, previousChars, this.key2);

            const grid1 = this.generateDynamicGrid(this.key1, seed1);

            const grid2 = this.generateDynamicGrid(this.key2, seed2);

            const pos1 = this.findPosition(grid1, char1);

            const pos2 = this.findPosition(grid2, char2);

            const decryptedChar1 = grid1[pos1.row][pos2.col];

            const decryptedChar2 = grid2[pos2.row][pos1.col];

            plaintext += decryptedChar1 + decryptedChar2;

            previousChars += decryptedChar1 + decryptedChar2;

        }

        return this.cleanDecryptedText(plaintext);

    }

    private cleanDecryptedText(text: string): string {

        let result = '';

        let i = 0;

        while (i < text.length) {

            if (i === text.length - 1) {

                result += text[i];

                i++;

            } else if (text[i + 1] === 'X' &&

                      (i + 2 === text.length ||

                       (i + 2 < text.length && text[i] === text[i + 2]))) {

                result += text[i];

                i += 2;

            } else {

                result += text[i] + text[i + 1];

                i += 2;

            }

        }

        return result;

    }

    public displayGridsForPosition(position: number, previousText: string = ''): void {

        const seed1 = this.generateSeed(position, previousText, this.key1);

        const seed2 = this.generateSeed(position + 1, previousText, this.key2);

        const grid1 = this.generateDynamicGrid(this.key1, seed1);

        const grid2 = this.generateDynamicGrid(this.key2, seed2);

        console.log(`Grids for position ${position} with previous text "${previousText}":`);

        console.log('Grid 1:');

        for (const row of grid1) {

            console.log(row.join(' '));

        }

        console.log('\nGrid 2:');

        for (const row of grid2) {

            console.log(row.join(' '));

        }

    }

    public setKeys(key1: string, key2: string): void {

        this.key1 = this.prepareKey(key1);

        this.key2 = this.prepareKey(key2);

    }

}