

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ  
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ТЕЛЕКОММУНИКАЦИЙ  
И ИНФОРМАТИКИ»

Кафедра прикладной математики и кибернетики

**Расчетно-графическая работа**

по дисциплине  
«Защита информации»  
на тему

**Реализация протокола доказательства с нулевым знанием для задачи  
«Гамильтонов цикл»**

Выполнил студент \_\_\_\_\_ Рудских Владислав Евгеньевич  
Ф.И.О.

Группы \_\_\_\_\_ ИВ-222

Работу принял \_\_\_\_\_ Ассистент кафедры ПМиК  
Истомина А. С.  
подпись

Защищена \_\_\_\_\_ Оценка \_\_\_\_\_

Новосибирск – 2025

# СОДЕРЖАНИЕ

ПОСТАНОВКА ЗАДАЧИ .....	2
ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ .....	3
Протокол доказательства с нулевым знанием .....	4
РЕАЛИЗАЦИЯ .....	5
СТРУКТУРА ПРОЕКТА .....	5
ОПИСАНИЕ ФУНКЦИЙ .....	5
ФАЙЛ GRAPH.HPP .....	5
ФАЙЛ GRAPH.CPP .....	6
ФАЙЛ MAIN.CPP .....	7
РАБОТА ПРОТОКОЛА .....	10
ТЕСТИРОВАНИЕ .....	11
ВЫВОД .....	15
ПРИЛОЖЕНИЕ .....	15

## **ПОСТАНОВКА ЗАДАЧИ**

В соответствии с вариантом №2 требуется разработать программу, реализующую протокол доказательства с нулевым знанием для задачи «Гамильтонов цикл». Задача является NP-полной и не имеет быстрых методов решения, поэтому для тестирования необходимо будет генерировать правильные решения при помощи дополнительно разработанных программ. Необходимо считывать информацию о графах из файла. В файле описание графа будет определяться следующим образом:

- в первой строке файла содержатся два числа и  $n < 1001$  и  $m \leq n^2$ , количество вершин графа и количество ребер соответственно;
- в последующих  $m$  строках содержится информация о ребрах графа, каждое из которых описывается с помощью двух чисел (номера вершин, соединяемых этим ребром);
- Указывается последовательность вершин, задающая гамильтонов цикл (этот пункт можно вынести в отдельный файл);

## ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

### **Протокол доказательства с нулевым знанием**

Доказательство с нулевым разглашением (информации) в криптографии (англ. Zero-knowledge proof) — интерактивный криптографический протокол, позволяющий одной из взаимодействующих сторон («The verifier» — проверяющей) убедиться в достоверности какого-либо утверждения (обычно математического), не имея при этом никакой другой информации от второй стороны («The prover» — доказывающей). Причём последнее условие является необходимым, так как обычно доказать, что сторона обладает определёнными сведениями в большинстве случаев тривиально, если она имеет право просто раскрыть информацию. Вся сложность состоит в том, чтобы доказать, что у одной из сторон есть информация, не раскрывая её содержания. Протокол должен учитывать, что доказывающий сможет убедить проверяющего только в случае, если утверждение действительно доказано. В противном случае сделать это будет невозможно, или крайне маловероятно из-за вычислительной сложности.

# **РЕАЛИЗАЦИЯ**

## **СТРУКТУРА ПРОЕКТА**

Проект состоит из трех файлов:

- main.cpp - основной файл с реализацией протокола ZKP
- graph.cpp - реализация классов Graph и HamiltonianCycle
- graph.hpp - заголовочный файл с объявлениями классов

## **ОПИСАНИЕ ФУНКЦИЙ**

### **ФАЙЛ GRAPH.HPP**

Класс Graph:

- Graph(int n = 0) - конструктор, инициализирует граф с n вершинами
- void addEdge(int u, int v) - добавляет ребро между вершинами u и v
- bool hasEdge(int u, int v) const - проверяет наличие ребра между вершинами
- void loadGraphFromFile(const std::string& filename) - загружает граф из файла
- void print() const - выводит информацию о графе

Класс HamiltonianCycle:

- bool isValid(const Graph& graph) const - проверяет корректность гамильтонова цикла
- void loadCycleFromFile(const std::string& filename) - загружает цикл из файла
- void print() const - выводит информацию о цикле

### **ФАЙЛ GRAPH.CPP**

Graph::Graph(int n):

- Инициализирует количество вершин n и ребер m
- Создает матрицу смежности размером n×n, заполненную нулями

`Graph::addEdge(int u, int v):`

- Проверяет корректность номеров вершин
- Добавляет ребро в список edges, если его еще нет
- Обновляет матрицу смежности

`Graph::hasEdge(int u, int v) const:`

- Проверяет границы вершин
- Возвращает значение из матрицы смежности

`Graph::loadGraphFromFile(const std::string& filename):`

- Открывает файл для чтения
- Читает количество вершин n и ребер m
- Инициализирует матрицу смежности
- Читает ребра из файла и добавляет их в граф
- Закрывает файл и выводит статистику

`Graph::print() const:`

- Выводит количество вершин и ребер
- Выводит список ребер
- Выводит часть матрицы смежности (первые  $10 \times 10$  элементов)

`HamiltonianCycle::isValid(const Graph& graph) const:`

- Проверяет, что цикл не пуст
- Проверяет замкнутость цикла (первая == последняя вершина)
- Проверяет длину цикла (должна быть n+1)
- Проверяет, что все вершины посещены ровно один раз
- Проверяет существование всех ребер цикла в графе
- Возвращает true, если все проверки пройдены

`HamiltonianCycle::loadCycleFromFile(const std::string& filename):`

- Открывает файл для чтения
- Читает размер цикла и последовательность вершин
- Закрывает файл и выводит информацию

`HamiltonianCycle::print() const:`

- Выводит длину цикла
- Выводит первые 15 вершин цикла

## ФАЙЛ MAIN.CPP

Класс Verifier:

1. Verifier(const Graph& g):
  - a. Конструктор, инициализирует верификатор с ссылкой на граф
  - b. Инициализирует генератор случайных чисел
2. bool receiveCommitment(const vector<vector<int>>& encryptedGraph):
  - a. Проверяет размер зашифрованного графа
  - b. Проверяет размер каждой строки матрицы
  - c. Возвращает true, если размеры корректны
3. int sendChallenge():
  - a. Генерирует случайный запрос (0 или 1)
  - b. 0 - раскрыть изоморфизм
  - c. 1 - раскрыть цикл
4. bool verifyResponse(int challenge, const vector<vector<int>>& encryptedGraph, const vector<int>& response):
  - a. В зависимости от запроса вызывает соответствующую функцию проверки
5. bool verifyIsomorphism(const vector<int>& permutation, const vector<vector<int>>& encryptedGraph):
  - a. Проверяет, что permutation - корректная перестановка вершин
  - b. Проверяет изоморфизм между исходным графом и зашифрованным
6. bool verifyCycle(const vector<int>& response, const vector<vector<int>>& encryptedGraph):
  - a. Проверяет длину цикла в ответе
  - b. Проверяет замкнутость цикла
  - c. Проверяет, что все вершины различны
  - d. Проверяет существование всех ребер цикла в зашифрованном графе

Класс Prover:

1. Prover(const Graph& g, const HamiltonianCycle& hc):
  - a. Конструктор, инициализирует доказывающего с графом и циклом
  - b. Инициализирует генератор случайных чисел
2. void preparePermutation():
  - a. Создает случайную перестановку вершин
  - b. Перемешивает вершины с помощью std::shuffle
  - c. Выводит первые 10 элементов для отладки
3. vector<vector<int>> createEncryptedGraph():
  - a. Создает зашифрованный граф путем применения перестановки к исходному графу
  - b. Возвращает матрицу смежности зашифрованного графа
4. vector<int> respondToChallenge0():
  - a. Возвращает перестановку вершин (для запроса 0)
5. vector<int> respondToChallenge1():
  - a. Создает ответ для запроса 1

- b. Добавляет длину цикла
- c. Преобразует исходный цикл с помощью перестановки
- d. Возвращает зашифрованный цикл

Основные функции:

1. bool runZKPProtocol(const Graph& graph, const HamiltonianCycle& hc, int rounds = 20):
  - a. Основная функция протокола ZKP
  - b. Создает экземпляры Prover и Verifier
  - c. Выполняет указанное количество раундов:
    - i. Prover создает перестановку и зашифрованный граф
    - ii. Verifier проверяет зашифрованный граф
    - iii. Verifier отправляет случайный запрос
    - iv. Prover отвечает на запрос
    - v. Verifier проверяет ответ
  - d. Возвращает true, если все раунды пройдены успешно
2. int main(int argc, char\* argv[]):
  - a. Точка входа программы
  - b. Обрабатывает аргументы командной строки
  - c. Загружает граф и гамильтонов цикл из файлов
  - d. Проверяет корректность цикла
  - e. Запускает протокол ZKP
  - f. Выводит результаты

## РАБОТА ПРОТОКОЛА

Протокол работает в несколько раундов:

1. Подготовка: Prover создает случайную перестановку вершин
2. Шифрование: Prover применяет перестановку к графу, получая изоморфный граф
3. Передача: Prover отправляет зашифрованный граф Verifier'у
4. Запрос: Verifier случайным образом выбирает один из двух запросов:
  - a. Раскрыть перестановку (доказать изоморфизм)
  - b. Раскрыть гамильтонов цикл в зашифрованном графе
5. Ответ: Prover отправляет соответствующий ответ
6. Проверка: Verifier проверяет корректность ответа

Протокол обеспечивает нулевое разглашение, так как:

- При запросе 0 Verifier узнает только перестановку, но не узнает цикл
- При запросе 1 Verifier узнает только цикл в зашифрованном графе, но не может восстановить исходный цикл
- Для успешного обмана Prover должен угадать запрос заранее (вероятность 1/2 в каждом раунде)

## ТЕСТИРОВАНИЕ

Для среднего графа (20 вершин):

```
✓ Проверка пройдена успешно

==== Раунд 15 ====
Перестановка (первые 10): 39 22 24 15 19 44 48 41 14 45 ...
Верификатор выбирает запрос: раскрыть цикл
Доказывающий: отправлен зашифрованный цикл
✓ Проверка пройдена успешно

==== Раунд 16 ====
Перестановка (первые 10): 12 39 18 25 41 33 21 24 20 14 ...
Верификатор выбирает запрос: раскрыть изоморфизм
Доказывающий: отправлена перестановка
✓ Проверка пройдена успешно

==== Раунд 17 ====
Перестановка (первые 10): 12 30 45 23 47 48 44 3 29 7 ...
Верификатор выбирает запрос: раскрыть изоморфизм
Доказывающий: отправлена перестановка
✓ Проверка пройдена успешно

==== Раунд 18 ====
Перестановка (первые 10): 3 46 25 33 36 4 21 15 24 11 ...
Верификатор выбирает запрос: раскрыть изоморфизм
Доказывающий: отправлена перестановка
✓ Проверка пройдена успешно

==== Раунд 19 ====
Перестановка (первые 10): 8 14 21 32 34 38 4 45 43 15 ...
Верификатор выбирает запрос: раскрыть изоморфизм
Доказывающий: отправлена перестановка
✓ Проверка пройдена успешно

==== Раунд 20 ====
Перестановка (первые 10): 32 48 44 16 33 15 39 5 28 9 ...
Верификатор выбирает запрос: раскрыть цикл
Доказывающий: отправлен зашифрованный цикл
✓ Проверка пройдена успешно

=====
Все 20 раундов пройдены успешно!
Успешных раундов: 20/20

VVV ДОКАЗАТЕЛЬСТВО ПРИНЯТО! VVV
Доказывающий успешно доказал знание гамильтонова цикла.
skif@DESKTOP-J0LT0TD:~/ZI/rep/crypt/rgr$ |
```

Для маленького графа (10 вершин):

```
✓ Проверка пройдена успешно

==== Раунд 15 ====
Перестановка (первые 10): 1 4 3 7 0 5 9 2 6 8
Верификатор выбирает запрос: раскрыть цикл
Доказывающий: отправлен зашифрованный цикл
✓ Проверка пройдена успешно

==== Раунд 16 ====
Перестановка (первые 10): 9 5 8 6 1 4 7 2 3 0
Верификатор выбирает запрос: раскрыть цикл
Доказывающий: отправлен зашифрованный цикл
✓ Проверка пройдена успешно

==== Раунд 17 ====
Перестановка (первые 10): 1 2 6 0 9 5 4 8 3 7
Верификатор выбирает запрос: раскрыть цикл
Доказывающий: отправлен зашифрованный цикл
✓ Проверка пройдена успешно

==== Раунд 18 ====
Перестановка (первые 10): 0 1 3 9 6 7 5 8 2 4
Верификатор выбирает запрос: раскрыть изоморфизм
Доказывающий: отправлена перестановка
✓ Проверка пройдена успешно

==== Раунд 19 ====
Перестановка (первые 10): 5 2 4 3 7 8 6 9 0 1
Верификатор выбирает запрос: раскрыть изоморфизм
Доказывающий: отправлена перестановка
✓ Проверка пройдена успешно

==== Раунд 20 ====
Перестановка (первые 10): 4 0 5 3 8 1 9 2 6 7
Верификатор выбирает запрос: раскрыть цикл
Доказывающий: отправлен зашифрованный цикл
✓ Проверка пройдена успешно

=====
Все 20 раундов пройдены успешно!
Успешных раундов: 20/20

VVV ДОКАЗАТЕЛЬСТВО ПРИНЯТО! VVV
Доказывающий успешно доказал знание гамильтонова цикла.
skif@DESKTOP-J0LT0TD:~/ZI/rep/crypt/rgr$ |
```

Для большого графа (50 вершин):

```
✓ Проверка пройдена успешно

==== Раунд 25 ===
    Перестановка (первые 10): 5 19 32 16 23 12 3 20 46 49 ...
Верификатор выбирает запрос: раскрыть цикл
Доказывающий: отправлен зашифрованный цикл
✓ Проверка пройдена успешно

==== Раунд 26 ===
    Перестановка (первые 10): 2 14 3 26 33 5 32 27 43 7 ...
Верификатор выбирает запрос: раскрыть изоморфизм
Доказывающий: отправлена перестановка
✓ Проверка пройдена успешно

==== Раунд 27 ===
    Перестановка (первые 10): 16 14 28 43 5 48 32 20 4 12 ...
Верификатор выбирает запрос: раскрыть изоморфизм
Доказывающий: отправлена перестановка
✓ Проверка пройдена успешно

==== Раунд 28 ===
    Перестановка (первые 10): 32 3 31 48 17 39 14 0 2 29 ...
Верификатор выбирает запрос: раскрыть цикл
Доказывающий: отправлен зашифрованный цикл
✓ Проверка пройдена успешно

==== Раунд 29 ===
    Перестановка (первые 10): 49 11 40 15 26 16 20 7 31 10 ...
Верификатор выбирает запрос: раскрыть изоморфизм
Доказывающий: отправлена перестановка
✓ Проверка пройдена успешно

==== Раунд 30 ===
    Перестановка (первые 10): 41 26 0 42 21 45 31 44 27 29 ...
Верификатор выбирает запрос: раскрыть изоморфизм
Доказывающий: отправлена перестановка
✓ Проверка пройдена успешно

=====
Все 30 раундов пройдены успешно!
Успешных раундов: 30/30

//V ДОКАЗАТЕЛЬСТВО ПРИНЯТО! //V
Доказывающий успешно доказал знание гамильтонова цикла.
skif@DESKTOP-J0LT0TD:~/ZI/rep/crypt/rgr$ |
```

## **ВЫВОД**

В ходе выполнения работы был изучен и реализован протокол доказательства с нулевым разглашением для задачи о гамильтоновом цикле. На основе теоретических основ протокола разработано приложение на языке C++, которое моделирует взаимодействие доказывающего и верификатора.

Реализация наглядно демонстрирует принципы работы протоколов с нулевым разглашением и может служить основой для дальнейшего исследования более сложных криптографических схем. Полученная программа успешно подтверждает знание гамильтонова цикла без раскрытия информации о самом цикле, что соответствует требованиям безопасности в криптографических системах.

## ПРИЛОЖЕНИЕ

graph.cpp:

```
#include "graph.hpp"
#include <fstream>
#include <iostream>
#include <algorithm>
#include <stdexcept>

using namespace std;

Graph::Graph(int n) : n(n), m(0) {
    adjacency.resize(n, vector<int>(n, 0));
}

void Graph::addEdge(int u, int v) {
    if (u >= 0 && u < n && v >= 0 && v < n && u != v) {
        if (!hasEdge(u, v)) {
            edges.push_back({u, v});
            adjacency[u][v] = 1;
            adjacency[v][u] = 1;
        }
    }
}

bool Graph::hasEdge(int u, int v) const {
    if (u < 0 || u >= n || v < 0 || v >= n) return false;
    return adjacency[u][v] == 1;
}

void Graph::loadGraphFromFile(const std::string& filename) {
    ifstream file(filename);
    if (!file.is_open()) {
        throw runtime_error("Не удалось открыть файл графа: " +
filename);
    }

    file >> n >> m;
    cout << "Чтение графа: n=" << n << ", m=" << m << endl;

    adjacency.clear();
    adjacency.resize(n, vector<int>(n, 0));
    edges.clear();

    int edges_read = 0;
    for (int i = 0; i < m; i++) {
        int u, v;
```

```

        if (!(file >> u >> v)) {
            throw runtime_error("Ошибка чтения ребра " +
to_string(i) + " " + to_string(m));
        }

        if (u < 0 || u >= n || v < 0 || v >= n) {
            throw runtime_error("Неверные номера вершин в
ребре: " +
to_string(u) + " " +
to_string(v));
        }

        addEdge(u, v);
        edges_read++;
    }

    if (edges_read != m) {
        cout << "Предупреждение: прочитано " << edges_read <<
" ребер вместо " << m << endl;
        m = edges_read;
    }

    file.close();
    cout << "Граф успешно загружен: " << n << " вершин, " << m
<< " ребер" << endl;
}

void Graph::print() const {
    cout << "Граф (n=" << n << ", m=" << m << ") :" << endl;
    cout << "Ребра: ";
    for (const auto& [u, v] : edges) {
        cout << "(" << u << "," << v << ")";
    }
    cout << endl;

    cout << "Матрица смежности:" << endl;
    for (int i = 0; i < min(n, 10); i++) { // показываем только
первые 10 строк
        for (int j = 0; j < min(n, 10); j++) { // и первые 10
столбцов
            cout << adjacency[i][j] << " ";
        }
        if (n > 10) cout << "...";
        cout << endl;
    }
    if (n > 10) cout << "... (показаны только первые 10x10
элементов)" << endl;
}

```

```

bool HamiltonianCycle::isValid(const Graph& graph) const {
    if (cycle.empty()) {
        cout << "Цикл пуст!" << endl;
        return false;
    }

    // Проверяем, что цикл начинается и заканчивается в одной
вершине
    if (cycle[0] != cycle.back()) {
        cout << "Цикл не замкнут: первая вершина=" << cycle[0]
             << ", последняя=" << cycle.back() << endl;
        return false;
    }

    // Проверяем длину цикла
    int expectedLength = graph.n + 1;
    if (cycle.size() != expectedLength) {
        cout << "Неверная длина цикла: ожидается " <<
expectedLength
             << ", получено " << cycle.size() << endl;
        return false;
    }

    // Проверяем, что все вершины графа присутствуют ровно один
раз
    vector<bool> visited(graph.n, false);
    for (size_t i = 0; i < cycle.size() - 1; i++) {
        int v = cycle[i];
        if (v < 0 || v >= graph.n) {
            cout << "Неверный номер вершины: " << v << endl;
            return false;
        }
        if (visited[v]) {
            cout << "Вершина " << v << " встречается более
одного раза!" << endl;
            return false;
        }
        visited[v] = true;
    }

    // Проверяем все ли вершины посещены
    for (int i = 0; i < graph.n; i++) {
        if (!visited[i]) {
            cout << "Вершина " << i << " не посещена!" << endl;
            return false;
        }
    }

    // Проверяем все ребра цикла

```

```

        for (size_t i = 0; i < cycle.size() - 1; i++) {
            int u = cycle[i];
            int v = cycle[i + 1];
            if (!graph.hasEdge(u, v)) {
                cout << "Ребро (" << u << ", " << v << ")"
отсутствует в графе!" << endl;
                return false;
            }
        }

        cout << "Гамильтонов цикл корректен!" << endl;
        return true;
    }

    void HamiltonianCycle::loadCycleFromFile(const std::string&
filename) {
        ifstream file(filename);
        if (!file.is_open()) {
            throw runtime_error("Не удалось открыть файл цикла: " +
filename);
        }

        int size;
        if (!(file >> size)) {
            throw runtime_error("Ошибка чтения размера цикла");
        }

        cycle.resize(size);
        for (int i = 0; i < size; i++) {
            if (!(file >> cycle[i])) {
                throw runtime_error("Ошибка чтения вершины " +
to_string(i));
            }
        }

        file.close();
        cout << "Цикл загружен: " << size << " вершин" << endl;
    }

    void HamiltonianCycle::print() const {
        cout << "Гамильтонов цикл (" << cycle.size() << " вершин) :
";
        int limit = min(static_cast<int>(cycle.size()), 15);
        for (int i = 0; i < limit; i++) {
            cout << cycle[i] << " ";
        }
        if (cycle.size() > 15) {
            cout << "... " << cycle.back();
        }
    }
}

```

```
    cout << endl;
```

```
}
```

### graph.hpp:

```
#ifndef GRAPH_HPP
#define GRAPH_HPP

#include <vector>
#include <string>
#include <utility>

struct Graph {
    int n; // количество вершин
    int m; // количество ребер
    std::vector<std::pair<int, int>> edges; // список ребер
    std::vector<std::vector<int>> adjacency; // матрица смежности

    Graph(int n = 0);
    void addEdge(int u, int v);
    bool hasEdge(int u, int v) const;
    void loadGraphFromFile(const std::string& filename);
    void print() const;
};


```

```
struct HamiltonianCycle {
    std::vector<int> cycle; // последовательность вершин цикла
    bool isValid(const Graph& graph) const;
    void loadCycleFromFile(const std::string& filename);
    void print() const;
};

#endif
```

### main.cpp:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <random>
#include <ctime>
#include <cstring>
#include <stdexcept>
#include "graph.hpp"
```

```
using namespace std;
```

```
// Класс для верификатора
class Verifier {
private:
```

```

        const Graph& graph;
        mt19937 rng;

    public:
        Verifier(const Graph& g) : graph(g) {
            rng.seed(time(nullptr) + 1);
        }

        bool receiveCommitment(const vector<vector<int>>&
encryptedGraph) {
            // Проверяем, что зашифрованный граф имеет правильный
размер
            if (encryptedGraph.size() != (size_t)graph.n) {
                cout << " Неверный размер зашифрованного графа" <<
endl;
                return false;
            }
            for (const auto& row : encryptedGraph) {
                if (row.size() != (size_t)graph.n) {
                    cout << " Неверный размер строки в
зашифрованном графе" << endl;
                    return false;
                }
            }
            return true;
        }

        int sendChallenge() {
            uniform_int_distribution<int> dist(0, 1);
            return dist(rng);
        }

        bool verifyResponse(int challenge,
                            const vector<vector<int>>&
encryptedGraph,
                            const vector<int>& response) {

            if (challenge == 0) {
                return verifyIsomorphism(response, encryptedGraph);
            } else {
                return verifyCycle(response, encryptedGraph);
            }
        }

    private:
        bool verifyIsomorphism(const vector<int>& permutation,
                              const vector<vector<int>>&
encryptedGraph) {

```

```

        if (permutation.size() != (size_t)graph.n) {
            cout << " Неверный размер перестановки" << endl;
            return false;
        }

        // Проверяем, что permutation - это перестановка вершин
        vector<bool> seen(graph.n, false);
        for (int v : permutation) {
            if (v < 0 || v >= graph.n) {
                cout << " Неверный номер вершины в
перестановке: " << v << endl;
                return false;
            }
            if (seen[v]) {
                cout << " Вершина " << v << " повторяется в
перестановке" << endl;
                return false;
            }
            seen[v] = true;
        }

        // Проверяем, что encryptedGraph изоморфен исходному
        // графу
        for (int i = 0; i < graph.n; i++) {
            for (int j = i + 1; j < graph.n; j++) {
                bool originalEdge = graph.hasEdge(i, j);
                bool encryptedEdge =
(encryptedGraph[permutation[i]][permutation[j]] == 1);
                if (originalEdge != encryptedEdge) {
                    cout << " Несоответствие ребра: (" << i <<
", " << j << ")";
                    cout << "оригинал=" << originalEdge << ", "
зашифровано=" << encryptedEdge << endl;
                    return false;
                }
            }
        }
        return true;
    }

    bool verifyCycle(const vector<int>& response,
                     const vector<vector<int>>& encryptedGraph)
{
    if (response.empty()) {
        cout << " Пустой ответ" << endl;
        return false;
    }
}

```

```

// Первый элемент - длина цикла
int cycleLength = response[0];
if (cycleLength != graph.n + 1) {
    cout << " Неверная длина цикла: " << cycleLength
        << ", ожидается " << graph.n + 1 << endl;
    return false;
}

if (response.size() < (size_t)cycleLength + 1) {
    cout << " Недостаточно данных в ответе" << endl;
    return false;
}

// Извлекаем цикл из response
vector<int> cycle(response.begin() + 1,
response.begin() + 1 + cycleLength);

// Проверяем, что цикл начинается и заканчивается в
одной вершине
if (cycle[0] != cycle.back()) {
    cout << " Цикл не замкнут" << endl;
    return false;
}

// Проверяем, что все вершины (кроме первой/последней)
различны
vector<bool> visited(graph.n, false);
for (int i = 0; i < graph.n; i++) {
    int v = cycle[i];
    if (v < 0 || v >= graph.n) {
        cout << " Неверный номер вершины в цикле: " <<
v << endl;
        return false;
    }
    if (visited[v]) {
        cout << " Вершина " << v << " повторяется в
цикле" << endl;
        return false;
    }
    visited[v] = true;
}

// Проверяем все ребра цикла в зашифрованном графе
for (int i = 0; i < graph.n; i++) {
    int u = cycle[i];
    int v = cycle[(i + 1) % cycleLength];
    if (encryptedGraph[u][v] != 1) {
        cout << " Отсутствует ребро (" << u << "," <<
v << ") в зашифрованном графе" << endl;
}

```

```

        return false;
    }
}

return true;
};

// Класс для доказывающего
class Prover {
private:
    const Graph& graph;
    const HamiltonianCycle& hamiltonianCycle;
    mt19937 rng;
    vector<int> permutation;

public:
    Prover(const Graph& g, const HamiltonianCycle& hc)
        : graph(g), hamiltonianCycle(hc) {
        rng.seed(time(nullptr));
    }

    void preparePermutation() {
        permutation.resize(graph.n);
        for (int i = 0; i < graph.n; i++) {
            permutation[i] = i;
        }
        shuffle(permutation.begin(), permutation.end(), rng);
    }

    // Выводим перестановку для отладки (первые 10
элементов)
    cout << " Перестановка (первые 10): ";
    for (int i = 0; i < min(10, graph.n); i++) {
        cout << permutation[i] << " ";
    }
    if (graph.n > 10) cout << "...";
    cout << endl;
}

vector<vector<int>> createEncryptedGraph() {
    vector<vector<int>> encrypted(graph.n,
vector<int>(graph.n, 0));

    // Применяем перестановку к исходному графу
    for (int i = 0; i < graph.n; i++) {
        for (int j = i + 1; j < graph.n; j++) {
            if (graph.hasEdge(i, j)) {
                int u = permutation[i];
                int v = permutation[j];

```

```

        encrypted[u][v] = 1;
        encrypted[v][u] = 1;
    }
}

return encrypted;
}

vector<int> respondToChallenge0() {
    return permutation; // возвращаем перестановку
}

vector<int> respondToChallenge1() {
    vector<int> response;

    // Добавляем длину цикла
    response.push_back(hamiltonianCycle.cycle.size());

    // Преобразуем цикл согласно перестановке
    for (size_t i = 0; i < hamiltonianCycle.cycle.size();
i++) {
        int originalVertex = hamiltonianCycle.cycle[i];
        int encryptedVertex = permutation[originalVertex];
        response.push_back(encryptedVertex);
    }

    return response;
}
};

// Протокол доказательства с нулевым разглашением
bool runZKPProtocol(const Graph& graph, const HamiltonianCycle&
hc, int rounds = 20) {
    Prover prover(graph, hc);
    Verifier verifier(graph);

    int successCount = 0;

    for (int round = 0; round < rounds; round++) {
        cout << "\n==== Раунд " << round + 1 << " ===" << endl;

        // 1. Доказывающий готовится
        prover.preparePermutation();
        auto encryptedGraph = prover.createEncryptedGraph();

        // 2. Доказывающий отправляет зашифрованный граф
        if (!verifier.receiveCommitment(encryptedGraph)) {
            cout << "Доказывающий: ошибка при создании"

```

```

зашифрованного графа" << endl;
        return false;
    }

    // 3. Верификатор отправляет случайный запрос
    int challenge = verifier.sendChallenge();
    cout << "Верификатор выбирает запрос: "
        << (challenge == 0 ? "раскрыть изоморфизм" :
    "раскрыть цикл") << endl;

    // 4. Доказывающий отвечает на запрос
    vector<int> response;
    if (challenge == 0) {
        response = prover.respondToChallenge0();
        cout << "Доказывающий: отправлена перестановка" <<
    endl;
    } else {
        response = prover.respondToChallenge1();
        cout << "Доказывающий: отправлен зашифрованный
цикл" << endl;
    }

    // 5. Верификатор проверяет ответ
    if (verifier.verifyResponse(challenge, encryptedGraph,
response)) {
        cout << "✓ Проверка пройдена успешно" << endl;
        successCount++;
    } else {
        cout << "✗ Проверка не пройдена!" << endl;
        return false;
    }
}

cout << "\n======" <<
endl;
cout << "Все " << rounds << " раундов пройдены успешно!" <<
endl;
cout << "Успешных раундов: " << successCount << "/" <<
rounds << endl;
return true;
}

int main(int argc, char* argv[]) {
try {
    if (argc < 3) {
        cout << "Использование: " << argv[0]
            << " <файл_графа> <файл_цикла>
[количество_раундов]" << endl;

```

```

        cout << "Пример: " << argv[0] << " graph.txt
cycle.txt 10" << endl;
        return 1;
    }

    string graphFile = argv[1];
    string cycleFile = argv[2];
    int rounds = (argc > 3) ? atoi(argv[3]) : 10;

    if (rounds <= 0) {
        cout << "Количество раундов должно быть
положительным числом" << endl;
        return 1;
    }

    cout << "==== Доказательство с нулевым разглашением для
гамильтона цикла ===" << endl;
    cout << "Файл графа: " << graphFile << endl;
    cout << "Файл цикла: " << cycleFile << endl;
    cout << "Количество раундов: " << rounds << endl;
    cout << "===== ===== ===== ===== =====" <<
endl;

    // Загрузка графа
    Graph graph;
    graph.loadGraphFromFile(graphFile);
    cout << endl;

    // Загрузка гамильтона цикла
    HamiltonianCycle hc;
    hc.loadCycleFromFile(cycleFile);
    cout << endl;

    // Проверка корректности цикла
    if (!hc.isValid(graph)) {
        cout << "ОШИБКА: Некорректный гамильтонов цикл!" <<
endl;
        return 1;
    }

    // Запуск протокола
    bool result = runZKPProtocol(graph, hc, rounds);

    if (result) {
        cout << "\n✓✓✓ ДОКАЗАТЕЛЬСТВО ПРИНЯТО! ✓✓✓" <<
endl;
        cout << "Доказывающий успешно доказал знание
гамильтона цикла." << endl;

```

```
        return 0;
    } else {
        cout << "\nXXX ДОКАЗАТЕЛЬСТВО ОТВЕРГНУТО! XXX"
<< endl;
        cout << "Доказывающему не удалось доказать знание
гамильтона цикла." << endl;
        return 1;
    }

} catch (const exception& e) {
    cerr << "Ошибка: " << e.what() << endl;
    return 1;
} catch (...) {
    cerr << "Неизвестная ошибка" << endl;
    return 1;
}
}
```