­­

Московский государственный университет имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Кафедра автоматизации систем вычислительных комплексов

Гудис Александр Вячеславович

**Исследование метода Отелло хеширования для доступа к таблицам классификации в архитектуре сетевого процессорного устройства**

КУРСОВАЯ РАБОТА

**Научный руководитель:**

к.ф-м.н, доцент

Волканов Дмитрий Юрьевич

Москва, 2024

**Аннотация**

В работе исследуется метод хеширования данных для организации таблиц классификации типа MAC-VLAN. Рассматривается и изучается метод Отелло хеширования для представления такой таблицы. Сравнительные экспериментальные исследования с уже существующими классическими методами хеширования для доступа к таблицам классификации показали, что Отелло хеширование обладает высокой эффективностью в рамках поставленной задачи, а также гибкостью настройки.

Оглавление

[Введение 5](#_Toc198627708)

[1. Цель работы 6](#_Toc198627709)

[2. Предметная область 7](#_Toc198627710)

[2.1 Сетевое процессорное устройство 7](#_Toc198627711)

[2.2 Классификация пакетов 8](#_Toc198627712)

[2.3 Хеширование данных 9](#_Toc198627713)

[3. Постановка задачи 11](#_Toc198627714)

[3.1 Задачи таблицы классификации 11](#_Toc198627715)

[3.2 Задача представления таблицы классификации 12](#_Toc198627716)

[4. Обзор метода 13](#_Toc198627717)

[4.1. Значимые критерии 13](#_Toc198627718)

[4.2. Отелло хеширование 13](#_Toc198627719)

[5. Исследование и решение задачи 15](#_Toc198627720)

[5.1. Основные понятия 15](#_Toc198627721)

[5.2. Операция поиска 15](#_Toc198627722)

[5.3. Операция построения структуры 16](#_Toc198627723)

[5.4. Операция вставки 17](#_Toc198627724)

[5.5. Операция удаления 18](#_Toc198627725)

[5.6. Группа параллельных Отелло. POG 18](#_Toc198627726)

[6. Программная реализация 19](#_Toc198627727)

[6.1 Описание имитационной модели 19](#_Toc198627728)

[7. Экспериментальное исследование 20](#_Toc198627729)

[7.1 Цель исследования 20](#_Toc198627730)

[7.2 Методика экспериментов 20](#_Toc198627731)

[7.3 Результаты экспериментов 21](#_Toc198627732)

[7.4 Выводы 24](#_Toc198627733)

[Заключение 25](#_Toc198627734)

[Литература 26](#_Toc198627735)

# Введение

В современном мире сетевые технологии играют важнейшую роль в обеспечении коммуникации и передачи данных. Развитие технологий и растущие потребности общества требуют более эффективных решений в области сетевых взаимодействий. Ключевыми устройствами, отвечающими за высокую пропускную способность и минимизацию задержек в компьютерных сетях, являются коммутаторы и маршрутизаторы. Они выполняют обработку сетевых пакетов, определяют их маршруты, управляют приоритетами передачи данных, выявляют и блокируют вредоносный трафик и выполняют другие важные функции.

В настоящее время активно разрабатываются программируемые коммутаторы, отличительной чертой которых является реализация обработки пакетов на уровне программного обеспечения, а не аппаратного. Их центральный компонент — сетевое процессорное устройство (СПУ) [1] — позволяет гибко настраивать логику обработки данных, обеспечивая при этом высокую производительность. Такие коммутаторы являются важной частью программно-конфигурируемых сетей (ПКС), но также находят применение и в традиционных сетевых архитектурах.

Использование программной обработки сетевых пакетов в СПУ позволяет поддерживать широкий спектр протоколов и работать на различных уровнях модели OSI. Однако такая гибкость сопровождается увеличенными требованиями к памяти, необходимой для хранения алгоритмов обработки, а также снижением скорости обработки пакетов. Для решения этой проблемы требуется оптимизация структур данных, отвечающих за хранение правил маршрутизации и обработки.

Одним из ключевых методов оптимизации является хеширование данных, которое позволяет каждой хранимой единице назначить уникальный идентификатор, упрощающий и ускоряющий её поиск. Введение хеширования в процесс классификации пакетов может существенно сократить время его выполнения, приближая его к константному, однако это требует дополнительных затрат памяти.

Основная сложность хеширования связана с коллизиями, возникающими, когда разные данные получают одинаковый идентификатор. Чем больше выделенной памяти, тем ниже вероятность таких коллизий. Однако существуют различные методы хеширования, позволяющие минимизировать избыточное использование памяти без значительного увеличения времени поиска. В данной работе рассмотрен и изучен один из таких методов.

Структура работы. В первой главе сформулированы цели исследования. Вторая глава содержит описание предметной области, в рамках которой осуществляется данное исследование. В третьей главе приведено математическое описание рассматриваемых объектов и дана формулировка задачи в математических терминах. Четвёртая / пятая глава посвящена анализу существующих методов решения поставленной задачи. В пятой главе предложены подходы к решению задачи, основанные на результатах проведённого обзора. В шестой главе изложена структура имитационной модели, использованной в рамках исследования. В седьмой главе представлены описание экспериментального исследования, анализ полученных результатов и сформулированные на их основе выводы.

# Цель работы

Цель данной работы — исследование применимости метода Отелло хеширования для представления таблиц классификации в СПУ (сетевом процессорном устройстве).

Для достижения этой цели планируется:

1. Изучить метод Отелло хеширования на основе описанного в статье алгоритма [1];
2. Разработать модель представления таблиц классификации с использованием этого метода хеширования;
3. Реализовать предложенную модель на практике;
4. Выполнить экспериментальный и сравнительный с другими классическими методами хеширования анализ эффективности.

# Предметная область

## 2.1 Сетевое процессорное устройство

Сетевое процессорное устройство — это специализированный микропроцессор, предназначенный для обработки и управления сетевым трафиком в программируемых коммутаторах. Архитектура, пример которой представлен на рисунке 1, СПУ представляет собой баланс между высокой производительностью, характерной для узкоспециализированных решений, и гибкостью настройки, присущей процессорам общего назначения. Благодаря этому СПУ поддерживает широкий спектр сетевых стандартов и протоколов, обеспечивая при этом высокую пропускную способность.

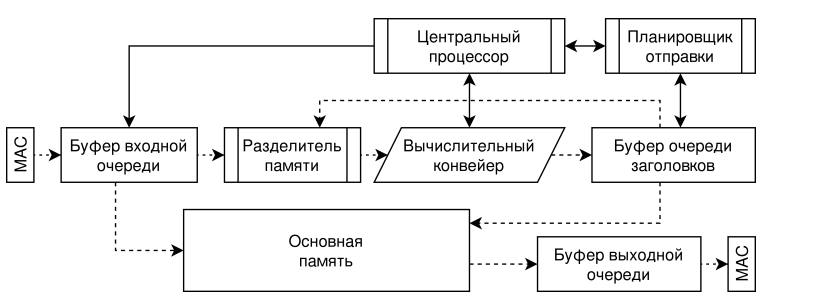


Рис. 1: Пример архитектуры СПУ

Как правило, обработка трафика в СПУ осуществляется с использованием конвейера вычислительных ядер. Такой подход позволяет разбить процесс обработки пакетов на последовательные этапы, что значительно ускоряет обработку данных. Однако задержка на любой из стадий конвейера может повлиять на общую производительность системы. Обработка сетевого пакета в СПУ включает следующие этапы:

1. прием пакета с входного порта;
2. извлечение заголовка пакета и временное сохранение его тела во внутренней памяти;
3. классификация пакета, заключающаяся в поиске соответствующего правила в таблице классификации;
4. модификация заголовка в соответствии с найденным правилом;
5. объединение измененного заголовка с телом пакета;
6. передача пакета в систему управления очередями;
7. отправка пакета на выходной порт согласно определенному правилу.

Третий шаг, классификация пакета — является одним из самых ресурсоемких этапов. Он представляет собой поиск по соответствующей таблице классификации, содержащей правила обработки пакетов.

## 2.2 Классификация пакетов

Таблица классификации представляет собой структуру данных, содержащую информацию о способе обработки пакетов определенного типа. Записи в таблице, называемые правилами, могут быть упорядочены по приоритету. Каждое правило включает в себя несколько полей: ключ, определяющий условия сопоставления, и значение, задающее соответствующие действия.

Ключ правила формируется на основе определенных частей заголовков пакета, таких как исходный и целевой адреса, порты, тип протокола и другие параметры. В зависимости от задачи ключ может быть задан как точное значение (константа) или как шаблон (wild-card), например, префикс IP-адреса. Поиск по таблице классификации может выполняться либо по полному соответствию, либо по частичному, например, по наиболее длинному совпадающему префиксу.

Значение правила играет различные роли: оно может представлять собой константу, используемую при обработке пакета, либо ссылку на исполняемый код, предназначенный для обработки пакетов данного типа. Такой подход особенно распространен в таблицах потоков (Flow table).

В данной работе рассматриваются таблицы классификации типа «MAC-VLAN» (таблица 1), работающие на втором (канальном) уровне модели ISO-OSI.

|  |  |
| --- | --- |
| MAC-VLAN table | |
| MAC-VLAN | port |
| xx:xx:xx:xx:xx:xx-xxx  (x – 4 бита) | 1 |
| … | … |
| xx:xx:xx:xx:xx:xx-xxx | n |

Таблица 1: Таблица классификации «MAC-VLAN»

Они позволяют сопоставлять MAC-адреса с виртуальными сетями VLAN, обеспечивая эффективную маршрутизацию трафика.

Правила в таких таблицах имеют следующий формат:  
<{MAC-addr, VLAN-id}, port-mask>, где:

* {MAC-addr, VLAN-id} — ключ, представляющий собой 60-битную строку, полученную путем объединения MAC-адреса и идентификатора VLAN;
* port-mask — значение, представленное 8-битной строкой, указывающей, на какие порты следует направить пакет после обработки.

Поиск в таблице классификации MAC-VLAN осуществляется по точному совпадению ключа.

## 2.3 Хеширование данных

Хеш-таблица [2; 3] представляет собой структуру данных, организованную в виде ассоциативного массива, который обеспечивает эффективное хранение и извлечение пар ключ-значение. Простейший вариант такой таблицы показан на рисунке 2. На этапе инициализации массив заполняется специальными значениями, обозначающими пустые ячейки (NULL).

Основой работы хеш-таблицы являются одна или несколько хеш-функций, преобразующих множество ключей в множество хеш-кодов. Хеш-код представляет собой индекс массива, по которому можно получить доступ к соответствующей записи.

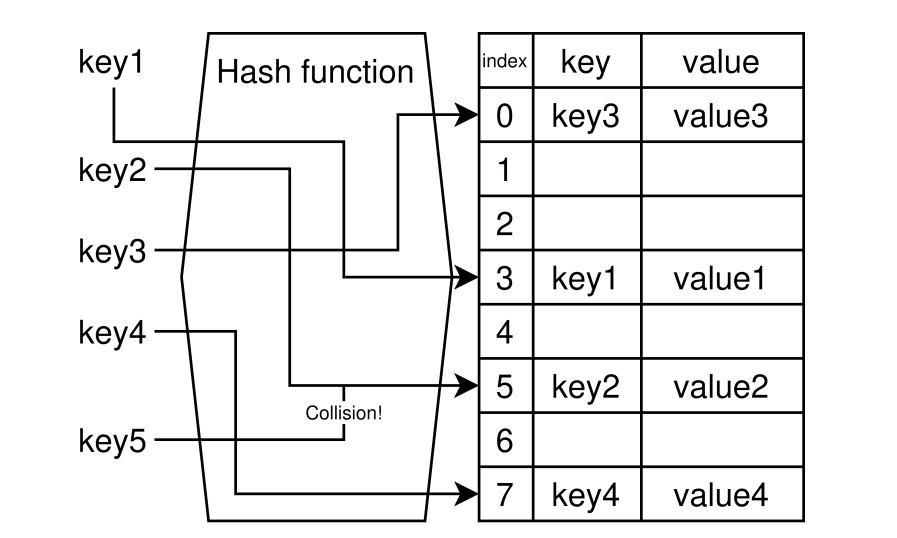


Рис. 2: Пример хеш-таблицы

Для использования в структурах данных хеш-функции должны обладать рядом свойств:

1. Фиксированная длина выходных данных – все хеш-коды находятся в заранее заданном диапазоне;
2. Сюрьективность – каждому ключу соответствует определенный хеш-код, который не обязательно является уникальным;
3. Равномерное распределение – хеш-коды должны быть распределены как можно более равномерно по множеству ключей;
4. Лавинный эффект – небольшое изменение входных данных в общем случае может приводить к значительному изменению хеш-кода, что обеспечивает учет каждого бита ключа.

В теории хеш-функции позволяют получать доступ к данным за константное время, не зависящее от объема хранимой информации. Если удается подобрать хеш-функцию, обеспечивающую биективное отображение ключей в хеш-коды, такое хеширование называют идеальным. Однако при расширении множества ключей идеальное хеширование, как правило, теряет свои преимущества.

Одна из ключевых проблем при работе с хеш-таблицами — коллизии, возникающие, когда разные ключи получают одинаковый хеш-код. В таких ситуациях необходимо применять методы разрешения коллизий, чтобы корректно сохранить и извлечь данные. Наиболее распространенный способ — метод хеширования цепочками: массив ячеек хранит ссылки на связанные списки, в которых размещаются записи с одинаковым хеш-кодом, однако со временем такая структура данных может деградировать и перестать обладать стандартными преимуществами хеш-таблиц и станет более похожа на классический список.

Для минимизации коллизий важным свойством хеш-функции является равномерность распределения хеш-кодов, что повышает эффективность и отказоустойчивость хеш-таблицы. Однако в некоторых случаях коллизии могут накапливаться до такой степени, что добавление новых записей становится невозможным. В таких ситуациях применяют перехеширование — смену хеш-функции и пересборку структуры.

Дополнительный инструмент для улучшения распределения хеш-кодов — соль (salt), представляющая собой дополнительный константный параметр хеш-функции. Использование соли позволяет расширить множество возможных хеш-функций и повышает вероятность выбора оптимальной функции для текущего множества ключей.

# Постановка задачи

## 3.1 Задачи таблицы классификации

Исходная постановка задачи описана в [X тут статья]. Приведем её постановку в контексте решаемой задачи.

Таблица классификации MAC-VLAN представляет собой структурированный набор правил, организованный в соответствии с определенным методом представления. Этот метод решает три основные задачи: поиск правила по заданному ключу, добавление новой записи и удаление существующей. В рамках описания данных задач используются следующие обозначения:

Набор правил:

*,*

Где

*;*

*i –* номер правила в таблице.

Правило представляет собой совокупность ключа (key) и значения (value), либо является пустым NULL и в классификации не участвует. Ключ является битовым значением заданной длины , значение также является полем определенной длины и соответствует порту в таблице классификации.

**Задача построения структуры данных**

**Дано:**

Набор правил таблицы классификации MAC-VLAN .

**Требуется:**

Построить структуру данных Отелло для последующих операций поиска в ней.

**Задача поиска правила**

**Дано:**

Ключ .

**Требуется:**

Найти правило вида , если оно представлено в исходной таблице, в противном случае вернуть NULL.

**Задача вставки правила**

**Дано:**

Правило , ранее не присутствовавшее в таблице.

**Требуется:**

Получить такую таблицу , что и для любого . Новая таблица содержит добавленное правило и доступна его операция поиска.

**Задача удаления правила**

**Дано:**

Ключ .

**Требуется:**

Получить такую таблицу , что и для любого . В новой таблице будет отсутствовать удалённое правило и его поиск возвращает результат NULL.

## 3.2 Задача представления таблицы классификации

**Дано:**

Набор констант:

где S – операция поиска, I – операция вставки, D – операция удаления.

**Требуется:**

Разработать способ представления таблицы классификации , такой чтобы для любого

*,*

*.*

# **Описание метода**

## 4.1. Значимые критерии

При анализе метода хеширования и оценке его эффективности в рамках решаемой задачи по классификации основными критериями являются:

* количество обращений к памяти во время процедуры поиска;
* количество вызовов хеш-функций во время процедуры поиска;
* количество обращений к памяти во время процедуры вставки;
* количество вызовов хеш-функций во время процедуры вставки;
* количество обращений к памяти во время процедуры удаления;
* количество вызовов хеш-функций во время процедуры удаления;
* ожидаемая максимально допустимая плотность записей.

## 4.2. Отелло хеширование

СХЕМА

Отелло-хеширование[4] в своей классической реализации без дополнительных улучшений алгоритма является структурой данных, которая позволяет разделить множество ключей или правил на два подмножества, опираясь при этом на использование двудольного графа, двух битовых массивов и пары различных хеш-функций. Размер этих массивов, как и диапазон значений, возвращаемых хеш-функциями, соответствует размеру долей графа. Каждый ключ отображается на ребро графа, при этом его вершины определяются значениями хеш-функций, вычисленных от ключа. Важно, чтобы граф оставался ацикличным. Доли графа связаны с соответствующими массивами, а каждая вершина, обладающая определенным цветом в соответствии с алгоритмом раскраски, отображается на конкретную ячейку своего массива.

Для задачи классификации сетевых пакетов Отелло хеш используется как базовая структура данных при реализации алгоритма Parallel Othello Group «POG» или же «группа параллельно работающих Отелло». Структура данных Отелло хеширования работает в двух различных плоскостях: плоскость данных и плоскость управления.

* **Структура запроса Othello (**Othello query structure**)**, предлагаемая к реализации на коммутаторе. Выполняет только запросы имен. Выделяется своей эффективностью по используемой памяти, времени выполнения запроса и полным отсутствием операций обновления.
* **Структура управления Othello (**Othello control structure**)**, предлагаемая к реализации на контроллере. Используется для операций добавления, удаления, изменения элементов структуры Отелло и рассылки этой информации на маршрутизаторы.

В Отелло хешировании принадлежность ключа к определённому подмножеству определяется с помощью побитового сложения (XOR) значений в соответствующих ячейках массивов. Повторное условно параллельное применение данного подхода позволяет разделить множество ключей на различных групп.

При вставке нового ключа сначала вычисляются значения хеш-функций, определяющие вершины графа и адреса ячеек в массивах. Затем проверяется, приведёт ли добавление нового ребра между этими вершинами к образованию цикла. Если цикл не возникает, ребро добавляется в граф. Далее, для корректного распределения и сохранения логики разбиения, изменяются значения битов в массивах вдоль достижимых путей от новых вершин. Такая коррекция возможна именно при отсутствии циклов.

При удалении достаточно просто убрать ребро из графа — при этом изменение битов в массивах не требуется.

Основным преимуществом Отелло хеширования является использование небольшого объема занимаемой памяти при хранении битовых массивов для эффективной работы в задаче классификации, а также фиксированное число вызовов хеш-функций и битовых операций XOR.

## 4.3. Основные понятия

Из постановки задачи известно, что множество всех имен (правил) в таблице классификации есть *,* пусть *.*

Отелло классифицирует *n* имен в два непересекающихся множества: .

Отелло можно расширить для классификации имен в d (d > 2)непересекающихся множеств. Подобная модификация структуры данных будет рассмотрена в разделе 5.6.

Как структура данных Отелло представляет собой кортеж из 7 значений, то есть каждое из которых определяется следующим образом:

* Целые числа и , описывающие размер Отелло, то есть битовых массивов, использующихся для классификации;
* Пара равномерных случайных хеш-функций *ha, hb*, сопоставляющих имена с целыми значениями *{0,1,··· ,ma−1}* и *{0,1,··· ,mb−1}* соответственно. Выбор хеш-функций является принципиально важным фактором для успешной работы структуры поиска;
* Битовые массивы *a* и *b*. Длины равны *ma* и *mb* соответственно;
* Двудольный граф G. Во время построения и обновления Отелло G используется для определения значений в a и b.

Возможны различные варианты указания длин массивов *a* и *b*. Требуется, чтобы , наилучшим вариантом является . Данное предположение будет проверено опытным путём.

## 4.4. Операция поиска

**Сложность:** О(1)

**Дано:**

Искомый ключ k.

**Требуется:**

Определить принадлежность правила с ключом k к множеству X или Y.

Для ключа *k* по соответствующему алгоритму вычисляется значение При . Если . Если Значения битовых массивов определяются во время построения Отелло, так что можно вычислить следующим образом:

## 4.5. Операция построения структуры

**Сложность:** О(n)

**Дано:**

Два набора имён , находящиеся в таблице классификации .

**Требуется:**

Построить структуру Отелло содержащую ациклический граф.

Граф G используется для определения пары хеш-функций и значений массивов .

.

Вершина , , что соответствует битам Существует ребро тогда и только тогда, когда такое что и , где – множество всех ключей ТК.

Для каждой вершины, которая связана хотя бы с одним ребром, соответствующий бит в массиве устанавливается в 0 или 1. Вершина, связанная с битом 0, окрашена в белый цвет, а вершина, связанная с битом 1, окрашена в черный цвет. Для вершин, которые не имеют связанных ребер, значение соответствующих битов может быть установлено в 0 или 1 произвольно, поскольку они не влияют ни на какое значение .

**Фаза I: Выбор пары хеш-функций.**

На этой фазе Отелло находит пару хеш-функций < На каждой попытке построения случайным образом выбираются две хеш-функции и создаётся граф G. Далее используется поиск в глубину (DFS) на G, чтобы проверить, содержит ли он цикл. Если два или более имени генерируют ребра с одинаковыми двумя конечными точками, будем считать, что есть цикл. Если G является циклическим, алгоритм будет выбирать другую пару хеш-функций до тех пор, пока не будет найден ациклический G.

**Фаза II: Вычисление битовых массивов.**

На этой фазе устанавливаются a и b.Сначала значения в a и b помечаются как неопределенные, что соответствует серому цвету вершин графа. Затем для каждого ребра в порядке DFS обхода:

* Если ни одно из значений не было установлено, тогда ;
* Если назначено только одно из , всегда можно назначить другое значение так, чтобы;
* Поскольку G ацикличен, следуя порядку DFS,мы никогда не увидим такое ребро, что и a[i], и b[j] имеютустановленныезначения.­

## 4.6. Операция вставки

**Сложность:** амортизированная О(1).

**Дано:**

Структура Отелло , добавляемое правило с ключом и принадлежностью к множеству X или Y.

**Требуется:**

Добавить ключ в Отелло так, чтобы стала доступна его операция поиска, возвращающая корректный результат.

Сначала алгоритм вычисляет ребро которое должно быть добавлено к G для , где При этом G можно разложить на компоненты связности, от расположения которых относительно добавляемого ребра возможны различные случаи вставки.

**Случай 1**

В битовых массивах есть неустановленные индексы, что соответствует серым вершинам в графе. Предположим, что в добавляемом ребре одна из вершин уже существует и покрашена, а другая нет. В таком случае действия эквиваленты алгоритму построения и значение пустой вершины будет установлено так, чтобы ребро соответствовало добавляемого правила.

**Случай 2**

принадлежат одному и тому же связному компоненту. Добавление ребра к графу может привести к появлению цикла. Необходимо повторно выбрать пару хеш-функций пока не будет найдена допустимая пара хеш-функций для нового набора имен , то есть заново применить операцию построения.

**Случай 3**

находятся в двух разных связных компонентах. Объединяя два связных компонента и новое ребро, мы получаем один связный компонент, который по-прежнему ацикличен, однако возможны подслучаи.

**Случай 3.1**

Добавляемое ребро удовлетворяет значению и перекраска/перестройка компоненты связности не требуется, так как можно однозначным образом классифицировать правило при операции поиска.

**Случай 3.2**

Если добавляемое ребро не удовлетворяет раскраске какой-либо компоненты связности, то необходимо обойти её в порядке DFS и произвести перекраску вершин. Утверждается, что хотя бы одна из двух компонент связности точно сохраняет свою раскраску.

## 4.7. Операция удаления

**Сложность:** О(1).

**Дано:**

Структура Отелло , удаляемое правило с ключом .

**Требуется:**

Удалить из графа G ребро, соответствующее правилу k, то есть .

Битовые массивы a и b не изменяются.

## 4.8. Группа параллельных Отелло. POG

Для классификации правил в таблице MAC-VLAN одной лишь структуры Отелло хеширования с её операциями недостаточно, потребуется модификация в виде группы параллельных Отелло. p-POG способна классифицировать правила в различных множеств, что соответствует ТК с p уникальными правилами коммутации.

Пусть – множества уникальных портов ТК. Для вычисления порта задействовано p Отелло, каждое из которых вычисляет соответствующий бит искомого числа, после чего оно переводится в десятичную систему счисления. Таким образом, результирующее

При этом для каждого Отелло определены правила, по которым она формирует свой собственный граф G и битовые массивы a, b на этапе построения. Принадлежность же правила к множеству X или Y определяется его переводом в двоичную систему счисления и индексом отвечающего за конкретный бит Отелло.

# **Программная реализация**

## 6.1 Описание имитационной модели

Имитационная модель (ИМ), реализованная на языке программирования Python версии 3.11, воспроизводит поведение различных методов организации таблицы классификации в условиях, заданных экспериментом. Структурная иерархия модели представлена на рисунке 3.

В процессе выполнения каждой процедуры в рамках ИМ формируется следующий набор данных:

* тип выполняемой процедуры;
* число обращений к памяти;
* количество вызовов хеш-функции;
* время выполнения данной операции;
* результат выполнения процедуры (успешно/неуспешно).

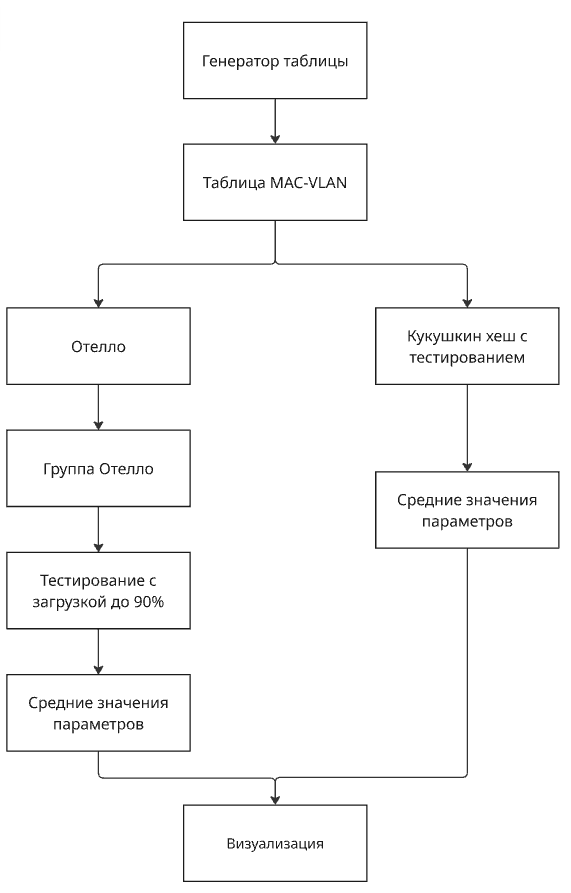


Рис. 3: Иерархия имитационной модели

# 7. Экспериментальное исследование

## 7.1 Цель исследования

Цель настоящего исследования заключается в количественной оценке обращений к памяти и вызовов хеш-функции, а также времени работы каждого из методов, возникающих в процессе выполнения описанных операций вставки, поиска и удаления правил таблицы MAC-VLAN в условиях, характеризующихся следующими параметрами:

* коэффициент заполнения хеш-таблицы достигает значения 20%;
* производится измерение как средних, так и максимальных значений числа указанных операций и времени;
* эксперимент проводится при объёмах данных, соответствующих 1 тысяче записей (правил).

Отметим, что небольшой уровень загрузки таблицы обусловлен неспособностью классического Кукушкиного хеша [5], используемого для сравнения с реализованным методом Отелло хеширования, обрабатывать фактор загрузки выше половины от имеющейся выделенной памяти под таблиц. При большом количестве правил время работы и количество операций Кукушкиного хеша становится вовсе несравнимы с реализованным в рамках курсовой методом Отелло.

## 7.2 Методика экспериментов

Для достижения требуемой плотности размещения записей, равной 20%, были выбраны следующие размеры массивов: 5000 элементов для набора, содержащего 1000 правил.

В реализациях Отелло хеширования и Кукушкиного хеша для упрощения экспериментов используются достаточно тяжеловесные, но зарекомендовавшие себя и использующиеся для широкого круга задач хеш-функции семейства SHA [6].

В случае Кукушкиного хеша таблица классификации заполняется постепенно добавлением новых правил, а Отелло хеширование строится сразу с помощью операции construct, которая по сути своей представляет аналогичное постепенное заполнение таблицы.

При этом выполняется следующий набор операций:

* вставка правила, отсутствующего в таблице;
* поиск правила, присутствующего в таблице;
* удаление присутствующего в таблице правила.

## 7.3 Результаты экспериментов

На диаграмме ниже приведены значения количества операций, выполняемых в процедурах для достижения плотности данных 20%. На рисунке 4 представлены результаты измерений числа обращений к памяти при использовании 1000 правил. Анализ данных графиков показывает, что среднее количество обращений к памяти для Отелло хеширования практически константно для всех операций и зависит исключительно от числа портов в таблице MAC-VLAN, то есть от уникальных множеств для группы параллельных Отелло. Отдельно отметим, что при удалении правила обращений к памяти не происходит вовсе.

В случае же Кукушкиного хеша наблюдается многократный рост в числе обращений к памяти на операции вставки нового правила. Обусловлено это тем, что при наличии цикла на этапе выбора позиций для размещения данных правила этот метод хеширования вынужден полностью перестраивать свою структуру практически в рекурсивном формате.

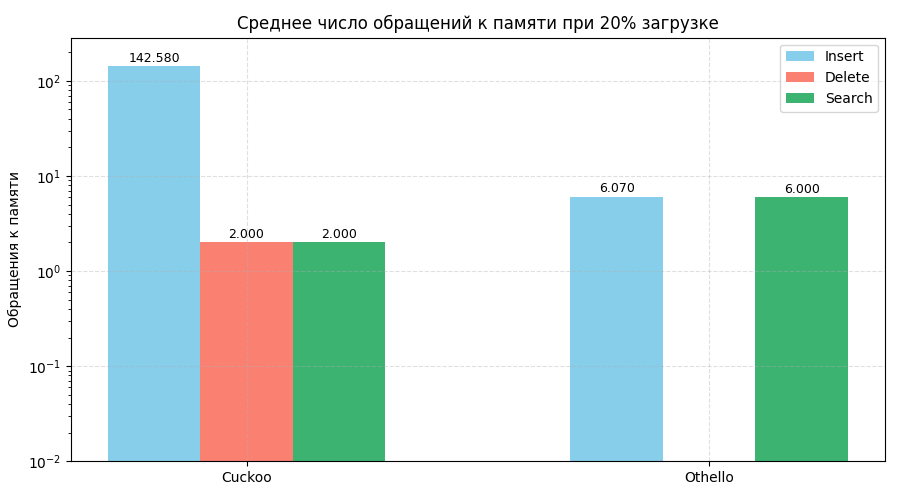
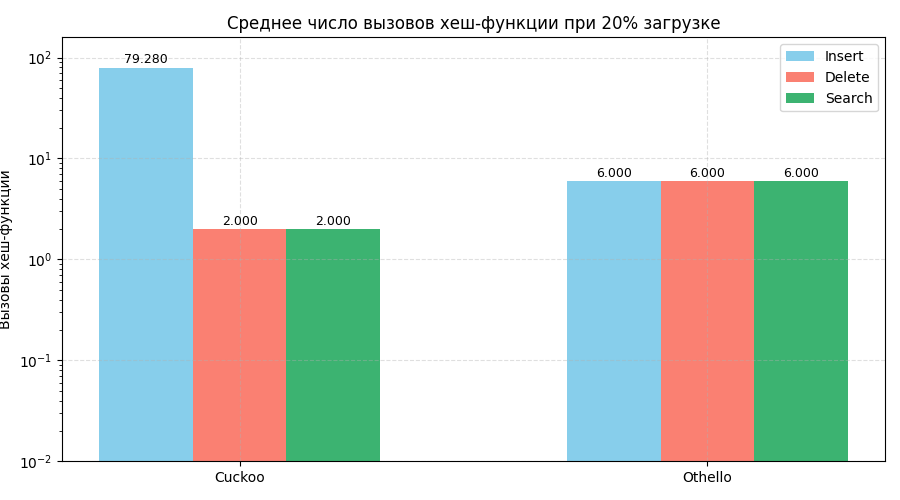


Рис. 4: Количество обращений к памяти при 1000 правил

На рисунке 5 изображено количество вызовов хеш-функции при 1000 тысячах правил. Заметим, что сохраняется тенденция диаграммы обращений к памяти: Отелло хеширование не зависит от числа правил при столь низкой загрузке и выполняет все перечисленные операции за константное число вызовов хеш-функций. При этом Кукушкин хеш из-за перестроения структуры и повторного перехеширования вынужден множество раз вызывать хеш-функции.

Рис. 5: Количество вызовов хеш-функции при 1000 правил

На рисунке 6 представлено среднее время выполнения каждой из описанных операций для реализованных методов при наличии 1000 правил в таблице. Заметим, что вставка для Отелло хеширования выполняет в несколько раз дольше, чем аналогичная операция для Кукушкиного хеша. Происходит это из-за того, что для данной операции необходимо выполнить множество команд по работе с графом, например, частичная перекраска вершин, что приводит к накладным расходам и увеличению средней времени работы операции. При этом все удаление и поиск в реализованных методах на небольшой нагрузке выполняются значительно быстрее.

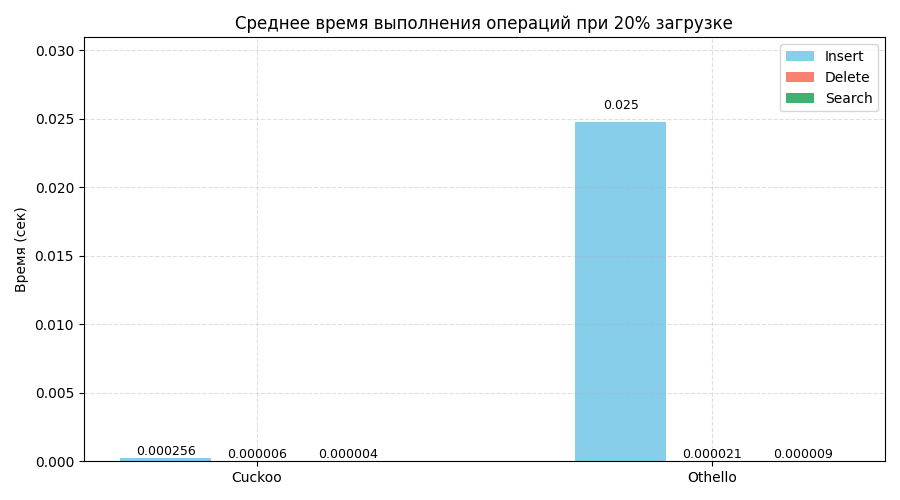


Рис. 6: Время выполнения операций при 1000 правил

## 7.4 Выводы

По результатам исследований можно сделать следующие выводы:

* Реализованный в настоящей работе способ Отелло хеширования значительно превосходит классический метод Кукушкиного хеширования в контексте обращений к памяти и вызовов хеш-функций и на низкой загрузке таблице практически не зависит от числа правил в ней;
* Отметим, что при достижении фактора загрузки таблицы в 50% и более по результатам многочисленных экспериментов с различными параметрами и правилами Кукушкин хеш не способен построить таблицу для классификации и на стандартном наборе хеш-функций не выходит из бесконечного цикла без возможности дополнительного увеличения занимаемой им памяти. При этом Отелло хеширования справляется с загрузкой таблицы до 95% и способен простроить свою структуру;
* Использование метода Отелло хеширование из-за наличия большого числа накладных расходов на работу с двудольным графом будет крайне целесообразно в ПКС с предварительно заданными и редко изменяющимися таблицами классификации MAC-VLAN, именно на них данный метод способен быстро и с минимальными затратами по памяти выполнять процесс классификации пакетов.

# Заключение

В данной работе проведено исследование метода Отелло хеширования и разработан эффективный подход к представлению таблицы классификации MAC-VLAN. Основные результаты можно сформулировать следующим образом:

1. На основании анализа существующих методов хеширования и исходной статьи авторов изучен и реализован метод Отелло хеширования;
2. Проведено экспериментальное исследование, подтвердившее эффективность реализованного метода по сравнению с классическими вариантами.

Анализ экспериментальных данных показал, что метод Отелло хеширования значительно превосходит классический Кукушкин хеш по числу обращений к памяти и вызовов хеш-функций даже на незначительной загрузке таблицы, при этом данный метод способен корректно функционировать и выполнять все описанные операции на загрузке превышающей 90%, что невозможно для Кукушкиного хеша без расширения занимаемой памяти. Однако в случае Отелло это сопровождается некоторыми затратами памяти и времени, обусловленными необходимостью хранения параметров двудольного графа. Это может стать критичным фактором при масштабировании системы.

Операции поиска и удаления в случае метода Отелло выполнятся эффективно и за константное время, что позволит в редко изменяющихся ПКС выполнять процесс классификации пакетов наилучшим образом.

В качестве направления для дальнейших исследований рассматривается возможность применения различных способов фильтрации [7] поступающих пакетов, что позволит не тратить время на поиск или удаление отсутствующих ключей в таблице, а также возможность использования модификаций классического Отелло хеширования таких как Лудо хеш [8].

# Литература

[1] Yu, Y., Belazzougui, D., Qian, C., Zhang, Q., 2018. Memory-efficient and ultra-fast network lookup and forwarding using Othello hashing. IEEE/ACM Transactions on Networking, 26(3), pp.1151-1164.

[1] Об одном подходе к построению сетевого процессорного устройства / С. О. Беззубцев [и др.] // Моделирование и анализ информационных систем. — 2019. — Т. 26, № 1. — С. 39—62.

[2] Maurer W. D., Lewis T. G. Hash table methods // ACM Computing Surveys (CSUR). — 1975. — Т. 7, № 1. — С. 5—19.

[3] A survey on learning to hash / J. Wang [и др.] // IEEE transactions on pattern analysis and machine intelligence. — 2017. — Т. 40, № 4. — С. 769—790.

[4] Yu, Y., Belazzougui, D., Qian, C., Zhang, Q., 2018. Memory-efficient and ultra-fast network lookup and forwarding using Othello hashing. IEEE/ACM Transactions on Networking, 26(3), pp.1151-1164.

[5] Pagh, R., Rodler, F.F., 2001, August. Cuckoo hashing. In European Symposium on Algorithms (pp. 121-133). Berlin, Heidelberg: Springer Berlin Heidelberg.

[6] Burhan Ul Islam Khan, Malik Arman Morshidi, Rashidah Funke Olanrewaju, Roohie Naaz Mir, 2022. Evolution and analysis of secured hash algorithm (sha) family.

[7] Dharmapurikar, S., Krishnamurthy, P. and Taylor, D.E., 2003, August. Longest prefix matching using bloom filters. In Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications (pp. 201-212).

[8] Shi, S., Qian, C., 2020. Ludo hashing: Compact, fast, and dynamic keyvalue lookups for practical network systems. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 4(2), pp.1-32.