Гудис А. В., Волканов Д. М.

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

## 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. В четвертой главе предложен подход к решению поставленной задачи и описан реализованный алгоритм. В пятой главе изложена структура имитационной модели, использованной в рамках исследования. В шестой главе представлены описание экспериментального исследования, анализ полученных результатов и сформулированные на их основе выводы.

# Постановка задачи

## 1.1 Задачи таблицы классификации

Исходная постановка задачи описана в [5]. 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

## **Описание метода**

## 2.1. Отелло хеширование

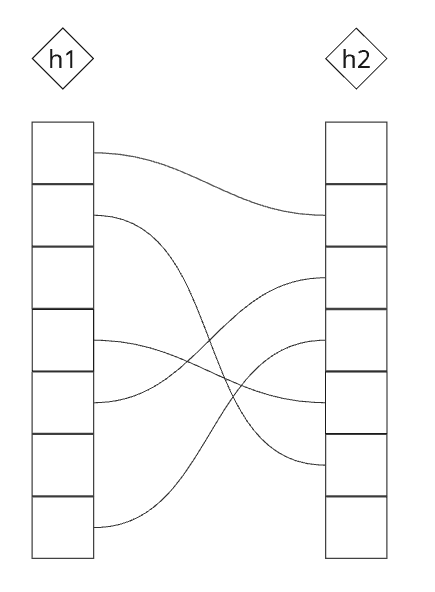


Рис. 3: Отелло хеширование. Общая схема.

Отелло-хеширование в своей классической реализации без дополнительных улучшений алгоритма является структурой данных, схема которой представлена на рисунке 3. Отелло хеширование 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.

## 2.3. 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,··· ,−1}* и *{0,1,··· ,−1}* соответственно. Выбор хеш-функций является принципиально важным фактором для успешной работы структуры поиска;
* Битовые массивы *a* и *b*. Длины равны и 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 .

## 2.4. Операция поиска

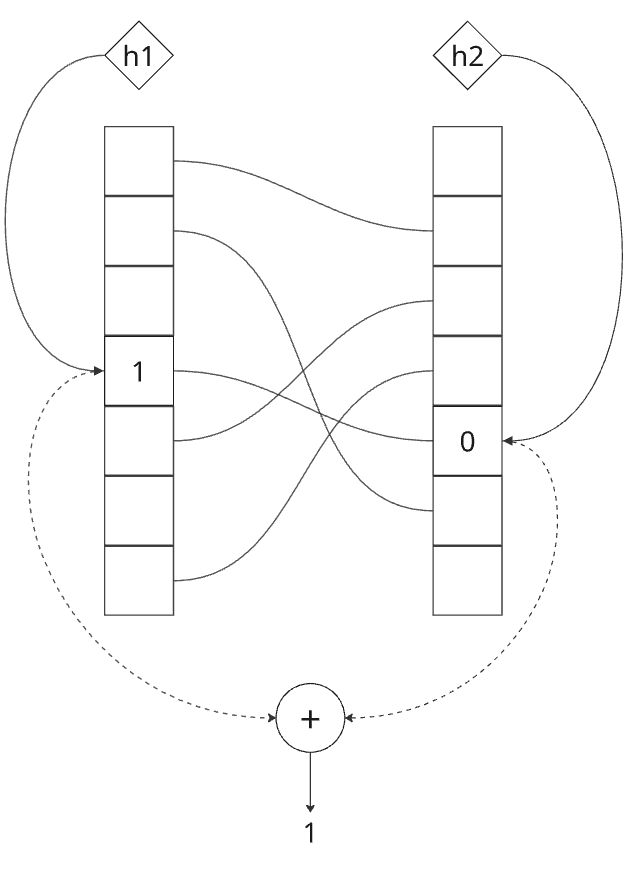


Рис. 4: Операция поиска.

**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:

## 2.5. 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.­

## 2.6. 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.

## 2.7. 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.

## 2.8. 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**

## 3.1 Описание имитационной модели

Имитационная модель (ИМ), реализованная на языке программирования Python версии 3.11, включает в себя несколько программных модулей и воспроизводит поведение различных методов организации таблицы классификации в условиях, заданных экспериментом. Структурная иерархия модели с основными модулями представлена на рисунке 5. Дополнительно используются сторонние библиотеки, такие как:

* networkx для реализации двудольного графа в Отелло хешировании и организации работы с ним;
* matplotlib для визуализации тестовых данных;
* numpy для упрощения работы с массивами тестовых данных при их визуализации.

А также ряд встроенных библиотек языка программирования Python:

* hashlib для получения хеш-функций;
* random для генерации тестовых правил ТК MAC-VLAN;
* time для замера времени работы основных методов реализованных хеш-структур;
* json для работы с автоматически сгенерированной ТК MAC-VLAN.

### 3.1 generate\_test

Модуль generate\_test отвечает за имитиацию получения таблицы классификации MAC-VLAN из внешней среды. Одним из аргументов, передаваемых на вход при запуске модуля, является количество правил итоговой синтетической ТК. Результатом работы модуля является таблица классификации MAC-VLAN в виде отдельного файла расширения .json с указанным числом правил.

### 3.2 othello

В данном модуле находится одноименный класс, описывающий всю необходимую функциональность классического Отелло хеширования. Реализованы такие методы как search, construct, insert и delete, отвечающие за поиск элемента, построение структуры по таблице, вставку нового правила и его удаление соответственно. Также в модуле описаны и реализованы методы работы с двудольным графом: его построение, перекраска и визуализация.

### 3.3. pog

Модуль pog описывает принципы и основные методы работы группы параллельных Отелло: подготовка каждого правила таблицы для разбиения под несколько Отелло структур, построение группы, добавление нового правила в таблицу, поиск и удаление правила.

### 3.4 cuckoo

Модуль описывает классический Кукушкин хеш с операциями вставки, поиска и удаления правила, а также отдельно вынесенной функцией тестирования, записывающий результаты в специальный файл.

### 3.5 ninety\_test

Данный модуль отвечает за тестирование Отелло хеширования с различной степенью загрузки битовых массивов. Результатом работы метода является текстовый файл с информацией о проведенном тестировании.

### 3.6 common

Объединяет в себе общие методы, необходимые для работы Отелло и Кукушкиного хеша: вывод отладочной информации о тестировании, визуализация полученных данных, различные функции по автоматической генерации правил MAC-VLAN таблицы, а также специализированный класс, отвечающий за сбор информации о числе вызовов хеш-функций и обращений к памяти в описанных выше хеш структурах.

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).

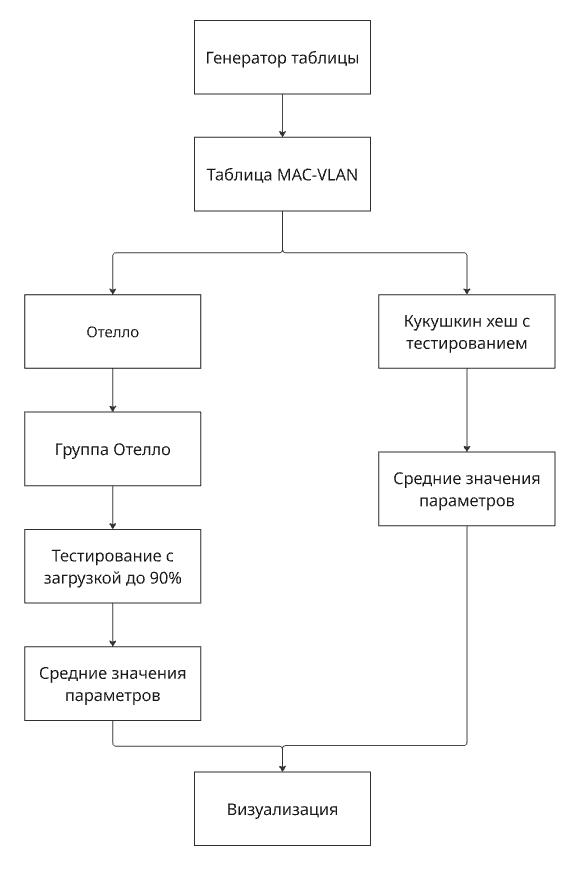


Рис. 5: Иерархия имитационной модели.

# 4. Экспериментальное исследование

## 4.1 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 [6], используемого для сравнения с реализованным методом Отелло хеширования, обрабатывать фактор загрузки выше половины от имеющейся выделенной памяти под таблицы. При большом количестве правил время работы и количество операций Кукушкиного хеша становится вовсе несравнимы с реализованным в рамках курсовой методом Отелло.

## 4.2 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 [7].

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.

## 4.3 Результаты экспериментов

На диаграмме ниже приведены значения количества операций, выполняемых в процедурах для достижения плотности данных 20%. На рисунке 6 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.

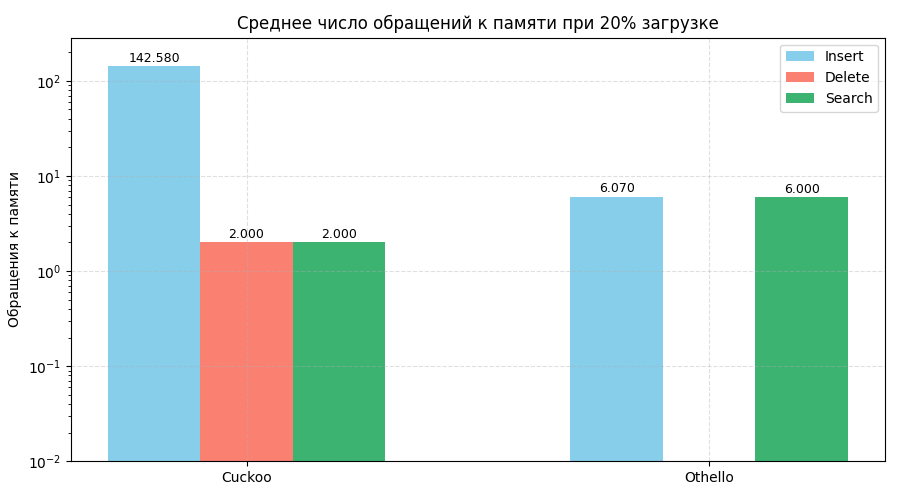
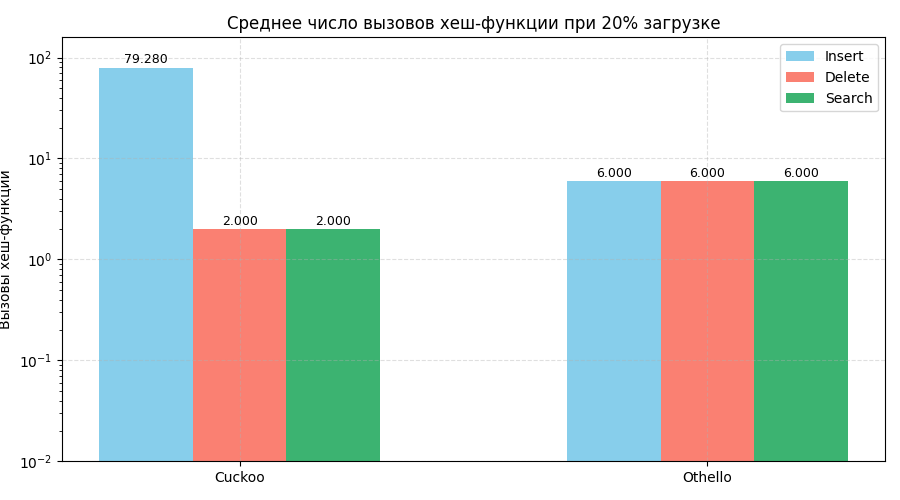


Рис. 6: Количество обращений к памяти при 1000 правил.

На рисунке 7 изображено количество вызовов хеш-функции при тысяче 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. 7: Количество вызовов хеш-функции при 1000 правил.

На рисунке 8 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.

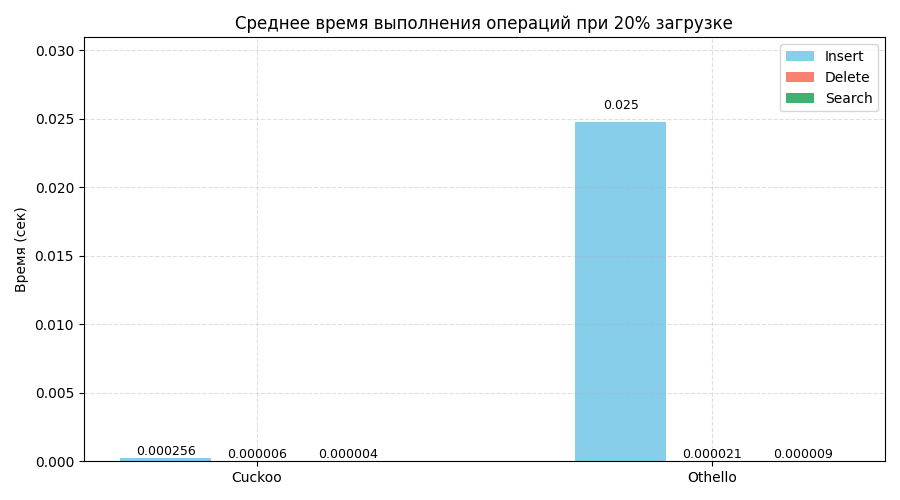


Рис. 8: Время выполнения операций при 1000 правил.

На рисунке 9 продемонстрирован график зависимости времени работы Отелло хеширования при операции вставки в зависимости от степени загрузки двудольных графов для таблицы из 1000 правил. Можно заметить, что время работы линейным образом растет с увеличением фактора загрузки, но при этом хеш структура функционирует и успешно выполняет операцию вставки за конечное время, что невозможно в случае Кукушкиного хеша, классическая реализация которого не позволяет превышать фактор загрузки выше половины.



# 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 [8] 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 [9].

# Литература

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

**[2]** Yu Y., Belazzougui D., Qian C., Zhang Q. Memory-efficient and ultra-fast network lookup and forwarding using Othello hashing // IEEE/ACM Transactions on Networking. — 2018. — Vol. 26, No. 3. — P. 1151–1164.

**[3]** Maurer W. D., Lewis T. G. Hash table methods // ACM Computing Surveys (CSUR). — 1975. — Vol. 7, No. 1. — P. 5–19.

**[4]** Wang J., и др. A survey on learning to hash // IEEE Transactions on Pattern Analysis and Machine Intelligence. — 2017. — Vol. 40, No. 4. — P. 769–790.

**[5]** Александров А. В., Волканов Д. Ю. Исследование хеширующих поисковых структур данных для таблицы классификации MAC-VLAN сетевого процессорного устройства // Ломоносовские чтения. Научная конференция. 20 марта – 3 апреля 2024 г. : тезисы докладов. — 2024. — С. 106–107.

**[6]** Pagh R., Rodler F. F. Cuckoo hashing // European Symposium on Algorithms. — Berlin, Heidelberg: Springer, 2001. — P. 121–133.

**[7]** Khan B. U. I., Morshidi M. A., Olanrewaju R. F., Mir R. N. Evolution and analysis of secured hash algorithm (SHA) family. — 2022.

**[8]** Dharmapurikar S., Krishnamurthy P., Taylor D. E. Longest prefix matching using Bloom filters // Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications. — 2003. — P. 201–212.

**[9]** Shi S., Qian C. Ludo hashing: Compact, fast, and dynamic key-value lookups for practical network systems // Proceedings of the ACM on Measurement and Analysis of Computing Systems. — 2020. — Vol. 4, No. 2. — P. 1–32.