

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

**«МИРЭА – Российский технологический университет»**

**РТУ МИРЭА**

Институт Кибернетики

Кафедра «Информационная безопасность» (БК №252)

**Отчёт**

по ПРЕДДИПЛОМНОЙ ПРАКТИКЕ

**тема:**

**«Алгоритм построения 2k-мультиколлизий для хэш- функции «Мора»**

Студент группы ККСО-0\_-15: Чижов Н.Д.

Руководитель работы: *зав. кафедрой, к.т.н., доцент Корольков А. В.*

Работа представлена к защите: «\_\_\_» декабря 2020 г.

Оценка: «\_\_\_\_\_\_\_\_\_\_\_\_»

Москва 2020

**Содержание**

[Введение 2](#_Toc59456267)

[Глава 1. Алгоритмы построения коллизий и мультиколлизий для функций сжатия и хэш-функций. 7](#_Toc59456268)

[1.1 Хэширование 7](#_Toc59456269)

[1.1.1 Хэш-функция Меркла-Дамгарда 10](#_Toc59456270)

[1.1.2 Конструкция Wide Pipe 13](#_Toc59456271)

[1.1.3 Конструкция HAIFA 14](#_Toc59456272)

[1.1.4 Криптографическая губка 15](#_Toc59456273)

[1.2 Методы разрешения коллизий 16](#_Toc59456274)

[1.2.1 Алгоритмы хэширования 19](#_Toc59456275)

[1.2.2 Таблица прямого доступа 19](#_Toc59456276)

[1.2.3 Метод остатков от деления 20](#_Toc59456277)

[1.2.4 Метод функции середины квадрата 21](#_Toc59456278)

[1.2.6 Метод свертки 21](#_Toc59456279)

[1.2.5 Открытое хэширование 22](#_Toc59456280)

[1.2.7 Закрытое хэширование 25](#_Toc59456281)

[Глава 2. Алгоритмы и структурам данных для построения атак на хэш-функции с контрольными суммами. 27](#_Toc59456282)

[2.1 Анализ методов построения мультиколлизий 28](#_Toc59456283)

[2.2 Обобщенная конструкция хэширования 30](#_Toc59456284)

[2.3 Определение методов усложнения построения мультиколлизий 33](#_Toc59456285)

………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………

# **Введение**

**Актуальность.** Криптографические методы не только позволяют обеспечивать конфиденциальность, но и контролировать целостность передаваемых данных. Контроль целостности, как правило, осуществляется методом вычисления контрольной суммы данных. В настоящее время разработано большое число алгоритмов, вычисляющих контрольные суммы передаваемых данных. Дело в том, что в большинстве случаев достаточно простой контрольной суммы, например, в тех случаях, когда объем информации не известен заранее или имеет значение скорость обработки. Но при использовании простой контрольной суммы можно подобрать несколько массивов данных, у которых будет получаться одинаковая контрольная сумма. Криптографически стойкие контрольные суммы получаются в случае применения к исходному тексту хэш-функции. Возьмем ситуацию: необходимо отправить по сети документ и нужна абсолютная уверенность в том, что полученный файл совершенно идентичен оригиналу и содержащиеся в нем цифры не были изменены «в пути», т.е. целостность документа была не нарушена. Для этого существуют алгоритмы идентификации цифровой информации. Такие алгоритмы работают только с цифровой информацией. При необходимости идентификации текстового документа или изображения, нужно с помощью кодирования вначале получить цифровой аналог данной информации. Различают несколько способов идентификации: контрольные суммы, контроль CRC, хэширование и цифровая подпись – таковы базовые средства аутентификации при цифровой передаче данных.

Контрольные суммы – наиболее простой способ проверки целостности данных, передаваемых в цифровом виде, при котором вычисляется контрольная сумма сообщения (определенное значение, которое и идентифицирует цифровую информацию).

Контроль CRC – более надежный способ цифровой идентификации данных. Это вычисление контрольного значения циклического избыточного кода данных (cyclic redundancy check – CRC).

Наиболее совершенный способ идентификации цифровой информации – алгоритмы цифровой подписи и хэширование.

Функция хэширования H (m) или хэш-функция (hash-function) –

это детерминированная функция, на вход которой подается строка битов произвольной длины, а выходом всегда является битовая строка фиксированной длины n.

Значение хэш-функции H (m) для входа m называют хэшем. В литературе можно широко распространены и другие названия, а именно Хэш = хэш-образ= хэш-код = свертка = дайджест сообщения = криптографическая контрольная сумма = цифровой отпечаток = код аутентичности сообщения = код обнаружения манипуляций. Исходная строка m, для которой вычислено хэш-значение, называется прообразом хэш-функции.

Свойства, которые должны быть присущи криптографическим хэш- функциям:

1. **Сопротивление поиску первого прообраза**: при наличии хэша h должно быть трудно найти какое-либо сообщение m, такое что h = hash(m), Это свойство связано с понятием односторонней функции. Функции, у которых отсутствует это свойство, уязвимы для атак нахождения первого прообраза.
2. **Сопротивление поиску второго прообраза**: при наличии сообщения m1, должно быть трудно найти другое сообщение m2 (не равное m1) такое, что hash(m1) = hash(m2). Это свойство иногда называют слабым сопротивлением поиску коллизий. Функции, у которых отсутствует это свойство, уязвимы для атак поиска второго прообраза.
3. **Стойкость к коллизиям** Коллизией для хэш-функции называется такая пара значений m1 и m2, , для которой hash(m1) = hash(m2). Так как количество возможных открытых текстов больше числа возможных значений свёртки, то для некоторой свёртки найдётся много прообразов, а следовательно, коллизии для хэш-функций обязательно существуют. Например, пусть длина хэш-прообраза 6 битов, длина свёртки 4 бита. Тогда число различных свёрток — , а число хэш-прообразов — , то есть в 4 раза больше, значит хотя бы одна свёртка из всех соответствует 4 прообразам.
4. **Псевдослучайность**: должно быть трудно отличить генератор псевдослучайных чисел на основе хэш-функции от генератора случайных чисел.

Таким образом, основная цель нашей выпускной квалификационной работы **-** разработка алгоритмов и реализация программного обеспечения для построения мультиколлизий хэш-функции «Мора».

Объектом исследования являются построения мультиколлизий для хэш функции.

Предметом исследования являются проблемы и перспективы использования построения атак на хэш-функции с контрольными суммами.

Для достижения указанных целей перед исследованием ставятся следующие основные задачи:

1. Провести обзор существующих алгоритмов построения коллизий и мультиколлизий для функций сжатия и хэш-функций на основе схемы Меркла-Дамгарда.

2. Провести обзор существующих алгоритмов и структур данных для построения атак на хэш-функции, использующие контрольные суммы.

3. Разработать алгоритмы построения мультиколлизий для хэш-функции «Мора». Оценить их сложности по времени и памяти.

4. Программная реализация разработанных алгоритмов.

5. Провести экспериментальные исследования с использованием разработанного программного обеспечения.

6. Обобщить результаты исследования и сформулировать выводы по всей работе.

Для решения поставленных в дипломной работе задач использованы следующие методы научного исследования: теоретические: анализ и систематизация теоретических и практических данных с целью определения актуальности и состояния разработки указанной проблемы; эмпирические: исследование, контрольный эксперимент, формирующий эксперимент.

Теоретическая значимость исследования заключается в том, что результаты работы могут послужить стимулом к новым практическим экспериментам в области исследований построения мультиколлизий. Практическая значимость заключается в исследовании алгоритма построения 2k-мультиколлизий для хэш функции «Мора», которые можно успешно внедрять в способы идентификации цифровой информации.

Методологической и теоретической основой исследования послужили работы зарубежных и российских авторов в области алгоритмов построения коллизий и мультиколлизий для функции сжатия и хэш-функций..

Структура и объем дипломной работы. Структура дипломной работы подчинена логике исследования и состоит из введения, трех глав, заключения и списка литературы. Работа содержит n страниц, m изображений, k таблиц, l приложений. В выпускной квалификационной работе используется j источников, что помогает наилучшим образом раскрыть заявленную тему.

Первая и вторая глава - теоретический анализ данной проблемы. Третья глава – анализ метода построения мультиколлизий для хэш-функции «Мора».

# **ГЛАВА 1. АЛГОРИТМЫ ПОСТРОЕНИЯ КОЛЛИЗИЙ И МУЛЬТКОЛЛИЗИЙ ДЛЯ ФУНКЦИЙ СЖАТИЯ И ХЭШ-ФУНКЦИЙ**

# **Хэширование**

В настоящее время количество хранимой информации стремительно растет. Это влечет за собой появление множества новых задач, связанных с хранением, сортировкой, поиском и другими видами обработки информации. Также важнейшим моментом является обеспечение безопасности и надежной передачи информации.

Безопасность всегда была неоднозначной темой, провоцирующей многочисленные горячие споры. И всё благодаря обилию самых разных точек зрения и «идеальных решений», которые устраивают одних и совершенно не подходят другим. Есть мнение, что взлом системы безопасности приложения

* всего лишь вопрос времени. Из-за быстрого роста вычислительных мощностей и увеличения сложности безопасные сегодня приложения перестанут завтра быть таковыми [9].

Процесс поиска данных в больших объемах информации сопряжен с временными затратами, которые обусловлены необходимостью просмотра и сравнения с ключом поиска значительного числа элементов. Сокращение поиска возможно осуществить путем локализации области просмотра. Например, отсортировать данные по ключу поиска, разбить на непересекающиеся блоки по некоторому групповому признаку или поставить в соответствие реальным данным некий код, который упростит процедуру поиска.

В настоящее время используется широко распространенный метод обеспечения быстрого доступа к информации, хранящейся во внешней памяти – хэширование [15]. Хэширование полезно, когда широкий диапазон возможных значений должен быть сохранен в малом объеме памяти, и нужен способ быстрого, практически произвольного доступа. Хэш-таблицы часто применяются в базах данных, и, особенно, в языковых процессорах типа компиляторов и ассемблеров, где они повышают скорость обработки таблицы идентификаторов. В качестве использования хэширования в повседневной жизни можно привести примеры распределение книг в библиотеке по тематическим каталогам, упорядочивание в словарях по первым буквам слов, шифрование специальностей в вузах и т.д.

**Хэширование** (или хэширование, англ. Hashing ) – это преобразование входного массива данных определенного типа и произвольной длины в выходную битовую строку фиксированной длины [9]. Такие преобразования также называются хэш-функциями или функциями свертки, а их результаты называют хэшем, хэш-кодом, хэш-таблицей или дайджестом сообщения (англ. message digest ).

Хэш-таблица– это структура данных, реализующая интерфейс ассоциативного массива, то есть она позволяет хранить пары вида «ключ- значение» и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу. Хэш-таблица является массивом, формируемым в определенном порядке хэш-функцией [24].

Принято считать, что хорошей, с точки зрения практического применения, является такая хэш-функция, которая удовлетворяет следующим условиям:

- функция должна быть простой с вычислительной точки зрения;

-функция должна распределять ключи в хэш-таблице наиболее равномерно;

- функция не должна отображать какую-либо связь между значениями ключей в связь между значениями адресов;

- функция должна минимизировать число коллизий – то есть ситуаций, когда разным ключам соответствует одно значение хэш- функции(ключи в этом случае называются синонимами ) [4].

Коллизией хэш-функции hash называются два параметра a и b, при hash(a) = hash(b). На практике это означает, что двум значениям(символам или последовательностям символов) соответствует один ключ-значение. Коллизии существуют для большинства хэш-функкций, но для «хороших» хэш-функций частота их возникновения близка к теоретическому минимуму. В некоторых частных случаях, когда множество различных входных данных конечно, можно задать инъективную хэш-функцию, по определению не имеющую коллизий. Однако для хэш-функций, принимающих вход переменной длины и возвращающих хэш постоянной длины, коллизии обязаны существовать, поскольку хотя бы для одного значения хэш-функции соответствующее ему множество входных данных будет бесконечно — и любые два набора данных из этого множества образуют коллизию [7].

При этом первое свойство хорошей хэш-функции зависит от характеристик компьютера, а второе – от значений данных.

Если бы все данные были случайными, то хэш-функции были бы очень простые (например, несколько битов ключа). Однако на практике случайные данные встречаются достаточно редко, и приходится создавать функцию, которая зависела бы от всего ключа [13]. Если хэш-функция распределяет совокупность возможных ключей равномерно по множеству индексов, то хэширование эффективно разбивает множество ключей. Наихудший случай – когда все ключи хэшируются в один индекс.

При возникновении коллизий необходимо найти новое место для хранения ключей, претендующих на одну и ту же ячейку хэш-таблицы. Причем, если коллизии допускаются, то их количество необходимо минимизировать. В некоторых специальных случаях удается избежать коллизий вообще. Например, если все ключи элементов известны заранее (или очень редко меняются), то для них можно найти некоторую инъективную хэш-функцию, которая распределит их по ячейкам хэш-таблицы без коллизий [13]. Хэш-таблицы, использующие подобные хэш-функции, не нуждаются в механизме разрешения коллизий, и называются хэш-таблицами с прямой адресацией. Хэш-таблицы должны соответствовать следующим свойствам:

Выполнение операции в хэш-таблице начинается с вычисления хэш- функции от ключа. Получающееся хэш-значение является индексом в исходном массиве [26].

Количество хранимых элементов массива, деленное на число возможных значений хэш-функции, называется коэффициентом заполнения хэш-таблицы ( load factor ) и является важным параметром, от которого зависит среднее время выполнения операций.

Операции поиска, вставки и удаления должны выполняться в среднем за время O. Однако при такой оценке не учитываются возможные аппаратные затраты на перестройку индекса хэш-таблицы, связанную с увеличением значения размера массива и добавлением в хэш- таблицу новой пары [29].

Механизм разрешения коллизий является важной составляющей любой хэш-таблицы.

Таким образом, особо актуальным является исследование разрешения коллизий, т.к. это и мировая проблема тоже, учитывая быстрое развитие как информационной безопасности, так и способов взлома информации.

# **Хэш-функция Меркла-Дамгарда**

Хэш-функции используются в качестве строительного блока во многих приложениях. В 2004 году серия атак показала наличие уязвимостей в широко распространенном алгоритме SHA-1. В связи с этим NIST обратился с рекомендацией перейти к использованию SHA-2 и в 2007 году объявил о конкурсе для нового стандарта хэширования SHA-3 [29].

В 1976 году Диффи и Хеллман впервые подчеркнули необходимость построения однонаправленной функции как составной части схемы цифровой подписи. Этот год можно считать отправной точкой развития хэш-функций.

Конструкция Меркла-Дамгарда была описана Ральфом Мерклом в его кандидатской диссертации в 1979 году. Суть конструкции заключается в итеративном процессе последовательных преобразований, когда на вход каждой итерации поступает блок исходного текста и выход предыдущей итерации. Входная строка x разбивается на t одинаковых по длине блоков, длина блока xi равна r. Длина блока r должна соответствовать длине входного блока функции сжатия f. [18] Рассмотрим подробнее шаги схемы (Рисунок 1).

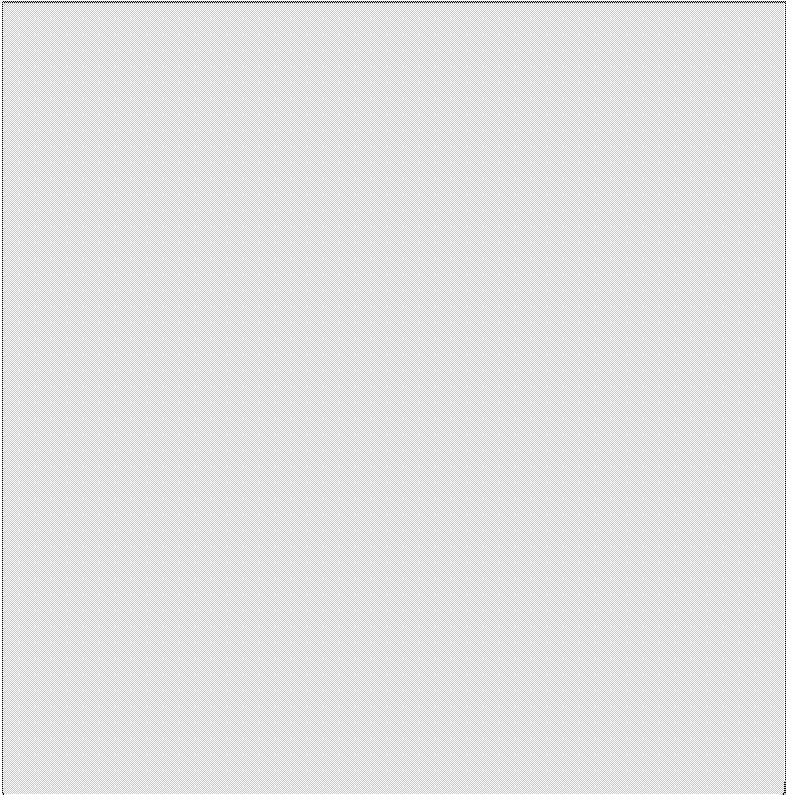
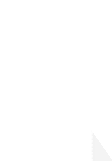
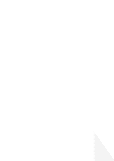
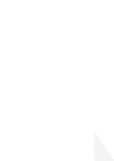
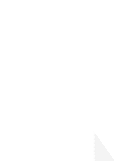
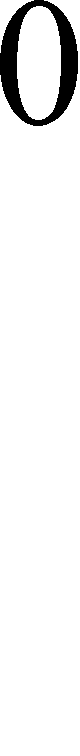
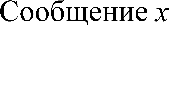


Рис. 1. Конструкция Меркла-Дамгарда

Разбить входные данные x на блоки x1,…,xt.

Дополнить последний блок xt нулевыми битами так, чтобы его длина равнялась r.

Добавить (t+1)-й блок, содержащий информацию о длине входных данных х.

Используя блок xi в качестве аргумента функции сжатия f, получить промежуточное значение *Hi*.

*Hi* обеспечивает обратную связь для *f* и вместе с блоком *xi*+1 используется в следующей итерации. Это предполагает наличие вектора инициализации *H*0, определенного предварительно.

После обработки всех блоков функция *g* отображает предварительное значение *Ht*+1 в окончательное значение хэш-суммы необходимой длины. Часто *g* – тождественная функция.

Роль функции сжатия может осуществлять любой блочный шифр *E*. Данная идея легла в основу развития конструкции Меркла-Дамгарда в схемах

Девиса-Мейера, Матиса-Мейера-Осеаса, Миагучи-Пренеля (Рисунок 2).

|  |  |  |
| --- | --- | --- |
|  |  |  |

Рис. 2. Схемы Девиса-Мейера, Матиса-Мейера-Осеаса, Миагучи-Пренеля

Поясним схемы, приведенные на рисунке 2. Блок *E* отождествляется с блочным алгоритмом шифрования с размером блока в *n* бит, *Ek*(*z*) – результат применения шифра *E* к блоку открытого текста *z* с использованием ключа *k*. Вход блока *E*,

помеченный символом «•», отвечает за ввод ключа. Все схемы предполагают наличие начального значения *H*0.

В схеме Девиса-Мейера блок сообщения *xi* и предыдущее значение хэш-функции *Hi*-1 поступают в качестве ключа и блока открытого текста со- ответственно на вход блочного шифра *E.* Получившийся в результате шифрования блок закрытого текста суммируется (операция XOR) с результатом предыдущей итерации хэширования (*Hi*-1). Результат суммирования *Hi* является результатом *i*-ой итерации хэш-функции на основе схемы Девиса- Мейера [19]:

*Hi*  *Ex* (*Hi*1 )  *Hi*1 .

*i*

В схеме Матиса-Мейера-Осеаса блок сообщения *xi* и предыдущее значение хэш-функции *Hi*-1 поступают в качестве блока открытого текста и ключа соответственно на вход блочного шифра *E.* Из-за возможных различий в размерах хэш-суммы и размере ключа шифра *E* значение *Hi*-1 подвергается предварительной обработке функцией *g*, реализующей отображение *n*-битного значения хэш- функции в *k*-битный ключ шифра *E*. В результате применения операции шифрования, получается блок закрытого текста, который суммируется с соответствующим ему блоком открытого текста (*xi*). Результат суммирования *Hi* является результатом *i*-ой итерации хэш-функции на основе схемы Матиса-Мейера-Осеаса: *Hi*  *Eg* ( *H i*1)(*xi* )  *xi* . [19]

Схема Миагучи-Пренеля является расширен ной версией схемы Матиса-Мейера-Осеаса. Отличие в том, что блок закрытого текста суммируется не только с соответствующим ему блоком открытого текста (*xi*), но и с результатом предыдущей итерации хэширования (*Hi*-1).

*Hi*  *Eg* ( *H* ) (*xi* )  *xi*  *Hi*1 .

*i*1

Конструкция Меркла-Дамгарда используется в таких известных хэш-функциях, как MD5, SHA-1, SHA-2. Широкую популярность схема приобрела благодаря доказательству того, что устойчивость к коллизиям всей конструкции зависит только от устойчивости к коллизиям функции сжатия. Несмотря на популярность схемы Меркла-Дамгарда, ряд работ показал недостатки данной конструкции, связанные с множественными коллизиями, дополнением сообщения до нужной длины, нахождением второго прообраза. Атаки направлены не на конкретный алгоритм, а применимы к любой хэш-функции, построенной по схеме, Меркла-Дамгарда [22]. Атака, основанная на множественных коллизиях, может быть осуществлена злоумышленником, даже если он имеет доступ к функции хэширования в целом, но не имеет возможности контролировать значения функции сжатия. Дополнение сообщения до нужной длины направлено на схему Девиса-Мейера, однако никак не затрагивает особенности самого блочного шифра, что делает эту атаку также достаточно общей [23]. В связи с этим стали появляться предложения по модификации столь популярной конструкции.

## **Конструкция Wide Pipe**

Если длина внутреннего состояния *Hi* совпадает с длиной хэш-суммы (*x*), то коллизия (*x*) = ( ) при входных данных *x* ≠ может быть обобщена на более длинные сообщения (*x*|| ) ≠ ( || ) добавлением одного и того же к обоим исходным сообщениям. Используя это свойство, в работе [3] автор показал, что для поиска 2*k*-коллизий требуется *k*·2*n*/2 операций, где *n* – длина внутреннего состояния (она же – длина хэш-суммы). Как меру борьбы с множественными коллизиями в 2004 г. Стефан Лакс предложил улучшенную версию схемы Меркла-Дамгарда. Данная конструкция очень похожа на свою предшественницу. Отличие в том, что длина внутреннего состояния *Hi* больше (например, в два раза), чем длина хэш-суммы. Функция осуществляет операции над блоком *H* +1, в результате которых его длина изменяется до нужной длины хэш-суммы. Таким образом, идея Лакса была крайне простой, но она сделала поиск множественных коллизий ресурсоемким [33].

В 2010 году предложена модификация Fast Wide Pipe, которая позволила увеличить скорость вычислений по сравнению с Wide Pipe в два раза. Каждое значение *Hi* делится на две половины. Первая половина подается на вход функции сжатия, а вторая – суммируется с результатом той же итерации. Однако данная схема требует дополнительной памяти.

В 2006 году предложен способ усложнения поиска коллизий хэш-функций методом рандомизации входных данных для функции сжатия. Такой способ позволил замаскировать коллизии в функции сжатия. Способ позиционируется авторами как отдельный режим работы криптосистемы хэширования без изменения самой ее конструкции. Может быть полезен в цифровых подписях для предотвращения сценария атаки нахождения второго прообраза.

# **Конструкция HAIFA**

Конструкция HAsh Iterative FrAmework (HAIFA), предложенная в 2007 году, имеет ряд конструктивных особенностей. Помимо нулей и ин- формации о длине входного сообщения, входные данные дополняются *r* битами, кодирующими длину хэш-суммы. В качестве аргументов функции сжатия, кроме внутреннего состояния и очередного блока сообщения, используются количество бит, уже поступавших на вход функции сжатия на предыдущих итерациях, и «соль». Использование количества бит, которые уже поступали на вход функции сжатия на предыдущих итерациях, в качестве аргумента функции сжатия – мера противодействия атаке с использованием неподвижных точек [21]. Добавление «соли» позволяет рассматривать данную конструкцию как экземпляр семейства рандомизированных хэш-функций и обеспечивает следующие преимущества:

**-** возможность оценить стойкость хэш-функции на теоретическом уровне;

- исключение атак, основанных на предварительных вычислениях;

**-** повышение стойкости цифровых подписей к атакам нахождения второго прообраза. Данная конструкция позволяет получать хэш-суммы различных длин в рамках одного приложения [27]. Используется универсальный вектор инициализации, предусмотренный разработчиком приложения, а вторичный вектор инициализации (длина которого соответствует желаемой длине хэш-суммы) получается как результат функции сжатия от универсального вектора и определенных значений остальных параметров.

# **Криптографическая губка**

Конструкция «криптографическая губка» была разработана группой криптографов во главе с Джоном Даеменом с целью заменить устаревшую конструкцию Меркла-Дамгарда. Впервые представлена в 2007 году на симпозиуме ECRYPT. Представляет собой отображение входных данных переменной длины в выходные данные также переменной длины. Преобразование (или перестановка) *f* оперирует с фиксированным количеством бит *b* = *r* + *c*, где *r* называется битовой скоростью, а *c* мощностью [26]. На начальном этапе, как и в конструкции Меркла-Дамгарда, входные данные рас- ширяются в соответствии с заданным алгоритмом, после чего разбиваются на блоки по *r* бит. Далее *b* бит состояния инициализируются нулями.

Конструкция включает две фазы (Рисунок 3).



Рис. 3. Криптографическая губка

В первой фазе (абсорбирование) *r*-битовые блоки сообщения суммируются (операция XOR) с первыми *r* битами внутреннего состояния – результата преобразования *f*. Когда эта операция проделана для всех блоков сообщения, фаза завершается. Далее, на фазе «отжима» первые *r* бит внутреннего состояния возвращаются в качестве выходных блоков *f* . Это действие повторяется, пока не будет получена желаемая длина хэш-суммы [9].

Внимательный взгляд на новую конструкцию позволяет заметить, что и она базируется на конструкции Меркла-Дамгарда, используя дополнительно операцию усечения.

# **1.2 Методы разрешения коллизий**

Коллизии осложняют использование хэш-таблиц, так как нарушают однозначность соответствия между хэш-кодами и данными. Тем не менее, существуют способы преодоления возникающих сложностей:

- метод цепочек (внешнее или открытое хэширование);

- метод открытой адресации (закрытое хэширование).

Метод цепочек. Технология сцепления элементов состоит в том, что элементы множества, которым соответствует одно и то же хэш-значение, связываются в цепочку-список [7]. В позиции номер i хранится указатель на голову списка тех элементов, у которых хэш-значение ключа равно i ; если таких элементов в множестве нет, в позиции i записан NULL. На [рис. 4.](https://intuit.ru/studies/courses/648/504/lecture/11467?page=1#image.38.1) демонстрируется реализация метода цепочек при разрешении коллизий. На ключ 002 претендуют два значения, которые организуются в линейный список.

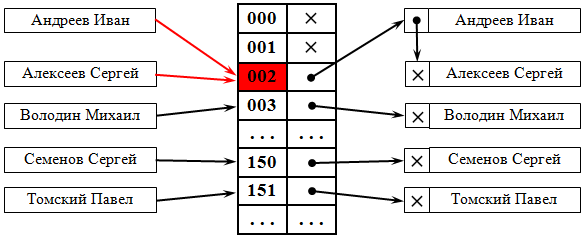


Рис. 4. Разрешение коллизий при помощи цепочек

Каждая ячейка массива является указателем на связный список (цепочку) пар ключ-значение, соответствующих одному и тому же хэш-значению ключа. Коллизии просто приводят к тому, что появляются цепочки длиной более одного элемента [9].

Операции поиска или удаления данных требуют просмотра всех элементов соответствующей ему цепочки, чтобы найти в ней элемент с заданным ключом. Для добавления данных нужно добавить элемент в конец или начало соответствующего списка, и, в случае если коэффициент заполнения станет слишком велик, увеличить размер массива и перестроить таблицу.

При предположении, что каждый элемент может попасть в любую позицию таблицы с равной вероятностью и независимо от того, куда попал любой другой элемент, среднее время работы операции поиска элемента составляет O(1+k), где k – коэффициент заполнения таблицы.

Метод открытой адресации. В отличие от хэширования с цепочками, при открытой адресации никаких списков нет, а все записи хранятся в самой хэш-таблице [1]. Каждая ячейка таблицы содержит либо элемент динамического множества, либо NULL.

В этом случае, если ячейка с вычисленным индексом занята, то можно просто просматривать следующие записи таблицы по порядку до тех пор, пока не будет найден ключ K или пустая позиция в таблице. Для вычисления шага можно также применить формулу, которая и определит способ изменения шага. На [рис. 5.](https://intuit.ru/studies/courses/648/504/lecture/11467?page=1#image.38.2)  разрешение коллизий осуществляется методом открытой адресации. Два значения претендуют на ключ 002, для одного из них находится первое свободное (еще незанятое) место в таблице.

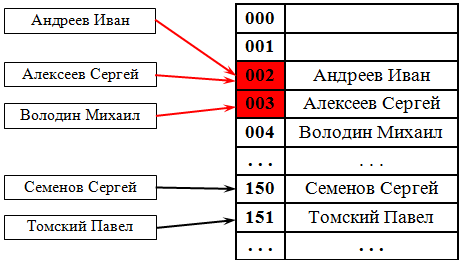


Рис. 5. Разрешение коллизий при помощи открытой адресации

При любом методе разрешения коллизий необходимо ограничить длину поиска элемента. Если для поиска элемента необходимо более 3 – 4 сравнений, то эффективность использования такой хэш-таблицы пропадает и ее следует реструктуризировать (т.е. найти другую хэш-функцию), чтобы минимизировать количество сравнений для поиска элемента [2].

Для успешной работы алгоритмов поиска, последовательность проб должна быть такой, чтобы все ячейки хэш-таблицы оказались просмотренными ровно по одному разу.

Удаление элементов в такой схеме несколько затруднено. Обычно поступают так: заводят логический флаг для каждой ячейки, помечающий, удален ли элемент в ней или нет. Тогда удаление элемента состоит в установке этого флага для соответствующей ячейки хэш-таблицы, но при этом необходимо модифицировать процедуру поиска существующего элемента так, чтобы она считала удаленные ячейки занятыми, а процедуру добавления – чтобы она их считала свободными и сбрасывала значение флага при добавлении [31].

# **1.2.1 Алгоритмы хэширования**

Существует несколько типов функций хэширования, каждая из которых имеет свои преимущества и недостатки и основана на представлении данных. Приведем обзор и анализ некоторых наиболее простых из применяемых на практике хэш-функций.

# **1.2.2 Таблица прямого доступа**

Простейшей организацией таблицы, обеспечивающей идеально быстрый поиск, является таблица прямого доступа. В такой таблице ключ является адресом записи в таблице или может быть преобразован в адрес, причем таким образом, что никакие два разных ключа не преобразуются в один и тот же адрес. При создании таблицы выделяется память для хранения всей таблицы и заполняется пустыми записями. Затем записи вносятся в таблицу – каждая на свое место, определяемое ее ключом. При поиске ключ используется как адрес и по этому адресу выбирается запись. Если выбранная запись пустая, то записи с таким ключом вообще нет в таблице. Таблицы прямого доступа очень эффективны в использовании, но, к сожалению, область их применения весьма ограничена [11].

Назовем пространством ключей множество всех теоретически возможных значений ключей записи. Назовем пространством записей множество тех ячеек памяти, которые выделяются для хранения таблицы. Таблицы прямого доступа применимы только для таких задач, в которых размер пространства записей может быть равен размеру пространства ключей. В большинстве реальных задач размер пространства записей много меньше, чем пространства ключей. Так, если в качестве ключа используется фамилия, то, даже ограничив длину ключа десятью символами кириллицы, получаем 3310 возможных значений ключей. Даже если ресурсы вычислительной системы и позволят выделить пространство записей такого размера, то значительная часть этого пространства будет заполнена пустыми записями, так как в каждом конкретном заполнении таблицы фактическое множество ключей не будет полностью покрывать пространство ключей [19].

В целях экономии памяти можно назначать размер пространства записей равным размеру фактического множества записей или превосходящим его незначительно. В этом случае необходимо иметь некоторую функцию, обеспечивающую отображение точки из пространства ключей в точку в пространстве записей, то есть, преобразование ключа в адрес записи: a=h(k), где a – адрес, k – ключ [19].

Идеальной хэш-функцией является инъективная функция, которая для любых двух неодинаковых ключей дает неодинаковые адреса.

# **1.2.3 Метод остатков от деления**

Простейшей хэш-функцией является деление по модулю числового значения ключа Key на размер пространства записи HashTableSize. Результат интерпретируется как адрес записи. Следует иметь в виду, что такая функция хорошо соответствует первому, но плохо – последним трем требованиям к хэш-функции и сама по себе может быть применена лишь в очень ограниченном диапазоне реальных задач. Однако операция деления по модулю обычно применяется как последний шаг в более сложных функциях хэширования, обеспечивая приведение результата к размеру пространства записей [14].

Если ключей меньше, чем элементов массива, то в качестве хэш-функции можно использовать деление по модулю, то есть остаток от деления целочисленного ключа Key на размерность массива HashTableSize, то есть:

Key % HashTableSize

Данная функция очень проста, хотя и не относится к хорошим. Вообще, можно использовать любую размерность массива, но она должна быть такой, чтобы минимизировать число коллизий. Для этого в качестве размерности лучше использовать простое число. В большинстве случаев подобный выбор вполне удовлетворителен [6]. Для символьной строки ключом может являться остаток от деления, например, суммы кодов символов строки на HashTableSize.

На практике, метод деления – самый распространенный.

//функция создания хэш-таблицы метод деления по модулю

int Hash(int Key, int HashTableSize) {

//HashTableSize

return Key % HashTableSize;

}

# **1.2.4 Метод функции середины квадрата**

Следующей хэш-функцией является функция середины квадрата. Значение ключа преобразуется в число, это число затем возводится в квадрат, из него выбираются несколько средних цифр и интерпретируются как адрес записи.

# **1.2.6 Метод свертки**

Еще одной хэш-функцией можно назвать функцию свертки. Цифровое представление ключа разбивается на части, каждая из которых имеет длину, равную длине требуемого адреса. Над частями производятся определенные арифметические или поразрядные логические операции, результат которых интерпретируется как адрес. Например, для сравнительно небольших таблиц с ключами – символьными строками неплохие результаты дает функция хэширования, в которой адрес записи получается в результате сложения кодов символов, составляющих строку-ключ [10].

В качестве хэш-функции также применяют функцию преобразования системы счисления. Ключ, записанный как число в некоторой системе счисления P, интерпретируется как число в системе счисления Q>P. Обычно выбирают Q=P+1. Это число переводится из системы Q обратно в систему P, приводится к размеру пространства записей и интерпретируется как адрес.

# **1.2.5 Открытое хэширование**

Основная идея базовой структуры при открытом (внешнем) хэшировании заключается в том, что потенциальное множество (возможно, бесконечное) разбивается на конечное число классов. Для В классов, пронумерованных от 0 до В-1, строится хэш-функция h(x) такая, что для любого элемента х исходного множества функция h(x) принимает целочисленное значение из интервала 0,1,...,В-1, соответствующее классу, которому принадлежит элемент х. Часто классы называют сегментами, поэтому будем говорить, что элемент х принадлежит сегменту h(x). Массив, называемый таблицей сегментов и проиндексированный номерами сегментов 0,1,...,В-1, содержит заголовки для B списков. Элемент х, относящийся к i -му списку – это элемент исходного множества, для которого h(x)=i. [10]

Если сегменты примерно одинаковы по размеру, то в этом случае списки всех сегментов должны быть наиболее короткими при данном числе сегментов. Если исходное множество состоит из N элементов, тогда средняя длина списков будет N/B элементов. Если можно оценить величину N и выбрать В как можно ближе к этой величине, то в каждом списке будет один или два элемента. Тогда время выполнения операторов словарей будет малой постоянной величиной, не зависящей от N.

Пример 1. Программная реализация открытого хэширования.

#include «stdafx.h»

#include <iostream>

#include <fstream>

using namespace std;

typedef int T; // тип элементов

typedef int hashTableIndex; // индекс в хэш-таблице

#define compEQ(a,b) (a == b)

typedef struct Node\_ {

T data;// данные, хранящиеся в вершине

struct Node\_ \*next; // следующая вершина

} Node;

Node \*\*hashTable;

int hashTableSize;

hashTableIndex myhash(T data);

Node \*insertNode(T data);

void deleteNode(T data);

Node \*findNode (T data);

int \_tmain(int argc, \_TCHAR\* argv[]){

int i, \*a, maxnum;

cout << «Введите количество элементов maxnum : «;

cin >> maxnum;

cout << «Введите размер хэш-таблицы HashTableSize : «;

cin >> hashTableSize;

a = new int[maxnum];

hashTable = new Node\*[hashTableSize];

for (i = 0; i < hashTableSize; i++)

hashTable[i] = NULL;

// генерация массива

for (i = 0; i < maxnum; i++)

a[i] = rand();

// заполнение хэш-таблицы элементами массива

for (i = 0; i < maxnum; i++) {

insertNode(a[i]);

}

// поиск элементов массива по хэш-таблице

for (i = maxnum-1; i >= 0; i--) {

findNode(a[i]);

}

// вывод элементов массива в файл List.txt

ofstream out(«List.txt»);

for (i = 0; i < maxnum; i++){

out << a[i];

if ( i < maxnum - 1 ) out << «\t»;

}

out.close();

// сохранение хэш-таблицы в файл HashTable.txt

out.open(«HashTable.txt»);

for (i = 0; i < hashTableSize; i++){

out << i << « : «;

Node \*Temp = hashTable[i];

while ( Temp ){

out << Temp->data << « -> «;

Temp = Temp->next;

}

out << endl;

}

out.close();

// очистка хэш-таблицы

for (i = maxnum-1; i >= 0; i--) {

deleteNode(a[i]);

}

system(«pause»);

return 0;

}

// хэш-функция размещения вершины

hashTableIndex myhash(T data) {

return (data % hashTableSize);

}

// функция поиска местоположения и вставки вершины в таблицу

Node \*insertNode(T data) {

Node \*p, \*p0;

hashTableIndex bucket;

// вставка вершины в начало списка

bucket = myhash(data);

if ((p = new Node) == 0) {

fprintf (stderr, «Нехватка памяти (insertNode)\n»);

exit(1);

}

p0 = hashTable[bucket];

hashTable[bucket] = p;

p->next = p0;

p->data = data;

return p;

}

//функция удаления вершины из таблицы

void deleteNode(T data) {

Node \*p0, \*p;

hashTableIndex bucket;

p0 = 0;

bucket = myhash(data);

p = hashTable[bucket];

while (p && !compEQ(p->data, data)) {

p0 = p;

p = p->next;

}

if (!p) return;

if (p0)

p0->next = p->next;

else

hashTable[bucket] = p->next;

free (p);

}

// функция поиска вершины со значением data

Node \*findNode (T data) {

Node \*p;

p = hashTable[myhash(data)];

while (p && !compEQ(p->data, data))

p = p->next;

return p;

}

Листинг .

# **1.2.7 Закрытое хэширование**

При закрытом (внутреннем) хэшировании в хэш-таблице хранятся непосредственно сами элементы, а не заголовки списков элементов, и пустые, пока не занятые ничем сегменты. Поэтому в каждой записи может храниться не более одного элемента. При закрытом хэшировании применяется методика повторного хэширования.

При поиске элемента Х необходимо просмотреть все местоположения, пока не будет найден элемент Х или пока не встретится пустой сегмент [14].

Если осуществляется попытка добавления элемента в класс, в котором уже имеется элемент - возникает коллизия, то происходит повторное хэширование относительно нового количества элементов так, чтобы перераспределить номера элементов. Каждое местоположение последовательно проверяется, пока для добавляемого элемента не найдется свободное. Если свободных мест не окажется, это будет означать, что таблица заполнена, соответственно, элемент Х добавить нельзя.

Существует три метода повторного хэширования, помогающих определить местоположение добавляемых элементов, чьи хэш-значения совпали с одним из элементов в таблице [14].

   1. Линейное опробование сводится к последовательному перебору сегментов таблицы с некоторым фиксированным шагом: A=H(X)+Ci, где i – номер попытки разрешить коллизию, C – константа, определяющая шаг перебора.

Если шаг равен единице, происходит последовательный перебор всех сегментов после текущего.

  2. Квадратичное опробование отличается формулой и зависит от попытки разрешить коллизию нелинейно. A=H(X)+Ci +Di2, где i – номер попытки разрешить коллизию, C и D – константы. Благодаря нелинейности квадратичного опробования уменьшается число проб при большом числе похожих значений хэш-функций. Но чем больше проб, тем больше вероятность того (растет в геометрической прогрессии), что получившееся значение адреса будет вне табличного пространства [17].

3. Двойное хэширование заключается в суммировании значений двух хэш-функций, работающих с одним и тем же элементом: A=H1(X)+H2(X) По мере заполнения хещ-таблицы могут возникнуть коллизии. Вследствие этого численное значение адреса может быть за пределами допустимых адресов. Эту проблему можно решить увеличением длины таблицы. Однако это хоть и сократит число коллизий и ускорит работу с хэш-таблицей, но приведет к нерациональному расходованию памяти [17].

Таким образом, хэширование, которое родилось еще в середине прошлого века, активно используется в наши дни везде, где требуется произвести быструю выборку данных. Появились новые методы хэширования, новые модификации алгоритмов, написанных ранее. Линейное хэширование, которое не имеет ничего общего с классической технологией линейной адресации, позволяет многим хэш-адресам расти и/или выступать в поли вставляемых и удаляемых элементов. Линейное хэширование может также использоваться для огромных баз данных, распределенных между разными узлами в сети. Разумеется, методы и сферы применения хэширования не ограничиваются тем, что представлено в этой работе. Не вдаваясь в строгий анализ эффективности, были рассмотрены только базовые, наиболее известные методы.

# **ГЛАВА 2. АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ ДЛЯ ПОСТРОЕНИЯ АТАК НА ХЭШ ФУНКЦИИ С КОНТРОЛЬНЫМИ СУММАМИ**

В последние годы отрасль криптографии, связанная с хэшированием, столкнулась с существенной проблемой – обеспечением стойкости к мультиколлизям, которые используются атакой Жукса. Зарождение этой проблемы началось с атаки «Дня рождения», ответом на которую стало увеличение в два раза длины результирующего хэш-значения. Такой выход оказался неадекватным имеющимся вычислительным ресурсам общества и ограничениям во времени выполнения хэширования. Для уменьшения длительности хэширования предложено каскадирование, описанное в работе, то есть распараллеливание вычислений хэш-значений малой разрядности, которые вычисляются, используя разные хэш-функции, и их конкатенация после завершения последней итерации. Каскадирование считалось эффективным средством противодействия атаке «Дня рождения» до 2004 года, когда Жукс опубликовал работу [29]. Находя, так называемые, мультиколлизии в одной из хэш-функций, вычисляющихся параллельно, Жукс показал, что стойкость хэш- значения, полученного путем каскадирования этих функций, ненамного больше стойкости одной из них (при условии одинаковой стойкости функций). Таким образом, проблема, связанная со стойкостью и продолжительностью хэширования, вернулась.

Подходы, предложенные Жуксом в его атаке, были обобщены для любого количества хэш-функций и усовершенствованы для более быстрого нахождения мультиколлизий в работах. Совершены ряд попыток предложить математические модели хэш-функций, стойких к таким атакам, которые согласно традиции, сложившейся в западной литературе, принято называть конструкциями. Наиболее известными являются конструкция HAIFA и конструкция удвоенного канала Люкса [24]. Первая использует ряд специфических аргументов, но теоретически не доказывает увеличения стойкости данной конструкции к атаке Жукса, а вторая решает задачу путем двукратного увеличения вычислений и длин промежуточных хэш-значений.

# **2.1 Анализ методов построения одиночной коллизии**

**Метод «грубой силы»**

Метод грубой силы подразумевает полный перебор всех возможных значений хэш функции. Например, пусть хэш функция генерирует 128 бит, тогда для поиска коллизии нам потребуется перебрать 2128 вариантов хэш функций, к тому же нам потребуется еще больше памяти, чтобы хранить все эти значения. Хорошо, мы можем улучшить алгоритм, воспользовавшись парадоксом дней рождений, т.е. взяв 264 значений, мы с вероятностью более 0.5 получим совпадение, тем не менее, даже распаллаллелив вычисления, нам все равно придется хранить более 264 байт данных. Далее будут рассмотрены более оптимальные методы поиска одиночных коллизий.

**Алгоритм Флойда**

Алгоритм Флойда предназначен для поиска цикла в последовательности значений итеративной функции. Алгоритм также носит название «Заяц и черепаха».

927 байт

Надо найти начало петли. Для решения поставленной задачи, пустим 2 итератора по списку, один будет двигаться в два раза быстрее (будет делать на каждый такт по два шага). Когда заяц (быстрый итератор) догонит черепаху (медленный итератор), мы знаем, что:

* Черепаха либо дошла до петли, либо уже по ней движется. Пройти петлю целиком она не могла
* Заяц пробежал по петле как минимум один раз
* Заяц пробежал ровно в два раза больше

Теперь ставим черепаху в начало списка и пускаем два итератора с одинаковой скоростью, пока они не пересекутся.

Причем же здесь хэш функция? Пусть наш список будет выглядеть так:

X0 – произвольное число

Xi+1 = hash(xi)

Для медленного итератора на каждой итерации вычисляем хэш значение от предыдущего значения один раз, для быстрого вычисляем hash(hash(x)). Найденная петля и будет наша коллизия. Плюс данного алгоритма в том, что он требует O(1) затрат памяти.

**def** floyd(f, x0):

*# Основная часть алгоритма: находим повторение x\_i = x\_2i.*

*# Заяц движется вдвое быстрее черепахи,*

*# и расстояние между ними увеличивается на единицу от шага к шагу.*

*# Однажды они окажутся внутри цикла, и тогда расстояние между ними*

*# будет делиться на λ.*

tortoise = f(x0) *# f(x0) является элементом, следующим за x0.*

hare = f(f(x0))

**while** tortoise != hare:

tortoise = f(tortoise)

hare = f(f(hare))

*# В этот момент позиция черепахи ν,*

*# которая равна расстоянию между черепахой и зайцем,*

*# делится на период λ. Таким образом, заяц, двигаясь*

*# по кольцу на одну позицию за один раз,*

*# и черепаха, опять начавшая движение со стартовой точки x0 и*

*# приближающаяся к кольцу, встретятся в начале кольца*

*# Находим позицию μ встречи.*

mu = 0

tortoise = x0

**while** tortoise != hare:

tortoise = f(tortoise)

hare = f(hare) *# Заяц и черепаха двигаются с одинаковой скоростью*

mu += 1

*# Находим длину кратчайшего цикла, начинающегося с позиции x\_μ*

*# Заяц движется на одну позицию вперёд,*

*# в то время как черепаха стоит на месте.*

lam = 1

hare = f(tortoise)

**while** tortoise != hare:

hare = f(hare)

lam += 1

**return** lam, mu

Листинг 2.1 Алгоритм «Флойда»

### **Алгоритм Брента**

Ричард Брент описал альтернативный алгоритм нахождения цикла, которому, подобно алгоритму черепахи и зайца, требуется лишь два указателя на последовательность. Однако он основан на другом принципе — поиске наименьшей степени 2i числа 2. По утверждению Брента, алгоритм работает на 36 % быстрее алгоритма Флойда.

*# Основная фаза: ищем степень двойки*

power = lam = 1

tortoise = x0

hare = f(x0) *# f(x0) — элемент/узел, следующий за x0.*

**while** tortoise != hare:

**if** power == lam: *# время начать новую степень двойки?*

tortoise = hare

power \*= 2

lam = 0

hare = f(hare)

lam += 1

*# Находим позицию первого повторения длины λ*

mu = 0

tortoise = hare = x0

**for** i **in** range(lam):

*# range(lam) образует список со значениями 0, 1, ... , lam-1*

hare = f(hare)

*# расстояние между черепахой и зайцем теперь равно λ.*

*# Теперь черепаха и заяц движутся с одинаковой скоростью, пока не встретятся*

**while** tortoise != hare:

tortoise = f(tortoise)

hare = f(hare)

mu += 1

**return** lam, mu

Листинг 2.2 Алгоритм «Брента»

**Лямбда-алгоритм Полларда**

Если у нас есть функция с коллизиями, и мы начнём её итерировать (т.е. как в прошлый раз: Xi+1 = h(Xi)) от двух разных случайных значений, скорее всего, цепочки пересекутся, после чего сольются.

942 байт

На этой идее основан следующий алгоритм:

Берём случайное x0, итерируем от него функцию Q раз, получаем xQ.

Берём случайное y0, и итерируем от него до тех пор, пока не получим то же самое xQ.

Теперь, зная количество шагов, которые пришлось пройти по первому и по второму пути, можно пустить по ним двух черепах, дав одной из них такую фору, чтобы они встретились.

**Параллельный лямбда-алгоритм Полларда**

Можно распараллелить предыдущий алгоритм по следующему принципу:

* Пусть у нас есть T компьютеров.
* Среди множества значений функции выберем «волшебные». Выберем так, чтобы до попадания в такое значение нужно было бы сделать в среднем попыток. Например, посчитаем X = и скажем, что первые X бит должны быть нулями.
* Сгенерируем T случайных чисел, раздадим их компьютерам.
* Каждый компьютер начинает итерировать функцию со своего значения и продолжает до тех пор, пока не получат «волшебное» значение. Получив, присылает в центр результат, своё начальное значение и количество шагов.
* Центр сравнивает результаты, у кого-нибудь да совпадёт. Из двух начальных значений и двух длин путей несложно найти коллизию.

Конечно, может ни у кого не совпасть. Но, по парадоксу дней рождений, совпадёт с вероятностью больше 0,5, нам этого достаточно**.**

# **2.1 Анализ методов построения мультиколлизий**

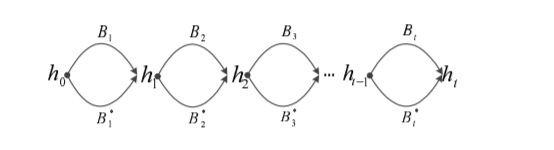
Впервые атака с использованием мультиколлизии предложена в работе Жуксом. Данная атака была предложена для хэш-функций, использующих каскадирование, и основанных на усиленной конструкции Меркла-Дамгарда. Согласно этой конструкции, информационное сообщение ***M*** дополняется до длины, кратной длине блока данных, разбивается на *l* частей и дополняется блоком *ml* 1, который содержит длину оригинального сообщения ***M***, а хэширование происходит согласно формуле:

*hi* *f* *mi* , *hi*1  , (1)

где *hi* – промежуточное хэш-значение, полученное на *i* -ом шаге;

*f*  – функция сжатия, обеспечивающая фиксированную длину результата.

Для идеальной хэш-функции с n-битным выходным значением трудоемкость поиска пары, дающей коллизию, составляет примерно 2n/2, а для поиска k-коллизии (мультиколлизии) – 2n(k-1)/k. Однако, основываясь на попарных коллизиях, Жуа на Crypto’04 [12] предложил метод конструирования 2t-коллизий для итеративной структуры с трудоемкостью всего t x 2n/2. Как показано на рис. 7, атакующий сначала генерирует t различных попарных коллизий {(B1, B1\*), (B2, B2\*), …, (Bt, Bt\*)}. Затем атакующий может сразу получить 2t-коллизию вида (b1, b2,…, bt), где bi – один из двух блоков Bi и Bi\*. Жукс предложил следующий алгоритм:



1. Пусть h0 будет равно вектору инициализации
2. Для i от 1 до t выполняем

* Ищем такие Bi и Bi\*, для которых gN(hi-1, Bi) = gN(hi-1, Bi\*)
* Составление результирующих 2*t* сообщений в форме (*b*1,…, *bt*), где *bi* – это один из двух блоков: *Bi* или *Bi*\*.

Некоторое время в качестве одного из метода противодействия мультиколлизиям рассматривалось дополнение исходного сообщения различного рода контрольными суммами. Например, в рассматриваемой хэш функции все блоки складываются друг с другом по модулю 264. Однако в работе [x] были предложены методы построения мультиколлизий для таких хэш функций.

Например, в работе [] предложен следующий метод построения мультиколлизий:

* Пусть завершающее преобразование выглядит следующим образом , где m1…mi – блоки исходного сообщения
* Определяем функцию , где n – количество выходных бит
* Последовательно строим коллизии для сообщений (), ()… (), так что
* Получаем 2*t* сообщений в форме (*m*1, *m*1,…, *mt,* , *m*t), где m*i* – это один из двух блоков: или \*.

**ОПИСАНИЕ ИССЛЕДУЕМОЙ ХЭШ ФУНКЦИИ**

Основу хэш функции «Мора» разбиение исходного сообщения на блоки и применение к ним функции сжатия. Функция сжатия вычисляется по формуле:

Где

**Xor** – операция покомпонентного сложения по модулю 2 двух двоичных

векторов одинаковой размерности

S - Нелинейное биективное преобразование множества двоичных векторов V4

задается подстановкой:

= V ec 04 Int4 : V4 ! V4; где 0 : Z24 ! Z24

L – линейное преобразование

P – перестановка полубайт

E(K;m) = X[K9]LPSX[K8]:::LPSX[K1](m):

ЗначенияKi 2 V64; i = 1; :::; 9 вычисляются следующим образом:

K1 = K;

Ki = LPS(Ki 1 Ci 1); i = 2; :::; 9:

Хэш функция имеет следующие параметры:

Длина блока данных – 64 бит

Матрица линейного преобразования имеет вид

L =

{

0x3a22, 0x8511, 0x4b99, 0x2cdd,

0x483b, 0x248c, 0x1246, 0x9123,

0x59e5, 0xbd7b, 0xcfac, 0x6e56,

0xac52, 0x56b1, 0xb3c9, 0xc86d

};

Итерационные константы

C =

{

0xc0164633575a9699,

0x925b4ef49a5e7174,

0x86a89cdcf673be26,

0x1885558f0eaca3f1,

0xdcfc5b89e35e8439,

0x54b9edc789464d23,

0xf80d49afde044bf9,

0x8cbbdf71ccaa43f1,

0xcb43af722cb520b9

};

Нелинейное преобразование

SBox =

{

15, 9, 1, 7, 13, 12, 2, 8, 6, 5, 14, 3, 0, 11, 4, 10

};

Перестановка полубайт.

Tau =

{

0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15

};

**ОПИСАНИЕ ПРОГРАММНОЙ РЕАЛИЗАЦИИ ЭКСПЕРИМЕНТА**

Для проведения экспериментальной верификации описанных теоретических результатов согласно общей схемы атаки (см. \*глава с описанием атаки) было разработано соответствующее программное обеспечение, состоящее из:

* программной реализации алгоритма хэширования «Мора»;
* программной реализации алгоритма, реализующего построение мультиколлизий для данной хэш функции согласно общей схеме.

Программная реализация алгоритмов написана на языке C# с использование .Net Framework версии 4.7.2. Исходный код программного обеспечения приведен в Приложении А.

**Хэш «Мора»**

**Constants** класс содержит значения постоянных переменных, а именно:

* L – матрица, используемая для умножения в линейном преобразовании множества двоичных векторов.
* C – итерационные константы
* SBox - нелинейное биективное преобразование множества двоичных векторов
* Tau – перестановка полубайт

**ByteUtils** предоставляет методы для работы с массивами байт:

* Xor – производит операцию исключающего или для 2х массивов
* RingSum – операция сложения 2ух элементов в поле GF(24)
* JoinBytes – объединяет 2 4х битных значение в одно 8ми битное
* SplitByte – разбивает 8ми битное значение на два 4 битных (значение дополняется до байта лидирующими нулями)

**HashFunction** реализация хэш функции «Мора»

* GetHash - функция, которая непосредственно производит хэширование
* G\_n – функция сжатия
* L – функция линейного преобразования
* P – функция перестановки полубайт
* S – функция нелинейного преобразования
* KeySchedule – вызов LPSX от итерационных констант
* E -

**ЗАКЛЮЧЕНИЕ**

В настоящей работе рассмотрена схема построение атаки на основе 2k мультиколлизий.

**СПИСОК ЛИТЕРАТУРЫ**

[1] Paul C. van Oorschot and Michael J. Wiener Parallel Collision Search with Cryptanalytic Applications 1996 September 23, <http://cr.yp.to/bib/1999/vanoorschot.pdf>

[2] Бондакова О.С. Описание низкоресурсной хэш функции «Мора», г.Москва. 2020

[3] Antonie Joux Multicollisions in iterated hash functions. Application to cascade construction, DSSCI Crypto Lab, France <https://www.iacr.org/archive/crypto2004/31520306/multicollisions.pdf>

[4] Praveen Gauravaram and John Kelsey Linear-XOR and Additive Checksums Don’t Protect Damgard-Merkle Hashes from Generic Attacks

[5]M.Nandi and D.R.Stinson Multicollision Attacks on Some Generalized Sequential Hash Functions, Canada, 2006 https://www.researchgate.net/publication/3086233\_Multicollision\_Attacks\_on\_Some\_Generalized\_Sequential\_Hash\_Functions