УДК 004.021

Гапанович Д.А., Чубариков В.Н.

Московский государственный университет имени М.В. Ломоносова, г. Москва, Россия

ХЭШ-АЛГОРИТМ С УПРАВЛЯЮЩЕЙ ДРЕВОВИДНОЙ СТРУКТУРОЙ И МЕТОД ЕГО РЕАЛИЗАЦИИ НА ПАРАЛЛЕЛЬНЫХ АРХИТЕКТУРАХ

Аннотация

Статья посвящена исследованию и разработке новых методов хэширования, обладающих возможностью адаптации к повышенным требованиям криптостойкости, а также поддерживающих распараллеливание вычислений, что особенно важно для ускорения вычисления хэш-значений больших и сверх больших файлов, и отражает тенденцию все более широкого распространения вычислительных архитектур с высокой степенью параллелизма. В работе предложен оригинальный метод построения хэш-алгоритма на основе древовидных управляющих структур, а также метод реализации предложенного хэш-алгоритма на параллельных архитектурах по технологии MapReduce.

Ключевые слова

Хэширование; хэш-функции; хэш-алгоритмы; криптографические хэш-алгоритмы; двоичные деревья; управляющие древовидные структуры; схема вычислений MapReduce; параллельные вычисления.

Gapanovich D.A., Chubarikov V.N.

Lomonosov Moscow State University, Moscow, Russia

HASH ALGORITHM WITH THE CONTROLLING TREE-LIKE STRUCTURE AND THE METHOD OF ITS IMPLEMENTATION ON PARALLEL ARCHITECTURES

Abstract

The article is devoted to research and development of new hashing methods that have the ability to adapt to increased requirements for cryptographic stability, and also support parallelization of computations, which is especially important for accelerating the computation of hash values of large and super large files and reflects the trend of increasingly widespreading computational architectures with a high degree of parallelisation . An original method for constructing a hash algorithm based on tree-like control structures is proposed, as well as a method for implementing the proposed hash algorithm on parallel architectures using MapReduce technology.

Keywords

Hashing; hash functions; hash algorithms; cryptographic hash algorithms; binary trees; control tree structures; MapReduce calculation scheme; parallel computations.

1. Введение

Хэширование является широко используемым в информатике и программировании алгоритмическим решением для оптимизации важнейших операций с данными, таких, как, например, поиск данных в больших информационных массивах, реорганизация информационных массивов, манипулирование с файлами в операционных системах, персонализация данных с целью обеспечение их целостности.

Область применения механизма хэширования чрезвычайно обширна — это реализация словарных систем, построение индексов в базах данных, построение таблиц имен в компиляторах, ускорение работы с файлами в операционных системах, организация списков ссылок в браузерах, организация словарей в программах переводчиках с одного языка на другой, конструирование электронных подписей электронных документов, реализация базовых операций в системах анализа больших данных и системах принятия решений и т.д.

Механизм хэширования был разработан в 50-х годах XX века, и одной из первых областей его применения стало решение «проблемы словаря». В

числе первопроходцев, опубликовавших свои методы хэширования в открытой печати, называются [1, 2]: Ханс Петер Лун (нем. Hans Peter Luhn) — сотрудник фирмы IBM, в 1953 г. предложивший идею «хэш-кодирования»; Арнольд Думи (англ. Arnold Dumey), в своей пионерской работе «Computers and automation» первым описавший метод хэширования применительно к построению словарей в том виде, в каком он известен и в наше время, и предложивший также использовать в качестве «хэш-адреса» остаток от деления на простое число; Уэсли Питерсон (англ. W. Wesley Peterson), который в 1957 г. в журнале «IBM Journal of Research and Development» опубликовал статью о поиске текста в больших файлах, где исследовал возможности хэширования и предложил метод «открытой адресации», и т.д. Среди отечественных ученых, участвовавших в развитии методов хэширования на начальных этапах его становления, следует назвать академика А.П. Ершова, независимо разработавшего в 1957 г. метод открытой адресации [2].

Особенно интенсивное развитие методов хэширования осуществлялось в 80-90 гг. 20-го столетия в связи с масштабными разработками в таких областях системного программирования, как операционные системы, компиляторы языков программирования, базы данных и, конечно же, в области информационной безопасности.

В этот период Дональд Кнут [2] особенно выделял как важное практическое открытие хэш-метод, предложенный Witold Litwin и получивший название линейным хэшированием, а также ряд сложных методов, гарантирующих время выполнения операторов поиска и вставки данных (ключей) в информационные массивы со средним временем O(1) независимо от размера ключа.

Исследование и разработка новых методов хэширования с учетом быстрого развития области информационных технологий и, в том числе, новых вычислительных архитектур с высокой степенью параллелизма, представляет собой актуальную задачу.

В данной работе предложен оригинальный метод построения хэш-алгоритма на основе древовидных управляющих структур, а также метод реализации предложенного хэш-алгоритма на параллельных архитектурах по технологии MapReduce.

2. Хэш-функции и их свойства

Идея метода хэширования состоит в разбиении (потенциально неконечного) множества ключей A (символьных строк или, возможно, абстрактных типов данных, таких как, например, записи файлов) на конечное число B классов, пронумерованных от 0 до ($B-1$) с помощью **хэш-функции** $h(z)$, отображающей любой ключ (объект) z из A в целочисленное значение, принадлежащее отрезку $[0,\ldots,B-1]$. Число $h(z)$, называемое **хэш-значением** z, ассоциируется с номером "класса", и, самое главное, это число в контексте некоторым образом организованной модели данных может почти биективно (с точностью до коллизий) соответствовать исходному объекту и использоваться вместо этого объекта в операторах/операциях манипулирования данными, существенно ускоряя их исполнение. А при использовании в приложениях информационной безопасности играть роль дайджеста или «отпечатков пальцев» файла, сообщения или некоторого блока данных.

Однако эффективность работы хэш-механизма зависит прежде всего от того, насколько удачно построена для него хэш-функция. Таким образом, не каждая функция вида $f: A \to [0-(B-1)]$ применима в хэшировании.

Процитируем наиболее распространенные определения хэш-функций.

Определение 1 [1]**.** Хэш-функцией называется такое математическое или алгоритмическое преобразование заданного блока данных, которое обладает следующими свойствами:

1. хэш-функция имеет бесконечную область определения,
2. хэш-функция имеет конечную область значений,
3. она необратима,
4. изменение входного потока информации на один бит меняет около половины всех бит выходного потока, то есть результата хэш-функции.

Определение 2 [3]. Пусть задано некоторое входное сообщение *M*. Хэш-функцией (или функцией сгущения, или контрольной функцией) называется легко вычислимое необратимое числовое отображение $h(M)$, ставящее в соответствие сообщению *M* некоторое «короткое» сообщение $h(M)$. Другими словами, хэш-функцией называется любая функция y=h(x_1 x_2 … x_n), которая сообщению x=x_1x_2...x_n произвольной длины *n* ставит в соответствие целое число «*y*» фиксированной длины (*x_i* – текстовое представление *i*-го элемента сообщения вместо традиционного с нижним индексом).

Определение 3 [4]: Хэш-функцией называется односторонняя функция, предназначенная для получения дайджеста или «отпечатков пальцев» файла, сообщения или некоторого блока данных.

Хэш-код создается функцией *H*:

$$h = H(M),$$

где M является сообщением произвольной длины и h является хэш-кодом фиксированной длины.

Опыт использования данного механизма и его теоретические исследования позволили выявить наиболее важные и полезные свойства хэш-

функций. При этом свойства хэш-функций могут отражать и специфические требования со стороны приложений, в которых они используются.

Так, например, широкое применение механизм хэширования получил в криптографии и в приложениях информационной безопасности. Хэш-функции, применяемые в этой сфере, часто называют **криптографическими**, и им приписываются дополнительные свойства с учетом интересов области использования.

Для таких хэш-функций сформулированы следующие наборы требований с учетом требований к их криптостойкости, которые мы и рассмотрим.

В [1] определяются три основных требования, на которых основано большинство применений хэш-функций в криптографии:

1) Необратимость или стойкость к восстановлению прообраза: для заданного значения хэш-функции m не должен быть вычислен блок данных X, для которого $H(X)=m$.

2) Стойкость к коллизиям первого рода или восстановлению вторых прообразов: для заданного сообщения M должно быть вычислительно невозможно подобрать другое сообщение N, для которого $H(N)=H(M)$.

3) Стойкость к коллизиям второго рода: должно быть вычислительно невозможно подобрать пару сообщений (M, M'), имеющих одинаковый хэш.

Для криптографических хэш-функций также важно, чтобы при малых изменениях аргумента значение функции сильно изменялось (лавинный эффект), т.е. значение хэша не должно давать утечки информации даже об отдельных битах аргумента. Это требование необходимо для обеспечения криптостойкости алгоритмов хэширования пользовательских паролей для получения ключей.

В работе [3] свойства криптографических хэш-функций определены с помощью следующих требований:

1) Для любого заданного x из A значение функции $y=h(x)$ должно вычисляться достаточно «легко» и «быстро».

2) Для любого «x» практически невозможно найти «y» такое, что $y=h(x)$.

3) Для любого сообщения «x» практически невозможно найти «x`» такое, что $x`\neq x$ и $h(x`)=h(x)$.

4) Практически невозможно найти пару различных сообщений x и $x`$ таких, что $h(x`)=h(x)$.

В работе [4] требования к хэш-функциям сформулированы следующим образом.

1) Хэш-функция должна быть применима к любому блоку данных любой длины.

2) Хэш-функция создает выход фиксированной длины.

3) $H(M)$ – вычислимо за полиномиальное время относительно длины сообщения M.

4) для любого данного значения хэш-кода h вычислительно невозможно найти M такое, что $H(M)=h$.

5) Для любого данного сообщения x вычислительно невозможно найти $y\neq x$ такое, что $H(y)=H(x)$.

6) Вычислительно невозможно найти пару (x,y) такую, что $H(y)=H(x)$.

Остановимся на последнем наборе требований [4].

Первые три требования предписывают, чтобы хэш-функция достаточно легко (не более чем за полиномиальное время относительно длины сообщения) вычисляла хэш-код для сообщения любой длины.

Четвертое свойство означает, что хэш-функция должна обладать свойством односторонности, т.е. при том, что с помощью ее вычисления легко получить хэш-код для данного сообщения, решить обратную задачу - восстановить сообщение по хэш-коду в вычислительном плане было практически невозможно.

Данное свойство необходимо, если для аутентификации исходного сообщения M применяется хэш-значение, используемое вместе с секретным значением исходного сообщения.

Пятое свойство гарантирует невозможность найти или построить другое сообщение, значение хэш-функции которого совпадало бы со значением хэш-функции данного сообщения, что предотвращает подделку сообщения, когда в качестве аутентификатора используется защищенный от изменения хэш-код.

Хэш-функция, которая удовлетворяет первым пяти свойствам, называется **простой** или слабой хэш-функцией. Если кроме того выполняется шестое свойство, то такая функция называется **сильной** хэш-функцией.

Шестое свойство защищает против класса атак, известных как атака «день рождения» [4].

Атака «дней рождения» позволяет находить коллизии для хэш-функции с длиной значений n бит в среднем за примерно $2^{(n/2)}$ вычислений хэш-функции. Поэтому n-битная хэш-функция считается криптостойкой, если вычислительная сложность нахождения коллизий для неё близка к $2^{(n/2)}$.

(Парадокс «дней рождения» состоит в противоречии интуитивного восприятия человеком с результатом математического расчета вероятности того, что в группе из 23 или более человек, хотя бы у двух из них дни рождения (число и месяц) совпадут, превышает 50%, а для 60 человек – более 99%. Это может показаться противоречащим здравому смыслу, так как вероятность родиться человеку в определённый день года мала (1/365), а вероятность того, что двое родились в конкретный день — ещё меньше, но является верным в соответствии с теорией

вероятностей. Подробнее этот вопрос описан в работе по адресу – http://ru.wikipedia.org/wiki/Парадокс_дней_рождения).

Все рассмотренные выше наборы требований к хэш-функциям равносильны и приводят к теории односторонних функций [3].

3. Подход к построению хэш-функций

Существует большое разнообразие хэш-функций, и исследования в этой области продолжаются. Широкое использование при разработке хэш-функций получили итерационные методы. Типичная итеративная последовательная схема построения хэш-функции [5] описана ниже.

В качестве ядра итеративного алгоритма выбирается сжимающая функция f (или функция свертки), которая преобразует k входных бит в n выходных бит, где n — разрядность хэш-функции, а k — произвольное число, большее n. При этом сжимающая функция должна удовлетворять всем условиям криптостойкости, описанным выше для хэш-функций.

Далее входные данные рассматриваются как последовательность (поток) блоков фиксированной длины, равной $(k-n)$ бит. Итеративный алгоритм использует для хранения результатов итераций рабочую переменную размером в n бит. Этой переменной перед выполнением первой итерации в качестве начального значения присваивается некоторое общеизвестное число. На каждой итерации исполняется следующая последовательность действий:

- выбирается следующий за текущим блок данных,
- формируется конкатенация этого блока с выходным значением сжимающей функции на предыдущем шаге итерации,
- вычисляется n-битное значение хэш-функции текущей итерации.

Работа итерационного алгоритма иллюстрируется на рис. 1.

Заметим, что описанная выше схема вычисления хэш-значения позволяет реализовать упомянутый выше лавинный эффект, т.е. обеспечить зависимость каждого бита значения хэш-функции от всего входного потока данных и, естественно, начального значения итерационного алгоритма.

Заметим также, что при реализации хэш-функций на основе итеративного алгоритма предполагается, что размер входного потока данных должен быть кратен $(k-n)$. Естественно, что данное ограничение легко снимается посредством расширения исходного потока данных необходимым числом бит некоторым заранее известным способом.

Описанная выше итерационная схема в литературе называется также конструкцией Меркла-Дамгарда. Она используется в таких известных хэш-функциях, как MD5, SHA-1, SHA-2. Широкую популярность схема приобрела после того, как было доказано, что устойчивость к коллизиям всей конструкции зависит только от устойчивости к коллизиям функции сжатия (или функция свертки).

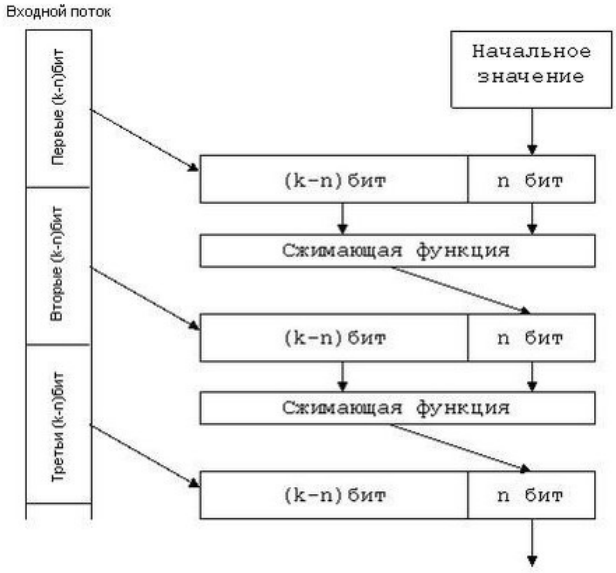

Рис.1. Схема работы итерационного алгоритма

4. Проектирование хэш-функции специального вида

В разделе 4 была рассмотрена типовая итерационная схема построения хэш-функций, называемая также конструкцией Меркла-Дамгарда, которая получила широкое использование на практике и, в частности, при разработке таких известных хэш-функциях, как MD5, SHA-1, SHA-2.

Впоследствии было показано, что данная конструкция обладает некоторыми недостатками, связанными с множественными коллизиями, наличием остаточной статистической зависимости в данных, с уязвимостью для атак в силу предсказуемости структуры алгоритма, которые частично переносятся на все подобные схемы.

Еще одним недостатком хэш-решений, построенных на основе известных итерационной схем, является линейный характер вычисления.

Учитывая, что основной тенденцией развития вычислительных средств является создание вычислительных систем с высокой степенью параллелизма, включая систем с массовым параллелизмом, интересным представляется поиск моделей вычисления хэш-функций, характеризующихся возможностью распараллеливания вычислений, а также возможностью параметризации самой управляющей схемы организации вычислений.

В поиске такого решения будем отталкиваться от вычислительной модели схемы Меркла-

Дамгарда, которую можно описать следующим образом.

На вход хэш-функции подаются данные, которые разбиваются на блоки одинаковой длины. В некоторых случаях при построении хэш-функций на основе рассматриваемой схемы, используется предварительная перестановка блоков для уменьшения статистической зависимости в исходных данных, которая биективно отображает исходную последовательность бит данных в другую последовательность.

Затем итеративно для каждого i-го блока данных ($\underline{x}\_i$) и хэш-значения $h\_\{i-1\}$ предыдущего блока (или инициализирующего хэш-значения) применяются функции свёртки, как правило, односторонние функции. Такие преобразования можно обозначить функцией $h\_\{i+1\} = f(h\_i, \underline{x}\_i)$, где h_i – результат предыдущей итерации (или инициализирующее значение на первом шаге итерации), $\underline{x}\_i$ – следующий блок данных, $h\_\{i+1\}$ – результат данной итерации.

Графически это можно изобразить как на рис.2.

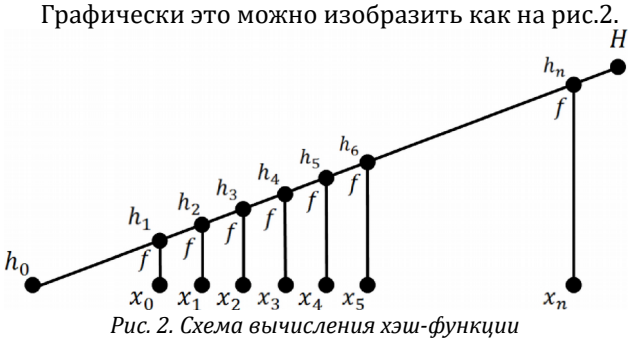

Рис. 2. Схема вычисления хэш-функции

Т.е. структуру алгоритма можно представить с помощью вырожденного дерева, определяющего порядок исполнения функциональных элементов дерева при его обходе снизу вверх слева направо.

Тогда в обобщённом виде схема вычисления хэш-функции будет представлять собой пару $H=<F,G>$, где $F = \{f\_1, f\_2, ...f\_k\}$ – множество функциональных элементов (функций свертки), преобразующих два блока данных в промежуточное хэш-значение (в случае базовой схемы F состоит из одного элемента – $\{f\}$), а G – управляющее дерево, определяющее порядок вычисления функциональных элементов.

Учитывая тот факт, что применяемые в хэш-алгоритмах функции свёртки, как правило, принимают на вход два блока данных (текущий блок входного файла и промежуточное хэш-значение, полученное на предыдущей итерации) и продуцируют один, естественно попытаться описать процесс вычисления хэш-значения с помощью двоичной иерархической структуры, осуществляющей свертку входного файла к вершине дерева-хэш-значению.

В этой интерпретации листьям такого дерева будут соответствовать блоки входного файла, а остальным узлам дерева, кроме вершины – промежуточные значения применения функции свертки к узлам на предыдущем уровне. Вершина же файла будет соответствовать итоговому хэш-значению файла.

Тогда представление управляющей структуры хэш-алгоритма в виде двоичного дерева имело бы вид, показанный на рис. 3.

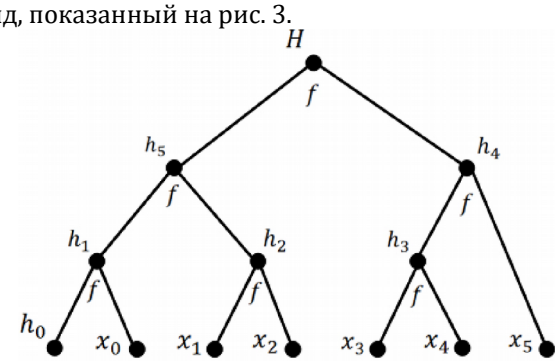

Рис. 3. Пример управляющей структуры в виде двоичного дерева

Утверждение 1. Такая управляющая структура обладает следующими свойствами — это:

1) Возможность использования рекурсивных алгоритмов обхода дерева вычислений хэш-значений без реального развёртывания самой древовидной структуры и хранения ее в памяти.

2) Относительно небольшая высота дерева, пропорциональная $\log_2(n)$, где n – количество блоков разбиения исходных данных.

3) Очевидная независимость конечного результата от выбранного маршрута обхода узлов дерева.

4) Независимость вычисления поддеревьев, что несёт в себе значительный потенциал для применения методов распараллеливания вычислений.

5) При вычислении хэш-значения посредством линейного и бинарного древовидного порядков количество применений функции свёртки одинаково.

Тогда базовая схема вычисления хэш-значений, соответствующая предлагаемому методу, будет состоять из следующих шагов:

1. Начальная обработка входных данных, включая определение длины входного файла в блоках, заполнение при необходимости последнего блока файла последовательностью бит до полноразмерного блока.

2. Генерация управляющей структуры в виде двоичного дерева. Такую генерацию можно назвать виртуальной, так как само дерево не разворачивается в памяти, а моделируется программным способом рекурсивным алгоритмом обхода узлов дерева, которые создаются и редуцируются динамически в процессе их обработки.

3. Обход узлов дерева с помощью выбранного алгоритма обхода и применение функции свертки к данным, соответствующим пройденным узлам-аргументам для получения (промежуточного) хэш-значения и сохранения его в данном узле для использования на более высоком уровне

обработки дерева.

4. Выполнение функции финализации, применяемой к корню дерева.

Базовая схема вычисления хэш-значений может быть расширена с помощью введения следующих дополнительных преобразований:

1) Начальная перестановка блоков.

2) Выбор функции свёртки из некоторого конечного набора функций.

3) Выбор типа дерева.

4) Выбор стратегии и алгоритма обхода дерева, а также способа хранения промежуточных значений.

5) Добавление блоков с мета-данными, например, размером файла.

6) Инициализация начальных значений алгоритма константами или посредством алгоритмических действий.

7) Функция финализации, обеспечивающая дополнительное перемешивание результата для создания лавинного эффекта.

Рассмотрим работу описанной базовой схемы на примере.

Предварительно введем следующие понятия.

Определение 4. **Полное бинарное дерево уровня n** – это дерево, в котором каждый узел уровня *n* является листом и каждый узел уровня меньше *n* имеет непустые правое и левое поддеревья.

Определение 5. **Степенным бинарным деревом** назовём дерево, у которого любое левое поддерево является полным по определению 4.

Пусть есть входной файл, разбитый последовательно на n блоков длины l. Если размер последнего блока данных меньше l, то недостающие биты доопределим заранее выбранным образом, например, блоком (0...01).

Для данного примера будем описывать управляющую структуру хэш-алгоритма в виде степенного бинарного дерева, листья которого будем интерпретировать как блоки данных входного файла, обозначенные на рис. 3 как *x_i*.

Достоинством выбранного типа древа, как отмечалось, является возможность моделировать и интерпретировать такие деревья с помощью рекурсивных последовательных алгоритмов, т.е. без их реального развертывания в памяти вычислителей.

На дереве такого типа выберем алгоритм обхода от левого листа к вершине так, что, если попадаем в некоторый узел, где значение на правом поддереве не вычислено, то сначала вычисляем значение этого поддерева с его левого листа, а затем выполняем вычисления для первоначального узла и продолжаем обход дерева до достижения ситуации, когда будет обработан корень дерева. В частности для рассматриваемого примера порядок обхода узлов показан на рисунке 4.

Рассмотрим характеристики выбранной схемы.

1) Вычислительные:

а) хранение дерева в памяти не требуется, а реализуется рекурсивным алгоритмом обхода дерева;

б) количество применений функции свёртки равно $n-1$, т.е. не превышает число сверток в случае линейно реализации;

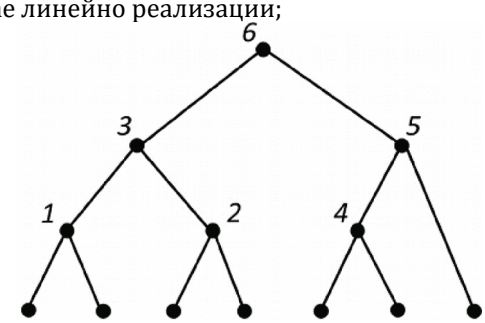

Рис. 4. Порядок операций при обходе дерева

в) требуемая память для хранения промежуточных значений пропорциональна высоте дерева ($\log_2(n)$, где *n* – количество блоков разбиения);

г) возможность параллелизма вычислений.

2) Характеристики безопасности:

а) при выборе функции свёртки, обладающей хорошим параметром перемешивания данных, устраняется зависимость в данных в общем случае не хуже, чем в случае линейной управляющей структуры;

б) при «обращении» хэша противником в случае линейной управляющей структуры на каждой итерации будет получаться блок исходных данных, а в случае древовидной управляющей структуры для получения части исходных данных нужно «обратить» количество итераций, равное высоте поддерева, а если в качестве управляющего дерева использовать деревья, построенные на основе псевдослучайной последовательности, то противник, получив блок исходных данных, даже не будет знать, что он получил.

Рассмотрим характеристики безопасности подробнее:

Устранение зависимостей в данных при линейной управляющей структуре достигается с помощью начальной перестановки блоков и устройством используемой функции свёртки. Последнее, зачастую, увеличивает время выполнения алгоритма, если устранение зависимостей достигается более сложными вычислениями или большим количеством операций/повторений (раундов).

При использовании древовидной управляющей структуры дополнительным фактором перемешивания служит разнообразие вариантов наборов аргументов для функции свёртки, т.е. не только результат предыдущей итерации и следующего блока исходных данных, но и применение функции свёртки и к промежуточным

значениям разных поддеревьев, и к исходным данным.

Теперь рассмотрим криптографическую устойчивость. Как уже было сказано, при линейном варианте противник после каждой итерации получает блок исходных данных, а при древовидной управляющей структуре - не всегда. Также выше были упомянуты "псевдослучайные" деревья, построенные на основе псевдослучайной последовательности. Они обеспечат дополнительный уровень криптографической безопасности в задачах, где требуется проверить подлинность и целостность большого фрагмента данных, передаваемых через открытый информационный канал между пользователями, если для генерации последовательности использовать разделяемый ими секретный ключ.

Важно отметить, что данный подход легко обобщается на случай к-деревьев, т.е. деревьев, каждый неконечный узел которого имеет k исходящих ребер, при этом все сформулированные утверждения для двоичных деревьев переносятся и на случай k- деревьев.

Как уже отмечалось, еще одним достоинством представления управляющих структур хэш-алгоритмов в виде дерева является независимость итогового результата вычисления хэш-значения от порядка свертывания поддеревьев, что обусловливает возможность использовать методы параллельных вычислений для сокращения времени получения результата. В следующем разделе показан один из возможных методов вычисления хэш-значений с использованием параллельных архитектур.

5. Метод реализации хэш-алгоритма с древовидной управляющей структурой на параллельных архитектурах

Вычисление хэш-значений для массивных и сверхмассивных файлов требует больших объёмов вычислений. Естественно, что использование технологий параллельных вычислений даст возможность существенно сократить время вычисления хэш-значений для файлов большого размера. Однако анализ используемых на практике алгоритмов хэширования показывает, что все они построены для реализации с помощью последовательных схем вычисления.

Введённая выше модель хэш-алгоритмов с разделением элементов функциональных преобразований и логики/схемы управления порядком вычислений, реализуемой с помощью древовидной управляющей структуры, предоставляет возможность построения алгоритмов, допускающих применение методов многопоточных вычислений.

Рассмотрим основную идею метода распараллеливания вычисления хэш-значений для предложенного выше подхода, в котором управляющая структура вычислений задается степенными двоичными деревьями.

Общий подход предлагаемого метода включает следующие шаги.

1) Разбиение исходного файла на q порций так, чтобы $q * 2\uparrow k + r = n$, где n – количество блоков в исходном файле, q – число предполагаемых независимых задач, k>1, $2\uparrow k$ – размер порции, определяющий объем работы для одной задачи, которому соответствует поддерево высотой k с $2\uparrow k$ листьями-блоками (полное бинарное дерево из определения 7.4), а r, $0<r=<2\uparrow k$ – оставшийся кусок данных, обрабатываемый (q+1)-ой задачей (заданной степенным деревом с r листьями из определения 7.5). Размер порции файла может определяться, исходя из ресурсных возможностей вычислительной платформы, например, числа доступных вычислительных узлов или размера локальной памяти узла для загрузки в нее целиком порции файла.

2) Независимое исполнение (q+1) задачи (на независимых рабочих процессах/вычислительных узлах), что равносильно свёртке управляющего дерева («обрезанию» снизу ветвей высотой k) к его верхушке-поддереву, т.е. степенному дереву с (q+1) листом, где каждый лист верхушки соответствует промежуточному хэш-значению, полученному в результате свертки исходящей из него ранее ветви высотой k.

3) Если число полученных (q+1) блоков (промежуточных хэш-значений) больше размера порции из $2\uparrow k$ блоков при начальном разбиении, то следует продолжить процесс параллельной редукции верхушки, повторяя пункт 1 для набора (файла) из (q+1) промежуточных хэш-значений (блоков).

4) Вычисление верхушки дерева на родительском процессе и выполнение функции финализации для получения конечного хэш-значения.

Данный метод легко реализуется в среде схемы MapReduce [6] следующим образом:

1) (q+1) задач вычисления поддеревьев размером $2\uparrow k$ из метода выше будут соответствовать q map-задачам (задачам-распределителям), которые будут назначаться мастер-процессом на вычислительные элементы (рабочие процессы).

2) Результатом выполнения задач-распределителей будет выходной файл с промежуточными хэш-значениями.

3) Формирование задачи-редуктора для свёртки промежуточных значений (верхушки дерева) к итоговому значению хэш-кода файла.

Следует отметить, что возможны разнообразные модификации базового алгоритма с целью улучшения его характеристик (повышение степени криптостойкости, усиление лавинного эффекта, снижение времени вычислений хэш-

значения).

В заключение оценим время выполнения хэш-алгоритма по определённому выше методу. Пусть L – размер файла в битах, l – размер блока данных в битах, а p – количество выполняемых параллельно задач. Тогда $n = L/l$ – количество блоков разбиения файла.

Пусть τ – время выполнения одной функции свёртки, а t – время, требуемое на организацию вычислений, связанных с алгоритмом обхода для одного узла. Заметим, что $\tau >> t$.

Тогда время T вычисления хэш-значения будет оцениваться следующим образом:

1) Линейный вариант: $T \approx (n - 1) \times \tau$ (в случае без инициализирующего хэш-значения, иначе – $(n \times \tau)$).

2) Древовидная схема управления: $T \approx (n/p - 1) \times \tau + (p - 1) \times \tau + (n - 1) \times t$,

где

$(n/p-1)\times\tau$ – время свёртки поддерева обработки одной порции файла;

$(p-1) \times \tau$ – время свертки верхушки дерева;

$(n-1)\times t$ – время, требуемое на вычисления, связанные с обходом дерева.

6. Заключение

В представленной работе рассмотрен оригинальный метод построения хэш-функций на основе разделения функциональных элементов хэш-алгоритма и его схемы управления в виде древовидной структуры. Достоинством такого подхода является возможность распараллеливания вычислений хэш-значений и, как следствие, значительного ускорения реализации хэш-алгоритмов на параллельных вычислительных архитектурах, а также гибкость в выборе способов смешивания исходных и промежуточных блоков данных для нивелирования статистических зависимостей в исходных данных. При этом важно отметить, что эти возможности обеспечиваются при относительно небольших накладных расходах на реализацию древовидной схемы управления. Для данного метода приведены временные оценки выполнения. Также предложен способ реализации предложенного хэш-алгоритма на параллельных архитектурах по схеме технологий MapReduce.

Представляется, что данный подход имеет перспективу использования на практике и заслуживает дальнейших исследований его возможностей.

Литература

1. https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5,
2. Donald Knuth The Art of Computer Programming, vol.3. Sorting and Searching. — P. 824. — ISBN 0-201-89685-0.
3. Mineev M.P., Chubarikov V.N. Lectures on arithmetic questions of cryptography. – Moscow. Scientific and Publishing Center «Ray», 2014. – 224p.
4. Laponina. O.R. Fundamentals of network security. Cryptographic algorithms and protocols of interaction. Lecture course. Tutorial. - Internet University of Information Technologies. 2005, 608 p.
5. Peterson W.W., Weldon E.J. Error-Correcting Codes, Pub. "Peace", Moscow, 1976, 595 p.
6. Rajaraman A., Leskovec J., Ullman J.D. Mining of Massive datasets. Moscow, DMK Press, 2016. – 498 p.

References

1. https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5,
2. Donald Knuth The Art of Computer Programming, vol.3. Sorting and Searching. — P. 824. — ISBN 0-201-89685-0.
3. Mineev M.P., Chubarikov V.N. Lectures on arithmetic questions of cryptography. – Moscow. Scientific and Publishing Center «Ray», 2014. – 224p.
4. Laponina. O.R. Fundamentals of network security. Cryptographic algorithms and protocols of interaction. Lecture course. Tutorial. - Internet University of Information Technologies. 2005, 608 p.
5. Peterson W.W., Weldon E.J. Error-Correcting Codes, Pub. "Peace", Moscow, 1976, 595 p.
6. Rajaraman A., Leskovec J., Ullman J.D. Mining of Massive datasets. Moscow, DMK Press, 2016. – 498 p.

Поступила 28.04.2017 г.

Об авторах:

Гапанович Дмитрий Антонович, студент механико-математического факультета, Московский государственный университет имени М.В. Ломоносова, dim.gapanovich@gmail.com

Чубариков Владимир Николаевич, доктор физико-математических наук, профессор, декан механико-математического факультета, Московский государственный университет имени М.В. Ломоносова, chubarik@mech.math.msu.su

Note on the authors:

Gapanovich Dmitry, student of Faculty of Mechanics and Mathematics, Lomonosov Moscow State University, dim.gapanovich@gmail.com

Chubarikov Vladimir, Doctor of Physical and Mathematical Sciences, Professor, Dean of the Faculty of Mechanics and Mathematics, Lomonosov Moscow State University, chubarik@mech.math.msu.su