ЛЕКЦИЯ 5

КРИПТОГРАФИЧЕСКИЕ ХЕШ-ФУНКЦИИ

Хеш-функции впервые упоминаются в середине XX-го века как решение задачи о словаре.

1. Задача о словаре

Задача о словаре состоит в следующем: имеется набор объектов и описаний. Эти объекты хранятся в оперативной памяти или на жёстком диске. Требуется по описанию объектов быстро найти эти объекты (это могут быть файлы в файловой системе, записи в базе данных).

В 1953 году этим занимался **Дональд Кнут**. В 1956 году это было явно описано именно как хеш-функция. В 1968 году на эту тему появляется публикация в «Communications of the ACM» — большая обзорная статья работника Bell Labs, который позже переходит в организацию NSA. Эта статья считается ключевой: в ней впервые подробным образом описывается, какие бывают хеш-функции, для чего они предназначены, какие плюсы и минусы у них есть.

2. Хеш-функции в программировании

2.1. Требования к хеш-таблицам и хеш-функциям

С точки зрения программиста, **хеш-функция** — это некий алгоритм, который переводит строку произвольной длины в строку фиксированной длины (вектор из бит или байт). Если есть файл размером несколько мегабайт или в гигабайт, то с помощью хеш-функции можно получить строчку определённого размера (например, 64 бита, 128 бит, 256 бит). Этот процесс называется **хешированием**. Причём к функции хеширования, которую используют в программировании, предъявляется два требования:

1. **Быстрота вычисления функции** — она не должна требовать каких-то серьёзных ресурсов.

2. **Минимум коллизий**.

2.2. Хеш-таблица

Предположим, что необходимо сохранить какое-нибудь соотношение «ключ-строка», причём этих значений имеется несколько десятков или сотен тысяч. Необходимо разместить их в оперативной памяти компьютера, а потом иметь возможность быстро находить значение по ключу.

...ту ячейку, на которую он указывает. В этой ячейке хранится односвязный или двусвязный список, список дублетов «ключ-значение». Если этот список пустой, то туда просто помещается новая пара «ключ-значение». Если список непустой, тогда проходим по всему списку и проверяем, нет ли совпадений по какому-то ключу. Если нет, то добавляем новый элемент; если есть, тогда заменяем элемент. По такому принципу работают хеш-таблицы.

Теперь рассмотрим подробнее понятие **коллизий**. Предположим, что объектов не 100 тысяч, а 1000 — столько же, сколько корзин выделили для хранения в хеш-таблице. Если в результате вычисления хеш-кода для всех ключей этот хеш-код будет одинаковый, то они все попадут в одну ячейку, а все остальные будут пустые. И тогда операция вставки, поиска, обновления приведёт к постоянному пролистыванию списка. Никакой пользы от введения хеш-таблицы не получится, то есть это плохая хеш-функция, у которой очень много коллизий.

У хорошей хеш-функции должно быть минимальное число коллизий: все пары «ключ-значение» должны равномерно распределяться по хеш-таблице. В зависимости от ключа, его хеш-значения (разных ключей) должны быть максимально различны насколько это возможно с учётом того, что хеш-код принимает ограниченное число значений.

В программировании по мере разрастания хеш-таблицы в неё добавляются новые ячейки, а у хеш-кода увеличивается область допустимых значений. Таким образом, стараются, чтобы в одной ячейке содержалось максимум одно значение, т. е. ключ-значение. Для этого и используется хеш-код в программировании.

Ключ может быть произвольной структуры данных — строчка, число с десятью знаками после запятой или какая-то более сложная структура. Если нужно использовать её в качестве ключа для хеш-таблицы, необходимо реализовать у этого ключа метод хеш-кода.

В языке Java есть функции hashcode() (для вычисления этого ключа) и equals() (для сравнения). Hashcode() возвращает 32-битное значение, то есть может принимать значения от -2^{31} до $(2^{31}-1)$. Но количество ячеек меньше, поэтому хеш-код, после того, как его вычислили для ключа, ещё повторно вычисляется. Самый простой способ — это поделить на n, где n — число корзин в хеш-таблице.

На самом деле этот алгоритм немножко сложнее: он пытается распространить по максимуму значения по всей хеш-таблице, делает определённые предположения о том, какая же функция хеш-кода — плохая или хорошая. Для поиска в таблице функция hashcode() должна быть быстро вычислима, и для того, чтобы эта таблица была заполнена оптимальным образом, число коллизий должно быть минимальным.

! *Конспект не проходил проф. редактуру, создан студентами и, возможно, содержит смысловые ошибки.*
Следите за обновлениями на lectoriy.mipt.ru.

3. Примеры «хороших» хеш-функций

1. **Контрольная сумма** — это число фиксированной длины. Например, в DES используется ключ длиной в 56 бит и контрольная сумма ключа размером 8 бит. Эти 8 бит можно рассматривать как хеш-код от предыдущих 56 бит, который сокращает область значений.

2. **Cyclic redundancy check** — от самых разных длин: от 1-го бита до 64-х бит.

3. **Алгоритм Пирсона** для строк — популярный алгоритм для вычисления хеш-суммы от строки.

4. **Деление на простое число.** Если вычислять хеш-код по модулю числа вида 10^{n} или числа вида 2^{n}, то получится слишком много коллизий для таких чисел (хеш-код от них одинаковый). Поэтому используют деление именно по модулю простого числа.

4. Криптографические хеш-функции

Если рассматривать с точки зрения криптографии, то хеш-функция — это такая функция, которая устойчива ко криптографическим атакам двух типов.

1. **Атака на восстановление первого прообраза.** Предположим, что злоумышленник знает хеш-код некоторого сообщения, то есть он знает, что h — это результат вычисления хеш-функции от какого-то сообщения. Сложность вычисления прообраза — это сложность поиска такого сообщения M, что для него хеш-функция равна заданной.

2. **Устойчивость второго рода.** Когда ничего не задано, кроме алгоритма криптографического хеширования, злоумышленник ищет два таких сообщения, у которых совпадает хеш-функция (задача поиска коллизий). Если такие коллизии легко найти алгоритмически или вычислительно, то рассматриваемая хеш-функция считается плохой с криптографической точки зрения.

Более серьёзное определение использовалось при разработке современного российского стандарта **«Стрибог»**:

1. **Сложность к вычислению прообраза:** есть известно значение функции, тогда должно быть сложно найти такое сообщение, хеш-функция от которого равна известному.

2. **Стойкость вычисления второго прообраза:** пусть есть одно значение, и известен хеш-код этого значения. Тогда злоумышленнику должно быть сложно найти еще одно такое значение, чтобы его хеш-функция совпадала с хеш-функцией первого значения.

3. **Сложность к поиску коллизий:** должно быть сложно найти два таких сообщения, которые не равны, но у них равны хеш-коды.

! *Для подготовки к экзаменам пользуйтесь учебной литературой.*
Об обнаруженных неточностях и замечаниях просьба писать на pulsar@phystech.edu

Конспект не проходил проф. редактуру, создан студентами и, возможно, содержит смысловые ошибки. Следите за обновлениями на lectoriy.mipt.ru.

4. **Стойкость к удлинению прообраза:** если злоумышленник не знает сообщение, но знает его длину и хеш-код от него, то ему должно быть сложно подобрать такое сообщение, которое, будучи дописанным к оригинальному, даст какую-нибудь известную хеш-функцию. Другими словами, не должно быть возможно злоумышленнику что-то менять путём дополнения в сообщении, получая известный выход. Это можно сформулировать по-другому: хеш-функция не должна быть хорошо аддитируема.

5. Идеальная хеш-функция

Будем говорить об идеальной хеш-функции с криптографической точки зрения, у которой длина n (то есть на выходе n бит). Тогда вычисление прообраза должно требовать как минимум 2^{n} операций (под одной операцией имеются в виду операции, сравнимые с вычислением хеширования).

Злоумышленник будет искать прообраз для идеальной хеш-функции следующим образом: у него есть число h, и ему надо найти такое M, что $f(M) = h$. Если это идеальная хеш-функция, то злоумышленнику остается лишь перебирать все возможные M и проверять, чему равна хеш-функция от этого сообщения. Результат вычисления, если M перебирается полностью, есть фактически случайное число. Если число h лежит в диапазоне от 0 до 2^{n}, то тогда в среднем злоумышленник будет тратить на поиски нужного h $\frac{2^{\mathrm{n}}}{2} = 2^{\mathrm{n-1}}$ итераций. Таким образом, вычисление прообраза займёт в два раза меньше итераций, чем в идеальном случае.

Вычисление второго прообраза останется с числом 2^{n}.

В поиске коллизий оценка даст $\sqrt{2^{\mathrm{n}}}$, причём это не совсем точный результат. Это связано с **парадоксом дней рождения**, который состоит в следующем: чтобы хотя бы у двух человек в классе вероятность совпадения даты рождения превышала $\frac{1}{2}$, нужно чуть более 20-ти человек, а не 180, как кажется на первый взгляд. Это происходит из-за того, что нужно анализировать не совпадения у одного человека с другим, а проводить анализ всех возможных пар, которые создаются. Добавление одного человека увеличивает количествой сочетаний не на одну пару, а на n пар, где n — число уже существующих. Число таких пар растёт примерно квадратично. Поэтому, самая грубая оценка числа человек в классе — $\sqrt{365}$. Чем больше это число, тем точнее эта оценка (по закону больших чисел). Таким образом, при достаточно больших числах (2^{n}) оценка $\sqrt{2^{\mathrm{n}}}$ будет достаточно точной.

Если злоумышленник хочет написать программу по поиску коллизий, ему будет оптимально вначале завести себе словарь коллизий. Соответственно, дальше он вычисляет хеш-функцию от очередного сообщения и проверяет, принадлежит эта хеш-функция очередному сообщению или нет. Если принадлежит, то коллизия найдена, и тогда можно найти по словарю исходное сообщение с данным хеш-кодом. Если нет, то он пополняет словарь.

На деле таким алгоритмом никто не пользуется, потому что не хватит памяти для словаря. Но если бы памяти хватило, то этот алгоритм имел бы вычислительную сложность порядка $\sqrt{2^{\mathrm{n}}}$. Для поиска прообразов используются так называемые **радужные таблицы**.

Удлинение прообраза займёт 2^{n} операций, потому что придётся перебирать все воз-

Для подготовки к экзаменам пользуйтесь учебной литературой. Об обнаруженных неточностях и замечаниях просьба писать на pulsar@phystech.edu

можные удлинения суффикса M.

6. «Стрибог»

У стандарта на хеш-функцию ГОСТ Р 34.11-94 были обнаружены уязвимости. Ожидаемая стойкость была одна, а фактическая стойкость (необходимое количество операций для поиска прообраза) оказалась другой.

Чтобы пройти соответствие международным стандартам, была придумана новая функция, которую условно назвали «Стрибог». Концепция этой криптографической хеш-функции состоит в использовании минимального числа элементов, но необходимого для того, чтобы обеспечить устойчивость ко всем известным криптографическим атакам на хеш-функцию.

Принципы построения хеш-функции «Стрибог»:

1. Неприменимость известных атак.
2. Использование хорошо изученных конструкций и преобразований.
3. Обеспечение каждого структурного элемента конкретными свойствами.
4. Использование самого простого для анализа и реализации варианта (если их несколько).
5. Максимальная производительность программной реализации.

В этой хеш-функции используются разные конструкции.

6.1. Конструкция Меркле – Дамгарда

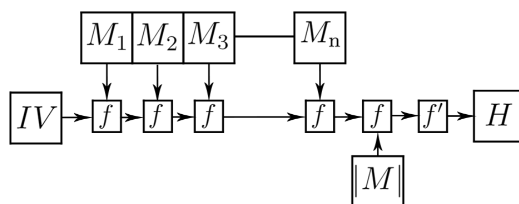

Рис. 5.1

f — раундовая функция сжатия, $|M|$ — битовая длина сообщения, f' — завершающее преобразование.

Сообщение разбивается на блоки. Используется вектор инициализации и раундовая функция, куда подаются два входа. Подаётся некий блок сообщений и то, что получилось на предыдущем этапе. На первом блоке используется вектор инициализации. Таких этапов проводится много, пока всё сообщение не обработается. Затем проводится завершающее преобразование.

Если длина сообщения не кратна 64 битам, то оно дополняется некоторым блоком, а в конце дописывается число целых блоков. Это проводится для того, чтобы было трудно провести атаку на удлинение прообраза.

Свойства MD-конструкции:

1. Если длины двух разных сообщений различны, последние блоки после дополнения должны гарантированно различаться.

2. При этом хеш-функция устойчива к коллизии, если раундовая функция устойчива к коллизии.

3. Без завершающего преобразования при известной коллизии можно легко найти другие коллизии, удлинняя сообщение; при известной коллизии можно найти мультиколлизии; если известно $H(X)$ при неизвестном X, то нетрудно найти $H(X||Y)$ для любого Y.

Сложность поиска прообраза и сложность удлинения прообраза составляют 2^{n}, коллизии — $\sqrt{2^{\mathrm{n}}}$. Поиск второго прообраза — $\frac{2^{\mathrm{n}}}{L}$, где L — длина сообщения. Именно из-за этого ограничения для каждой криптографической хеш-функции, которая основана на MD-кодах, всегда указывают, какова максимальная длина файлов, которые можно использовать в качестве аргумента хеширования.

6.2. Завершающее преобразование

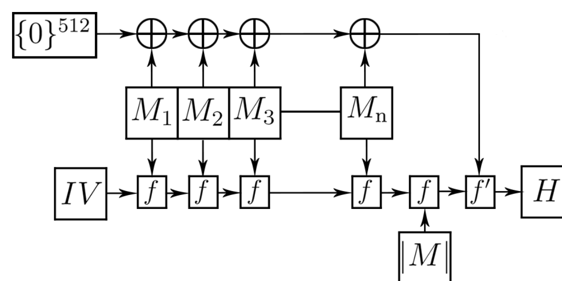

Рис. 5.2

Все блоки сообщения суммируются по модулю 2^{512}, и в конце результат используется как вход для завершающей функции. Сложение происходит непобитово. Это обеспечивает защиту от нескольких атак: защита от удлинения прообраза, от мультиколлизий и от дифференциального криптоанализа.

6.3. Конструкция Миагучи – Пренели

Надёжная хеш-функция должна обечпечивать шифрование так, чтобы не было возможности восстановить предыдущий блок $H_{\mathrm{i-1}}$ и вход M, зная выход H. Также не должно быть возможно восстановить предыдущий блок, зная M и H.

Любая функция шифрования (DES, GOST, AES) обеспечивает выполнение данных условий.

Однако существуют дополнительные требования специально для хеш-функций.

Конструкция **Миагучи – Пренели** выглядит следующим образом.

Используется некая функция шифрования, но кроме того, что сообщение и предыдущий результат передаются как аргументы функции шифрования, само сообщение и предыдущий результат складываются по модулю 2 с результатом шифрования. Это делается для обеспечения отсутствия фиксированных точек (пара (H_{i}, M) называется

! **_Конспект не проходил проф. редактуру, создан студентами и, возможно, содержит смысловые ошибки. Следите за обновлениями на lectoriy.mipt.ru._**

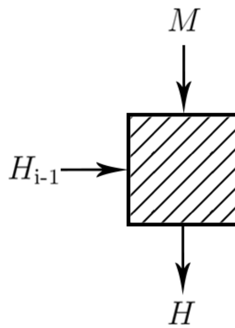

Рис. 5.3

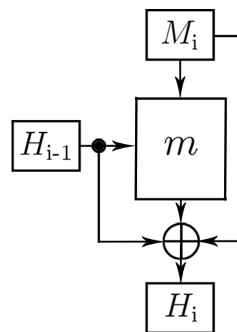

Рис. 5.4

фиксированной точкой, если $f(H_i, M) = H_i$).

Если не использовать сложение сообщения и предыдущего результата, то злоумышленник сможет понять, что если выход совпадает с предыдущим блоком, тогда он сможет сделать определённые выводы о самом сообщении. Или наоборот, он может пытаться подобрать такие сообщения, чтобы на выходе получались одинаковые блоки.

6.4. Функция сжатия

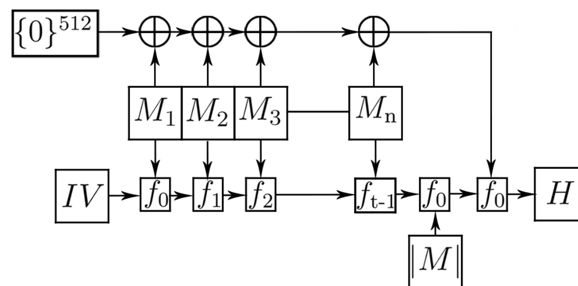

Рис. 5.5

Здесь на каждом раунде меняется функция сжатия.

Функция сжатия в новом стандарте ГОСТ основана на **XSPL-шифре** — это шифр, основанный на таких операциях как побитовое сложение, SP-сеть и линейное преобразование.

! **_Для подготовки к экзаменам пользуйтесь учебной литературой. Об обнаруженных неточностях и замечаниях просьба писать на_** pulsar@phystech.edu

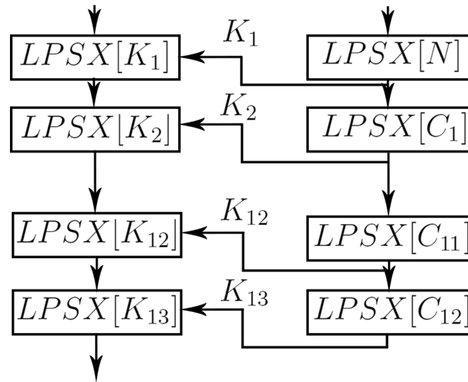

Рис. 5.6

Расписание ключей является важной частью этого шифра. Здесь функция генерации ключа такая же, как функция раунда шифрования. Используется некое инициализирующее значение. Шифруя это значение с некоторыми константами, получаем раундовые ключи. Причём первой константой N здесь выступает число раундов функции хеширования. Это как раз то число, которое меняется для каждой функции сжатия. $C_1 ... C_{12}$ — это некие константы, которые заданы в стандарте.

Результат этого шифрования используется как раундовые ключи. Есть фрагмент сообщения, который подвергается хешированию (оно используется как аргумент шифрования). В качестве ключа используются ключи K_1, K_2 и т. д.

Для того, чтобы хеш-функция была устойчивой, используется надёжный блочный шифр в качестве элемента конструкции хеш-функции.

Существует множество криптографических атак, и чтобы от них защититься, необходимо использовать такие сложные конструкции как блочные шифры.

В **XSPL-шифре** 13 итераций. В нём используется сложение по модулю 2 векторов по 512 бит, нелинейная взаимнооднозначная замена байтов, перестановка внутри 512-битного вектора, преобразование i-го бита ($i \to 8i \mod 63$) и умножение векторов по 64 бит на фиксированную матрицу.

Этот шифр используется для того, чтобы предотвратить различные атаки на хеш-функцию: атаки скольжения, атаки отражения, разностных связных ключей и т. д. (атаки и на сам блочный шифр, и на результирующую функцию).

В этом шифре, в отличие от ГОСТа 1994-го года, блок дополняется нулями с единицей в конце (00...01) (раньше добавлялись только нули). Это усложняет атаку удлинением.

Ранее вектор инициализации не был фиксирован, то есть он не был описан в стандарте, но сейчас вектор инициализации состоит либо из всех нулей и единицы в конце для функции длиной 256 бит, либо из одних нулей для 512-битной функции.

Стандарт внедряет две функции хеширования разной длины, то есть из одного сообщения можно получить либо хеш-функцию длиной 256 бит (это минимальная возможная длина) либо 512 бит, которая считается более надёжной. Для них используются немного разные векторы инициализации.

XSPL обладает производительностью 51 такт на 1 байт исходного текста. Это намного медленнее, чем какие-нибудь генераторы псевдослучайной последовательности, но более надёжно.

7. Практическое применение хеш-функций

7.1. Проверки целостности

Многие организации, выкладывая файлы на сервер, дополнительно указывают его **чек-сумму** — это код, который человек, скачав файл, может вычислить самостоятельно и сравнить с тем, что указано на сайте. Если этот код совпадает, значит файл как минимум был скопирован с того же сайта, то есть его не подменили по пути и не внедрили в него вирусы.

Указывают обычно две чексуммы. Например CRC32 (это стандартный циклический код), MD5. В последнее время указывают SHA-2 или SHA-3 (SHA-256 либо SHA-512).

Допустим, имеется файл, который представляет из себя сообщение M, и злоумышленник хочет внедрить в него вирус. Если на сайте выложен код **CRC32** (он не стоек с криптографической точки зрения), то злоумышленник, внедрив вирус в файл, может дополнительно поменять файл таким образом, чтобы CRC совпали с предыдущим файлом. Более того, CRC32 короткий — 32 бита на выходе (можно перебрать полным перебором).

С **MD5** ситуация обстоит сложнее, но, тем не менее, для MD5 уже найдено множество коллизий, составлены радужные таблицы, и подменить файл достаточно просто.

SHA считаются надёжными с криптографической точки зрения. Если злоумышленник поменял в сообщении M несколько байт, то хеш-код полностью изменится. Злоумышленник будет пытаться дополнить этот файл бессмысленными наборами символов, которые не повлияют на работу вирусов и программы, таким образом, чтобы хеш-код от нового файла совпадал со старым файлом. Но он это сможет сделать, только если здесь будет использоваться ненадёжная хеш-функция. Если хеш-функция надёжная, то перебор файлов займёт слишком много времени.

7.2. Электронная подпись

Чтобы убедиться, что сообщение отправил конкретный отправитель, вместе с сообщением передаётся так называемая электронная подпись. Получатель проверяет, действительно ли электронная подпись относится к данному сообщению. Используя криптографию с открытыми ключами, такую электронную подпись построить возможно.

Но операции с открытыми ключами (подписывание, проверка подписей и т. д.) очень медленные (40 Кбайт/с). Более того, если подписывать всё сообщение целиком, то размеры этой подписи будут сопоставимы с размером сообщения. Поэтому подписывают не сообщение, а хеш-функцию от сообщения. И далее получатель, когда расшифровывает подпись, получает хеш-функцию. Далее он сравнивает хеш-функцию от того сообщения, которое он получил, и хеш-функцию, которая была получена в результате расшифровки. За счет того, что хеш-функция имеет фиксированную длину, она меньше, чем само сообщение. За счёт этого можно быстро вычислить электронную цифровую подпись. Размер этой подписи будет мал по сравнению с размером сообщения.

Если хеш-функция плохая, тогда злоумышленник сможет подменить сообщение таким образом, чтобы хеш-функция осталась прежней. А если хеш-функция хорошая, тогда этот алгоритм будет работать.