Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования «Российский химико-технологический университет имени Д.И. Менделеева»

Факультет цифровых технологий и химического инжиниринга

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 9

ПО КУРСУ

«АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ»

Ведущий преподаватель

Ассистент Крашенинников Р. С.

СТУДЕНТ группы КС-36 Лупинос А. В.

Москва

2025

Задание

В рамках лабораторной работы необходимо реализовать алгоритм хеширования RIPEMD-160.

Для реализованной хеш-функции провести следующие тесты:

1. Сгенерировать 1000 пар строк длиной 128 символов, отличающихся друг от друга 1, 2, 4, 8, 16 символами, и сравнить хеши для пар между собой, проведя поиск одинаковых последовательностей символов в хешах и подсчитав максимальную длину такой последовательности. Результаты для каждого количества отличий нанести на график, где по оси абсцисс количество отличий, а по оси ординат максимальная длина одинаковой последовательности.
2. Провести $N = 10^i$ (i от 2 до 6) генераций хешей для случайно сгенерированных строк длиной 256 символов и выполнить поиск одинаковых хешей в итоговом наборе данных, результаты привести в таблице, где первая колонка это N генераций, а вторая таблица наличие и количество одинаковых хешей, если такие были.
3. Провести 1000 генераций хеша для строк длиной $N = 2^i$ (i от 6 до 13), строки генерировать случайно для каждой серии; подсчитать среднее время и построить зависимость скорости расчета хеша от размера входных данных.

Описание алгоритма

Хеширование представляет собой процесс преобразования данных, таких как текст или файл, в код фиксированной длины, именуемый **хешем**. Хеш является уникальным идентификатором для каждого набора данных и используется для проверки их целостности или ускорения поиска. Например, хеш пароля обеспечивает безопасное хранение, а хеш файла подтверждает отсутствие изменений.

Хеш-функция — алгоритм, преобразующий данные произвольного размера в хеш фиксированной длины. В криптографии хеш-функции предотвращают подделку данных и обеспечивают защиту от несанкционированного доступа.

Хеш-функции применяются в следующих областях:

- Безопасное хранение паролей в базах данных.
- Проверка целостности файлов при их загрузке.
- Генерация цифровых подписей для защиты сообщений.
- Оптимизация поиска в хеш-таблицах[1].

Хорошая хеш-функция должна:

- Быть детерминированной (одинаковый вход всегда дает одинаковый хеш).
- Работать быстро, чтобы не замедлять программы.
- Иметь лавинный эффект (малое изменение входа сильно меняет хеш).
- Быть устойчивой к обратному вычислению (нельзя восстановить данные по хешу).
- Минимизировать коллизии (случаи, когда разные входы дают одинаковый хеш).

[1] Структура данных, предназначенная для хранения пар "ключ-значение" и обеспечивающая быстрый поиск, добавление и удаление элементов. Основной принцип ее работы заключается в использовании хеш-функции, которая преобразует ключ в индекс массива, где хранится соответствующее значение. Например, используется хеш-таблица для хранения пар "имя-возраст":

- Ключ "Anna" → хеш-функция выдает индекс 3 → значение 25 сохраняется в массиве по индексу 3.
- Для поиска возраста по имени "Anna" хеш-функция снова выдает индекс 3, и извлекается значение 25.

Криптографические хеш-функции, такие как RIPEMD-160, обладают дополнительными свойствами:

- Устойчивость к коллизиям: найти два разных входа с одинаковым хешем крайне сложно.
- Устойчивость к первым прообразам[2]: нельзя найти вход, зная только хеш.
- Устойчивость к вторым прообразам[3]: нельзя найти другой вход, дающий тот же хеш, что и заданный.

Коллизия возникает, когда два различных набора входных данных генерируют одинаковый хеш. Поскольку хеш-функция отображает бесконечное множество входов на конечное множество хешей, коллизии неизбежны. Однако криптографические хеш-функции, такие как RIPEMD-160, минимизируют вероятность коллизий, что критически важно для обеспечения безопасности.

RIPEMD-160 представляет собой криптографическую хеш-функцию, разработанную в 1996 году в Католическом университете Левена (Бельгия) группой ученых, включая Ханса Доббертина, Антуана Босселаерса и Барта Пренеля. Функция генерирует 160-битный (20-байтный) хеш и является улучшенной версией алгоритма RIPEMD, основанного на принципах MD4. Название RIPEMD расшифровывается как "RACE Integrity Primitives Evaluation Message Digest" (RIPEMD-160).

Алгоритм RIPEMD-160 обрабатывает входные данные в несколько этапов:

1. Подготовка входных данных (Padding — дополнение):

Первым этапом является подготовка входных данных для обработки. Длина входного сообщения должна быть приведена к значению, кратному 512 битам (64 байта), чтобы его можно было разделить на блоки фиксированного размера.

- Добавление бита '1': к концу сообщения добавляется один бит, равный 1 (в байтовом представлении это 0x80, или 10000000 в двоичном виде)

[2] Исходные данные, которые, при обработке хеш-функцией, дают конкретное значение хеша.
[3] Данные, которые, при обработке хеш-функцией, дают такое же значение хеша.

- Добавление нулей: после бита '1' добавляется необходимое количество нулевых битов, чтобы длина сообщения стала равной 448 битам по модулю 512. Это означает, что длина дополненного сообщения без учета последних 64 бит должна быть кратна 512.
    - Если длина исходного сообщения уже соответствует этому условию (например, 448 бит), все равно добавляется минимум 512 бит дополнения (то есть один полный блок).
- Добавление длины сообщения: В последние 64 бита (8 байт) записывается длина исходного сообщения (до дополнения) в битах как 64-битное целое число в формате little-endian (младший байт идет первым).
    - Если длина сообщения превышает $2^{64} - 1$ бит, записываются только младшие 64 бита длины.

Результат: после дополнения длина сообщения становится кратной 512 битам, и его можно разделить на блоки по 512 бит (16 слов по 32 бита).

Пример:

- Входное сообщение "abc" (3 символа, или 24 бита): 'a' = 0x61, 'b' = 0x62, 'c' = 0x63.
- Добавляем 0x80 (1 бит): $24 + 8 = 32$ бита.
- Добавляем нули до 448 бит: $448 - 32 = 416$ бит (52 байта) нулей.
- Добавляем длину (24 бита как 64-битное число): 0x00000018 0x00000000 (младший байт идет первым).
- Итоговая длина: 512 бит (1 блок).

2. Инициализация регистров состояния:

RIPEMD-160 использует пять 32-битных регистров состояния (A, B, C, D, E), которые также называют h0, h1, h2, h3, h4. Эти регистры инициализируются стандартными значениями в шестнадцатеричном формате (в формате little-endian): $h0 = 0x67452301$, $h1 = 0xEFCDAB89$, $h2 = 0x98BADCFE$, $h3 = 0x10325476$, $h4 = 0xC3D2E1F0$.

Эти значения выбраны так, чтобы минимизировать влияние начальных условий на результат и повысить безопасность алгоритма. Итоговый хеш будет сформирован из этих регистров после обработки всех блоков.

3. Разбиение на блоки:

Дополненное сообщение делится на блоки по 512 бит (64 байта). Каждый блок состоит из 16 слов по 32 бита (16 × 32 = 512). Эти слова обозначаются как $X[0]$, $X[1]$, ..., $X[15]$. Каждое слово извлекается из блока в формате little-endian, то есть младший байт идет первым.

Пример:

- Байты 0–3: 0x61 0x62 0x63 0x80 → $X[0]$ = 0x80636261;
- Байты 4–7: 0x00 0x00 0x00 0x00 → $X[1]$ = 0x00000000
- Байты 8–11: 0x00 0x00 0x00 0x00 → $X[2]$ = 0x00000000;
- ...
- Байты 56–59: 0x18 0x00 0x00 0x00 → $X[14]$ = 0x00000018;
- Байты 60–63: 0x00 0x00 0x00 0x00 → $X[15]$ = 0x00000000.

4. Обработка каждого блока: параллельные ветви:

RIPEMD-160 обрабатывает каждый 512-битный блок в 80 шагов (5 раундов по 16 шагов в каждой из двух параллельных ветвей). Параллельная обработка в двух ветвях (левой и правой) — ключевая особенность алгоритма, которая повышает его безопасность, усложняя атаки, такие как дифференциальный криптоанализ.

4.1. Инициализация ветвей:

Для каждого блока создаются две копии регистров состояния:

- Левая ветвь: A, B, C, D, E;
- Правая ветвь: A', B', C', D', E'.

Обе ветви изначально инициализируются текущими значениями регистров $h0, h1, h2, h3, h4$:

- $A = A' = h0$;
- $B = B' = h1$;
- $C = C' = h2$;
- $D = D' = h3$;
- $E = E' = h4$.

4.2. Раунды и шаги:

Каждая ветвь выполняет 80 шагов (5 раундов по 16 шагов). Параметры для каждого шага включают:

- Логические функции (F, G, H, I, J);
- Константы (K_j для левой ветви, K'_J для правой);
- Перестановки слов ($r[j]$ и $r'[j]$);
- Сдвиги ($s[j]$ и $s'[j]$).

В каждом раунде используется одна из пяти логических функций, которые применяются к трем регистрам (B, C, D или B', C', D'). Логические функции для каждого раунда:

- Раунд 1 (шаг 0–15): $F(x, y, z) = x \oplus y \oplus z$ (XOR);
- Раунд 2 (шаг 16–31): $G(x, y, z) = (x \wedge y) \vee (\neg x \wedge z)$;
- Раунд 3 (шаг 32–47): $H(x, y, z) = (x \vee \neg z) \oplus y$;
- Раунд 4 (шаг 48–63): $I(x, y, z) = (x \wedge z) \vee (y \wedge \neg z)$;
- Раунд 5 (шаг 64–79): $J(x, y, z) = x \oplus (y \vee \neg z)$.

В правой ветви порядок обратный: J, I, H, G, F.

Каждый шаг использует предопределенные константы:

- Для левой ветви:
  - Раунд 1: $K_0 = 0$x00000000;
  - Раунд 2: $K_1 = 0$x5A827999;

- Раунд 3: $K_2 = 0x6ED9EBA1$;
- Раунд 4: $K_3 = 0x8F1BBCDC$;
- Раунд 5: $K_4 = 0xA953FD4E$.

- Для правой ветви:
  - Раунд 1: $K'_0 = 0x50A28BE6$;
  - Раунд 2: $K'_1 = 0x5C4DD124$;
  - Раунд 3: $K'_2 = 0x6D703EF3$;
  - Раунд 4: $K'_3 = 0x7A6D76E9$;
  - Раунд 5: $K'_4 = 0x00000000$.

Эти константы основаны на математических значениях (например, $\sqrt{2}$, $\sqrt{3}$) и добавляются для повышения сложности атак.

Перестановка слов. Каждый шаг использует одно из 16 слов блока $X[i]$, но порядок выбора слов различается. На рисунке 1 представлен выбор 32-битных слов из сообщения для левой $r(j)$ и для правой $r'(j)$ ветвей, где j – номер шага (начиная с 0 до 79).

- r(j) = j при $(0 \le j \le 15)$
- r(16..31) = 7; 4; 13; 1; 10; 6; 15; 3; 12; 0; 9; 5; 2; 14; 11; 8
- r(32..47) = 3; 10; 14; 4; 9; 15; 8; 1; 2; 7; 0; 6; 13; 11; 5; 12
- r(48..63) = 1; 9; 11; 10; 0; 8; 12; 4; 13; 3; 7; 15; 14; 5; 6; 2
- r(64..79) = 4; 0; 5; 9; 7; 12; 2; 10; 14; 1; 3; 8; 11; 6; 15; 13
- r'(0..15) = 5; 14; 7; 0; 9; 2; 11; 4; 13; 6; 15; 8; 1; 10; 3; 12
- r'(16..31) = 6; 11; 3; 7; 0; 13; 5; 10; 14; 15; 8; 12; 4; 9; 1; 2
- r'(32..47) = 15; 5; 1; 3; 7; 14; 6; 9; 11; 8; 12; 2; 10; 0; 4; 13
- r'(48..63) = 8; 6; 4; 1; 3; 11; 15; 0; 5; 12; 2; 13; 9; 7; 10; 14
- r'(64..79) = 12; 15; 10; 4; 1; 5; 8; 7; 6; 2; 13; 14; 0; 3; 9; 11

Рисунок 1 – Выбор 32-битных слов из сообщения для левой и правой ветвей

Каждый шаг включает циклический сдвиг влево (rotate left, или rol) на количество бит, заданное массивами $s[j]$ и $s'[j]$ для левой и правой ветви соответственно, где j – номер шага от 0 до 79. Значения сдвигов для каждого шага находятся в диапазоне от 0 до 15 для левой и правой ветви. Наборы для поворота показаны на рисунке 2.

- s(0..15) = 11; 14; 15; 12; 5; 8; 7; 9; 11; 13; 14; 15; 6; 7; 9; 8
- s(16..31) = 7; 6; 8; 13; 11; 9; 7; 15; 7; 12; 15; 9; 11; 7; 13; 12
- s(32..47) = 11; 13; 6; 7; 14; 9; 13; 15; 14; 8; 13; 6; 5; 12; 7; 5
- s(48..63) = 11; 12; 14; 15; 14; 15; 9; 8; 9; 14; 5; 6; 8; 6; 5; 12
- s(64..79) = 9; 15; 5; 11; 6; 8; 13; 12; 5; 12; 13; 14; 11; 8; 5; 6
- s'(0..15) = 8; 9; 9; 11; 13; 15; 15; 5; 7; 7; 8; 11; 14; 14; 12; 6
- s'(16..31) = 9; 13; 15; 7; 12; 8; 9; 11; 7; 7; 12; 7; 6; 15; 13; 11
- s'(32..47) = 9; 7; 15; 11; 8; 6; 6; 14; 12; 13; 5; 14; 13; 13; 7; 5
- s'(48..63) = 15; 5; 8; 11; 14; 14; 6; 14; 6; 9; 12; 9; 12; 5; 15; 8
- s'(64..79) = 8; 5; 12; 9; 12; 5; 14; 6; 8; 13; 6; 5; 15; 13; 11; 11

Рисунок 2 – Наборы для битового поворота влево (операция rol)

4.3. Основное преобразование на шаге. Для каждого шага j (от 0 до 79):

- Левая ветвь:
  a. Вычисляется $F(B, C, D)$ (в зависимости от раунда);
  b. Выбирается слово $X[r[j]]$;
  c. Выполняется операция (все сложения выполняются по модулю 2^{32}[4]):

$$T = A + F(B, C, D) + X[r[j]] + K_j;$$

  d. Выполняется циклический сдвиг: $T = ROL(T, s[j])$;

[4] При выполнении операции сложения результат ограничивается 32 битами, а любое переполнение (старшие биты) отбрасывается. Это стандартный подход в криптографических алгоритмах, таких как RIPEMD-160, для работы с 32-битными числами. Пример: рассмотрим сложение двух чисел: 0*xEFB*18863 + 0*xC*3*D*2*E*1*F*0
Шаг 1: Сложение в обычном виде 0*xEFB*18863 = 4015800163 (в десятичной системе), 0*xC*3*D*2*E*1*F*0 = 3284165104. Тогда, 4015800163 + 3284165104 = 7299965267
Шаг 2: Переполнение. Максимальное 32-битное число: 2^32 - 1 = 4294967295. 7299965267 > 4294967295, значит, есть переполнение.
Шаг 3: Применение модуля. 2^32 = 4294967296. 7299965267 mod 4294967296 =7299965267 − 4294967296 = 3004997971. В шестнадцатеричном виде: 3004997971 = 0*xB*2*E*46973

e. Добавляется значение E: $T = T + E$;

f. Обновляются регистры:

$$E = D,\ D = C,\ C = ROL(B, 10),\ B = A,\ A = T.$$

- Правая ветвь. Аналогично, но с другими функциями, константами, перестановками и сдвигами:
  a. Вычисляется $F'(B', C', D')$ (в обратном порядке);
  b. Выбирается слово $X[r'[j]]$;
  c. Выполняется: $T' = A' + F'(B', C', D') + X[r'[j]] + K'_j$;
  d. Сдвиг: $T' = ROL(T', s'[j])$;
  e. Добавляется E': $T' = T' + E'$;
  f. Обновляются регистры:

$$E' = D',\ D' = C',\ C' = ROL(B', 10),\ B' = A',\ A' = T'.$$

4.4. Комбинирование ветвей:

После выполнения 80 шагов в обеих ветвях результаты комбинируются с текущими значениями регистров $h0,\ h1,\ h2,\ h3,\ h4$:

- $h0 = h0 + C + D'$;
- $h1 = h1 + D + E'$;
- $h2 = h2 + E + A'$;
- $h3 = h3 + A + B'$;
- $h4 = h4 + B + C'$.

Сложение выполняется по модулю 2^{32}. После этого обработка текущего блока завершена, и алгоритм переходит к следующему блоку, если он есть.

5. Формирование итогового хеша:

После обработки всех блоков значения регистров $h0$, $h1$, $h2$, $h3$, $h4$ объединяются в итоговый 160-битный хеш:

- Регистры записываются в формате little-endian (младший байт первого регистра идет первым);
- Итоговый хеш представляется как строка из 40 шестнадцатеричных символов (20 байт).

Пример: если после обработки $h0 = 0x9c1185a5$, $h1 = 0xc5e9fc54$, ..., то итоговый хеш – $9c1185a5c5e9fc54612808977ee8f548b2258d31$ (сообщение-пустая строка “”).

Алгоритм RIPEMD-160 обладает рядом характеристик, обеспечивающих его эффективность и криптографическую стойкость, что делает его пригодным для различных приложений. Однако существуют и определенные ограничения, связанные с вычислительной сложностью и потенциальными уязвимостями.

Преимущества:

- Лавинный эффект: изменение одного бита входных данных меняет около 50% битов хеша, усиливая защиту.
- Параллельная структура: две ветви обработки усложняют криптоанализ, повышая безопасность.
- Производительность: скорость около 45.5 Мбит/с сравнима с SHA-1, подходит для многих приложений.

Сложности и проблемы:

- Вычислительная нагрузка: 80 шагов на блок требуют больше операций, чем у MD5 (64 шага).
- Ограниченная длина хеша: 160 бит менее безопасны, чем 256 или 512 бит в современных алгоритмах.
- Уязвимость к атакам на основе длины: возможны атаки, позволяющие дополнять хеш без исходного сообщения.

Описание выполнения задачи

Для реализации задачи была написана программа на C++:

```
#include <iostream>

#include <fstream>

#include <string>

#include <cstring>

#include <random>

#include <chrono>

#include <iomanip>

#include <unordered_set>

#include <cmath>


using namespace std;


// Класс, реализующий алгоритм хеширования RIPEMD-160

class RIPEMD160 {

private:

   uint32_t state[5]{};        // Массив для хранения текущего состояния хеша (5 регистров)

   uint64_t count;           // Счетчик байтов, обработанных алгоритмом

   unsigned char buffer[64]{}; // Временный буфер для данных, пока не наберется 64 байта


   // Константы для разных раундов преобразований

   static constexpr uint32_t K[5] = {

         0x00000000, 0x5A827999, 0x6ED9EBA1, 0x8F1BBCDC, 0xA953FD4E

   };


   static constexpr uint32_t KK[5] = {

         0x50A28BE6, 0x5C4DD124, 0x6D703EF3, 0x7A6D76E9, 0x00000000
```

```
};

// Порядок использования слов сообщения в левой ветви алгоритма
static constexpr int r[80] = {
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
    7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8,
    3, 10, 14, 4, 9, 15, 8, 1, 2, 7, 0, 6, 13, 11, 5, 12,
    1, 9, 11, 10, 0, 8, 12, 4, 13, 3, 7, 15, 14, 5, 6, 2,
    4, 0, 5, 9, 7, 12, 2, 10, 14, 1, 3, 8, 11, 6, 15, 13
};

// Порядок использования слов в правой ветви (параллельная обработка)
static constexpr int rr[80] = {
    5, 14, 7, 0, 9, 2, 11, 4, 13, 6, 15, 8, 1, 10, 3, 12,
    6, 11, 3, 7, 0, 13, 5, 10, 14, 15, 8, 12, 4, 9, 1, 2,
    15, 5, 1, 3, 7, 14, 6, 9, 11, 8, 12, 2, 10, 0, 4, 13,
    8, 6, 4, 1, 3, 11, 15, 0, 5, 12, 2, 13, 9, 7, 10, 14,
    12, 15, 10, 4, 1, 5, 8, 7, 6, 2, 13, 14, 0, 3, 9, 11
};

// Величины сдвигов для левой ветви в каждом из 80 шагов
static constexpr int s[80] = {
    11, 14, 15, 12, 5, 8, 7, 9, 11, 13, 14, 15, 6, 7, 9, 8,
    7, 6, 8, 13, 11, 9, 7, 15, 7, 12, 15, 9, 11, 7, 13, 12,
    11, 13, 6, 7, 14, 9, 13, 15, 14, 8, 13, 6, 5, 12, 7, 5,
    11, 12, 14, 15, 14, 15, 9, 8, 9, 14, 5, 6, 8, 6, 5, 12,
    9, 15, 5, 11, 6, 8, 13, 12, 5, 12, 13, 14, 11, 8, 5, 6
};
```

```
    // Величины сдвигов для правой ветви
    static constexpr int ss[80] = {
        8, 9, 9, 11, 13, 15, 15, 5, 7, 7, 8, 11, 14, 14, 12, 6,
        9, 13, 15, 7, 12, 8, 9, 11, 7, 7, 12, 7, 6, 15, 13, 11,
        9, 7, 15, 11, 8, 6, 6, 14, 12, 13, 5, 14, 13, 13, 7, 5,
        15, 5, 8, 11, 14, 14, 6, 14, 6, 9, 12, 9, 12, 5, 15, 8,
        8, 5, 12, 9, 12, 5, 14, 6, 8, 13, 6, 5, 15, 13, 11, 11
    };

    // Логические функции для каждого раунда преобразований
    static uint32_t F(uint32_t x, uint32_t y, uint32_t z) { return x ^ y ^ z; }          // Простое исключающее ИЛИ
    static uint32_t G(uint32_t x, uint32_t y, uint32_t z) { return (x & y) | (~x & z); }  // Комбинация И, ИЛИ и НЕ
    static uint32_t H(uint32_t x, uint32_t y, uint32_t z) { return (x | ~y) ^ z; }          // Сложная комбинация с инверсией
    static uint32_t I(uint32_t x, uint32_t y, uint32_t z) { return (x & z) | (y & ~z); }  // Другая логическая комбинация
    static uint32_t J(uint32_t x, uint32_t y, uint32_t z) { return x ^ (y | ~z); }        // Финальная функция с ИЛИ и НЕ

    // Циклический сдвиг влево на заданное число бит
    static uint32_t ROL(uint32_t x, int n) { return (x << n) | (x >> (32 - n)); }

    // Основная функция преобразования блока данных (64 байта)
    void transform(const unsigned char* data) {
        // Инициализируем временные регистры текущими значениями состояния
        uint32_t a = state[0], b = state[1], c = state[2], d = state[3], e = state[4];
        uint32_t aa = a, bb = b, cc = c, dd = d, ee = e;
        uint32_t x[16]; // Массив для хранения 16 слов по 32 бита из входных данных

        // Преобразуем входной блок в 16 слов (little-endian порядок)
        for(int i = 0; i < 16; i++)
```

```
    x[i] = (data[4*i]) | (data[4*i+1] << 8) | (data[4*i+2] << 16) | (data[4*i+3] << 24);


uint32_t t; // Временная переменная для вычислений
// 80 шагов преобразования для левой и правой ветвей
for(int j = 0; j < 80; j++) {
    t = a;
    // Выбор функции и констант зависит от текущего раунда
    if(j < 16)
        t += F(b,c,d) + x[r[j]];
    else if(j < 32)
        t += G(b,c,d) + x[r[j]] + K[1];
    else if(j < 48)
        t += H(b,c,d) + x[r[j]] + K[2];
    else if(j < 64)
        t += I(b,c,d) + x[r[j]] + K[3];
    else
        t += J(b,c,d) + x[r[j]] + K[4];
    // Сдвиг результата и обновление регистров
    t = ROL(t,s[j]) + e;
    a = e; e = d; d = ROL(c,10); c = b; b = t;


    // Аналогичный процесс для правой ветви с другими функциями и перестановками
    t = aa;
    if(j < 16)
        t += J(bb,cc,dd) + x[rr[j]] + KK[4];
    else if(j < 32)
        t += I(bb,cc,dd) + x[rr[j]] + KK[3];
    else if(j < 48)
        t += H(bb,cc,dd) + x[rr[j]] + KK[2];
```

```
            else if(j < 64)
                t += G(bb,cc,dd) + x[rr[j]] + KK[1];
            else
                t += F(bb,cc,dd) + x[rr[j]];
            t = ROL(t,ss[j]) + ee;
            aa = ee; ee = dd; dd = ROL(cc,10); cc = bb; bb = t;
        }

        // Обновляем состояние хеша, добавляя результаты обеих ветвей
        t = state[1] + c + dd;
        state[1] = state[2] + d + ee;
        state[2] = state[3] + e + aa;
        state[3] = state[4] + a + bb;
        state[4] = state[0] + b + cc;
        state[0] = t;

        // Очищаем временные данные для безопасности
        memset(x, 0, sizeof(x));
    }

public:
    // Конструктор класса, инициализирует начальные значения состояния
    RIPEMD160() {
        state[0] = 0x67452301;
        state[1] = 0xEFCDAB89;
        state[2] = 0x98BADCFE;
        state[3] = 0x10325476;
        state[4] = 0xC3D2E1F0;
        count = 0;
```

```
}

// Добавление данных для хеширования
void update(const unsigned char* data, size_t len) {
    size_t i;
    size_t index = count % 64; // Текущая позиция в буфере
    count += len; // Увеличиваем счетчик обработанных байтов

    // Заполняем буфер данными и обрабатываем, если набирается 64 байта
    for (i = 0; i < len; i++) {
        buffer[index++] = data[i];
        if (index == 64) {
            transform(buffer);
            index = 0;
        }
    }
}

// Завершение хеширования и формирование итогового хеша
string final() {
    unsigned char finalcount[8];
    // Записываем длину сообщения в байтах (little-endian)
    for (int i = 0; i < 8; i++)
        finalcount[i] = static_cast<unsigned char>((count >> (i * 8)) & 255);

    // Добавляем padding: бит 1 и нули до нужной длины
    update((unsigned char*)"\200", 1);
    while ((count % 64) != 56)
        update((unsigned char*)"\0", 1);
```

```
        update(finalcount, 8); // Добавляем длину сообщения

        // Формируем итоговый хеш из состояния
        unsigned char digest[20];
        for (int i = 0; i < 20; i++) {
            digest[i] = static_cast<unsigned char>(state[i >> 2] >> ((i & 3) << 3));
        }

        // Преобразуем хеш в строку в шестнадцатеричном формате
        string result;
        result.reserve(40);
        for (unsigned char i : digest) {
            char buf[3];
            sprintf(buf, "%02x", i);
            result += buf;
        }
        return result;
    }

    // Удобный метод для хеширования строки
    string hash(const string& input) {
        update((unsigned char*)input.c_str(), input.length());
        return final();
    }
};

// Поиск длины совпадающего префикса двух строк
size_t find_max_matching_length(const string& str1, const string& str2) {
    size_t max_common_prefix_length = 0;
```

```
    // Сравниваем символы двух строк до первого различия
    for (size_t j = 0; j < str1.length(); ++j) {
        if (str1[j] == str2[j]) {
            max_common_prefix_length++;
        } else {
            break;
        }
    }
    return max_common_prefix_length;
}

// Генерация случайной строки заданной длины
string generate_random_string(size_t length) {
                                                static const char charset[] =
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
    random_device rd; // Источник случайных чисел
    mt19937 gen(rd()); // Генератор случайных чисел
    uniform_int_distribution<> dist(0, sizeof(charset) - 2);

    string str(length, 0);
    // Заполняем строку случайными символами из заданного набора
    for(size_t i = 0; i < length; ++i) {
        str[i] = charset[dist(gen)];
    }
    return str;
}

// Тест 1: Проверка различий хешей при изменении строк
void test_string_differences() {
```

```
ofstream out("hash_differences.csv");
out << "Num Differing Characters,Max Matching Hash Length\n";

const int variations[] = {1, 2, 4, 8, 16}; // Количество изменяемых символов
RIPEMD160 hasher;

// Проверяем для разных количеств изменений
for(int diff : variations) {
    size_t max_matching = 0;
    for(int i = 0; i < 1000; ++i) {
        string base = generate_random_string(128); // Базовая строка
        string modified = base; // Копия для изменений

        random_device rd;
        mt19937 gen(rd());
        uniform_int_distribution<> dist(0, 127);

        // Вносим заданное количество изменений
        for(int j = 0; j < diff; ++j) {
            int pos = dist(gen);
            modified[pos] = modified[pos] == 'A' ? 'B' : 'A';
        }

        // Вычисляем хеши и сравниваем их
        string hash1 = hasher.hash(base);
        string hash2 = hasher.hash(modified);
        max_matching = max(max_matching, find_max_matching_length(hash1, hash2));
    }
```

```
        out << diff << "," << max_matching << "\n";
    }
    out.close();
}

// Тест 2: Поиск коллизий хешей
void test_hash_collisions() {
    ofstream out("hash_collisions.csv");
    out << "N,Duplicates\n";

    RIPEMD160 hasher;
    // Проверяем для разного количества строк (10^2 до 10^6)
    for(int i = 2; i <= 6; ++i) {
        int N = static_cast<int>(pow(10, i));
        unordered_set<string> unique_hashes; // Множество уникальных хешей

        // Генерируем строки и хешируем их
        for(int j = 0; j < N; ++j) {
            string input = generate_random_string(256);
            unique_hashes.insert(hasher.hash(input));
        }

        // Считаем количество коллизий
        int duplicates = N - unique_hashes.size();
        out << N << "," << duplicates << "\n";
    }
    out.close();
}
```

```
// Тест 3: Измерение скорости хеширования
void test_hash_speed() {
    ofstream out("hash_speed.csv");
    out << "String Length,Average Time (ms)\n";

    RIPEMD160 hasher;
    // Проверяем для строк разной длины (2^6 до 2^13)
    for(int i = 6; i <= 13; ++i) {
        auto length = static_cast<size_t>(pow(2, i));
        double total_time = 0;

        // Замеряем время для 1000 хешированний
        for(int j = 0; j < 1000; ++j) {
            string input = generate_random_string(length);

            auto start = chrono::high_resolution_clock::now();
            hasher.hash(input);
            auto end = chrono::high_resolution_clock::now();

            total_time += chrono::duration<double, milli>(end - start).count();
        }

        // Вычисляем среднее время
        double avg_time = total_time / 1000.0;
        out << length << "," << fixed << setprecision(6) << avg_time << "\n";
    }
    out.close();
}
```

```
// Главная функция программы
int main() {
    cout << "\n=== Running RIPEMD-160 Hash Tests ===\n\n";

    // Запускаем тесты и выводим сообщения о прогрессе
    cout << "1. Testing string differences...\n";
    test_string_differences();
    cout << "   > Test 1 completed successfully\n\n";

    cout << "2. Testing hash collisions...\n";
    test_hash_collisions();
    cout << "   > Test 2 completed successfully\n\n";

    cout << "3. Testing hash performance...\n";
    test_hash_speed();
    cout << "   > Test 3 completed successfully\n\n";

    // Итоговое сообщение и информация о сохраненных результатах
    cout << "\n=== All tests completed successfully ===\n";
    cout << "Results saved to:\n";
    cout << "- hash_differences.csv\n";
    cout << "- hash_collisions.csv\n";
    cout << "- hash_speed.csv\n\n";
    return 0;
}
```

Программа реализует алгоритм хеширования RIPEMD-160 и проводит три теста для оценки его свойств.

Основные функции программы:

- Хеширование данных: вычисление 160-битного хеша для входных строк с использованием класса RIPEMD160.
- Тест различий: сравнение хешей строк с разным количеством изменений (1, 2, 4, 8, 16 символов) для проверки лавинного эффекта, результаты записываются в hash_differences.csv.
- Тест коллизий: генерация $10^2 - 10^6$ случайных строк и поиск совпадений хешей, результаты сохраняются в hash_collisions.csv.
- Тест скорости: измерение времени хеширования строк длиной $2^6 - 2^{13}$ байт, результаты записываются в hash_speed.csv.

Реализация алгоритма включает ключевые особенности для обеспечения корректности и безопасности.

Специфические элементы реализации:

- Класс RIPEMD160: использует параллельные ветви обработки (левая и правая) с различными логическими функциями (F, G, H, I, J), константами (K, K'), перестановками слов (r, r') и сдвигами (s, s').
- Обработка блоков: данные дополняются до длины, кратной 512 битам, с добавлением бита '1', нулей и длины сообщения в формате little-endian.
- Безопасность: временные массивы очищаются после использования (например, memset(x, 0, sizeof(x))), чтобы предотвратить утечку данных.
- Тестирование: используются генерация случайных строк через random_device и mt19937, а также высокоточный таймер chrono для замера времени.

Выводы

В ходе выполнения лабораторной работы был реализован алгоритм хеширования RIPEMD-160 в виде класса RIPEMD160 с методами для хеширования входных данных, обработки блоков и формирования итогового хэша. Проведены три эксперимента для оценки свойств алгоритма:

- Тест различий хешей: сгенерировано 1000 пар строк длиной 128 символов, отличающихся на 1, 2, 4, 8 и 16 символов. Для каждой пары вычислены хеши, определена максимальная длина одинаковой последовательности символов в хешах. Результаты представлены в таблице 1 и на графике (рис. 3), где по оси абсцисс — количество отличий, а по оси ординат — максимальная длина одинаковой последовательности. Максимальная длина составила 2 символа для всех случаев, что подтверждает лавинный эффект алгоритма.
- Тест коллизий: выполнено $N = 10^i$, (i от 2 до 6) генераций хешей для случайных строк длиной 256 символов с поиском одинаковых хешей. Результаты записаны в таблицу 2, где первая колонка — количество генераций N, вторая — количество одинаковых хешей. Коллизии не обнаружены даже при $N = 10^6$, что соответствует теоретической устойчивости RIPEMD-160 к коллизиям (2^{80} операций).
- Тест скорости: проведено 1000 генераций хешей для строк длиной $N = 2^i$, (i от 6 до 13), измерено среднее время хеширования. Результаты представлены в таблице 3 и на графике (рис. 4), где по оси абсцисс — длина строки, по оси ординат — среднее время в миллисекундах. Время хеширования растет линейно с увеличением длины строки: от 0.010172 мс для $N = 64$ до 0.313664 мс для $N = 8192$.

Количество отличающихся символов	Максимальная длина последовательности одинаковых символов
1	**2**
2	**2**
4	**2**
8	**2**
16	**2**

Таблица 1 – Результаты теста различий хешей

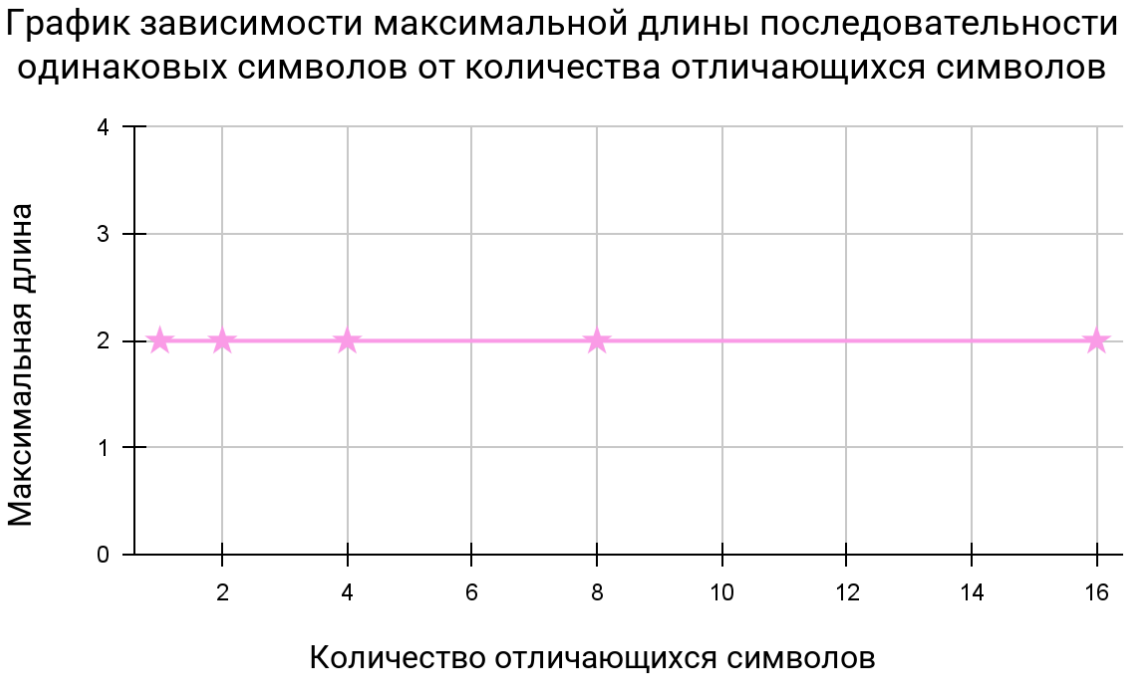

Рисунок 3 – График зависимости максимальной длины последовательности одинаковых символов от количества отличающихся символов

Количество генераций **N**	Количество одинаковых хешей
100	**0**
1000	**0**
10000	**0**
100000	**0**
1000000	**0**

Таблица 2 – Результаты теста на коллизии

Длина строки	Среднее время (мс)
64	0,010172
128	0,012798
256	0,017627
512	0,02503
1024	0,054008
2048	0,083617
4096	0,159004
8192	0,313664

Таблица 3 – Результаты теста скорости

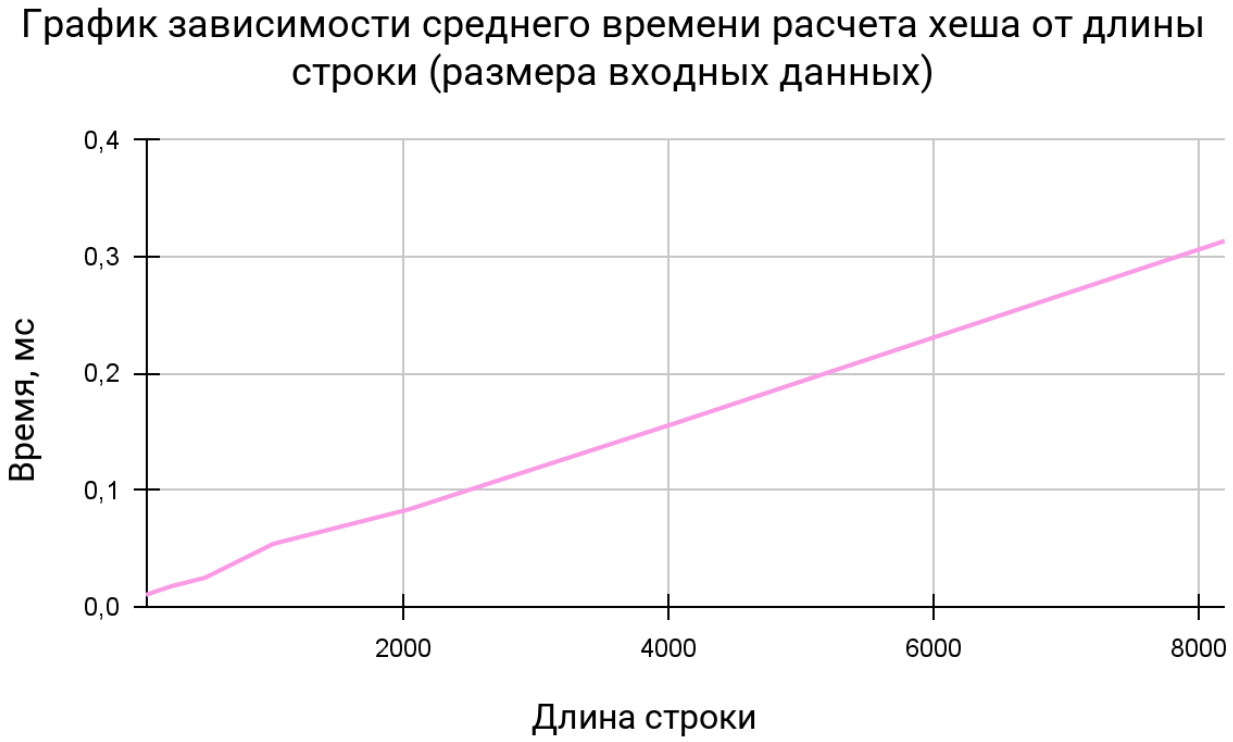

Рисунок 4 – График зависимости среднего времени расчета хеша от длины строки (размера входных данных)

RIPEMD-160 демонстрирует отличные криптографические свойства: даже изменение одного символа во входной строке приводит к значительным изменениям в хеше (максимальная длина одинаковой последовательности — всего 2 символа из 40), что делает алгоритм устойчивым к атакам, основанным на предсказании хеша.

Отсутствие коллизий при 10^6 генерациях подтверждает высокую криптографическую стойкость алгоритма. Для практического применения это означает, что RIPEMD-160 может быть использован в системах, где требуется защита от коллизий, например, в цифровых подписях.

Алгоритм показывает линейную зависимость времени обработки от длины входных данных, что делает его предсказуемым по производительности. Для строк длиной до 8192 байт время хеширования остается приемлемым (менее 0.32 мс), однако для больших данных (например, файлов) производительность может стать ограничивающим фактором по сравнению с более быстрыми алгоритмами, такими как MD5.