МОЛДАВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ МАТЕМАТИКИ И ИНФОРМАТИКИ

ДЕПАРТАМЕНТ ИНФОРМАТИКИ

Отчет по дисциплине:

“Криптография и безопасность информации”

**Лабораторная работа №2**  
**Алгоритмы хеширования**

**Автор:** Калинкова С.

группа I2302

**Проверила:** Чербу О.

Кишинев, 2024

**Алгоритм хеширования HAVAL**

Алгоритм хеширования HAVAL (Hash Algorithm for Variable Length) — это криптографический хеш-алгоритм, разработанный в 1992 году Юэем Женем и Джин-Жиа Чиеном. Он отличается от большинства других хеш-функций своей высокой гибкостью и настраиваемостью. HAVAL предоставляет возможность выбора длины итогового хеша и числа раундов обработки, что делает его уникальным среди других алгоритмов хеширования.

## **1. История и особенности HAVAL**

HAVAL был представлен в 1992 году как усовершенствованная и настраиваемая хеш-функция. Основная идея заключалась в том, чтобы предоставить пользователям возможность самостоятельно выбирать параметры безопасности: длину итогового хеша (от 128 до 256 бит) и количество раундов обработки (3, 4 или 5 раундов). Это делает HAVAL гибким инструментом, который можно настроить в зависимости от требуемого уровня безопасности и скорости работы.

### **Основные особенности HAVAL:**

* **Переменная длина хеша**: Вы можете выбрать длину хеша среди 128, 160, 192, 224 и 256 бит.
* **Переменное количество раундов**: HAVAL поддерживает 3, 4 или 5 раундов обработки.
* **Большая производительность**: HAVAL работает быстрее большинства других алгоритмов хеширования того времени (например, MD5 или SHA-1) при аналогичной длине хеша.

## **2. Структура и этапы работы HAVAL**

HAVAL разделяет входные данные на блоки по 1024 бита (128 байт) и выполняет последовательные раунды обработки для каждого блока. В зависимости от выбранного количества раундов, алгоритм применяет различные нелинейные функции для трансформации данных.

### **2.1 Инициализация**

Перед началом работы, HAVAL инициализирует восемь переменных состояния, которые будут изменяться в ходе обработки. Эти начальные переменные известны как H0, H1, H2, H3, H4, H5, H6 и H7. Они задаются конкретными константами, отличными для каждой хеш-функции.

### **2.2 Этапы обработки**

Каждый раунд обработки состоит из четырех основных операций:

* **Дополнение данных**: Исходные данные дополняются до длины, кратной 1024 битам. Дополнение включает специальный маркер и запись длины сообщения в последние 8 байт.
* **Разделение на блоки**: Входные данные делятся на блоки по 1024 бита (128 байт).
* **Раунды преобразований**: Каждый блок данных обрабатывается последовательно через серию нелинейных функций, зависящих от количества раундов (3, 4 или 5). Эти функции используют комбинацию битовых операций и циклических сдвигов для запутывания данных.
* **Финализация**: После обработки всех блоков, результаты каждого раунда объединяются для формирования итогового хеша. Финальные 128, 160, 192, 224 или 256 бит извлекаются из переменных состояния.

### **2.3 Финальные значения**

Результат хеширования зависит от того, какую длину хеша вы выбрали. HAVAL завершает работу с преобразованием состояний H0–H7 в строку фиксированной длины.

## **3. Переменная длина хеша и количество раундов**

Одна из уникальных особенностей HAVAL — возможность выбора длины хеша. Количество раундов напрямую влияет на устойчивость хеш-функции к атакам. Больше раундов делает хеш-функцию более устойчивой, но также замедляет ее выполнение. Вот некоторые ключевые моменты:

* **3 раунда**: Обеспечивают высокую скорость, но меньше устойчивости к атакам.
* **4 раунда**: Наиболее сбалансированный вариант между скоростью и безопасностью.
* **5 раундов**: Максимальная устойчивость к криптоаналитическим атакам, но сниженная скорость работы.

Таким образом, пользователи могут настроить HAVAL под конкретные потребности.

## **4. Примеры использования HAVAL**

HAVAL можно использовать во множестве приложений, где требуется криптографическая устойчивость. Вот несколько примеров:

### **4.1 Проверка целостности данных**

Благодаря своим свойствам устойчивости к коллизиям и высокой скорости работы, HAVAL подходит для использования в системах проверки целостности данных. Например, HAVAL может использоваться для создания цифровых отпечатков файлов и обнаружения изменений в них.

### 

### **4.2 Электронная подпись**

HAVAL может использоваться в системах цифровой подписи, где важно, чтобы небольшой хеш представлял большой объем данных.

### **4.3 Хеширование паролей**

HAVAL может использоваться для безопасного хеширования паролей в системах аутентификации. Благодаря возможности выбора длины хеша и количества раундов, HAVAL предоставляет гибкие решения для различных уровней безопасности.

## **5. Сравнение с другими хеш-алгоритмами**

### **5.1 Преимущества HAVAL:**

* Настраиваемая длина хеша и количество раундов.
* Высокая скорость работы при использовании 3 или 4 раундов.
* Гибкость в зависимости от требований к безопасности.

### **5.2 Недостатки HAVAL:**

* Более сложная реализация по сравнению с MD5 или SHA-1.
* Меньше распространен, чем другие хеш-функции, такие как SHA-256.

## **6. Уязвимости и атаки на HAVAL**

Хотя HAVAL предоставляет высокий уровень гибкости и безопасности, он не лишен уязвимостей. Криптоаналитики обнаружили некоторые коллизии в версиях HAVAL с 3 и 4 раундами, что делает их менее надежными для долгосрочного использования. Однако версия HAVAL с 5 раундами остается устойчивой к современным атакам, и её использование рекомендуется для критически важных приложений.

## **7. Разбор кода**

#### **1. Константы, используемые в HAVAL**

H0 = 0x243F6A88

H1 = 0x85A308D3

H2 = 0x13198A2E

H3 = 0x03707344

H4 = 0xA4093822

H5 = 0x299F31D0

H6 = 0x082EFA98

H7 = 0xEC4E6C89

Это начальные константы, которые инициализируют внутреннее состояние алгоритма. Они соответствуют начальным значениям, используемым в различных версиях HAVAL.

#### **2. Функция раунда (пример для одного раунда)**

def f1(x6, x5, x4, x3, x2, x1, x0):

return (x1 & x0) | (~x1 & x2)

Эта функция является примером одного из раундов обработки данных в HAVAL. Она использует логические операции "AND", "OR" и "NOT" для преобразования входных данных. В реальной реализации HAVAL используется несколько таких функций для разных раундов.

#### **3. Обработка блока данных (haval\_pass)**

def haval\_pass(data\_block, state):

# Проверяем размер блока перед разбиением

print(f"Processing block of size {len(data\_block)} bytes")

# Убеждаемся, что блок имеет длину ровно 32 байта

if len(data\_block) != 32:

raise ValueError(f"Expected block size of 32 bytes, but got {len(data\_block)} bytes")

x = struct.unpack('<8I', data\_block) # Разделение блока на 32-битные части

state[7] ^= f1(state[0], state[1], state[2], state[3], state[4], state[5], state[6])

return state

* **Проверка размера блока**: HAVAL обрабатывает данные блоками по 32 байта (256 бит). Если размер блока отличается, генерируется ошибка.
* **Разбиение на 32-битные части**: Входные данные разбиваются на 8 частей по 32 бита с помощью struct.unpack().
* **Функция раунда**: Для текущего состояния применяется одна из раундовых функций (здесь только пример f1), и результат XOR'ится с одной из переменных состояния.

#### **4. Основная функция HAVAL**

def haval(data):

# Инициализация состояния

state = [H0, H1, H2, H3, H4, H5, H6, H7]

# Дополнение данных до нужной длины

padded\_data = pad\_data(data)

# Разделение на блоки и обработка каждого блока

for i in range(0, len(padded\_data), 32):

block = padded\_data[i:i + 32]

state = haval\_pass(block, state)

# Финализация хэша

return finalize\_hash(state)

* **Инициализация состояния**: Начальные переменные состояния устанавливаются в соответствии с константами H0 — H7.
* **Дополнение данных**: Исходные данные дополняются до нужной длины с помощью функции pad\_data().
* **Обработка блоков**: Все блоки данных по 32 байта обрабатываются функцией haval\_pass(), которая обновляет внутреннее состояние.
* **Финализация**: После обработки всех блоков вызывается функция для формирования итогового хеша.

#### **5. Функция дополнения данных (pad\_data)**

def pad\_data(data):

original\_length = len(data)

# Добавляем бит завершения (0x80) к данным

data += b'\x80'

# Дополняем нулями, чтобы итоговая длина стала кратной 32 байтам

# Оставляем место для 8 байт, которые будут использоваться для хранения длины

while (len(data) + 8) % 32 != 0:

data += b'\x00'

# Добавляем длину исходного сообщения (в битах) в последних 8 байтах

data += struct.pack('<Q', original\_length \* 8)

print(f"Padded data length: {len(data)} bytes")

return data

* **Добавление бита завершения**: К данным добавляется специальный байт (0x80) для обозначения конца исходных данных.
* **Дополнение нулями**: После этого данные дополняются нулями, чтобы общая длина данных была кратна 32 байтам, за исключением 8 последних байт, которые будут использоваться для хранения длины исходного сообщения.
* **Добавление длины сообщения**: В последние 8 байт записывается длина исходного сообщения (в битах).

#### **6. Финализация хеша (finalize\_hash)**

def finalize\_hash(state):

return ''.join([f'{x:08x}' for x in state])

Функция преобразует каждую переменную состояния в строку шестнадцатеричных символов. Итоговый хеш — это объединение этих строк в одно значение.

#### **7. Тестирование (main)**

data = b"Hello, world!"

hash\_result = haval(data)

print(f"HAVAL hash: {hash\_result}")

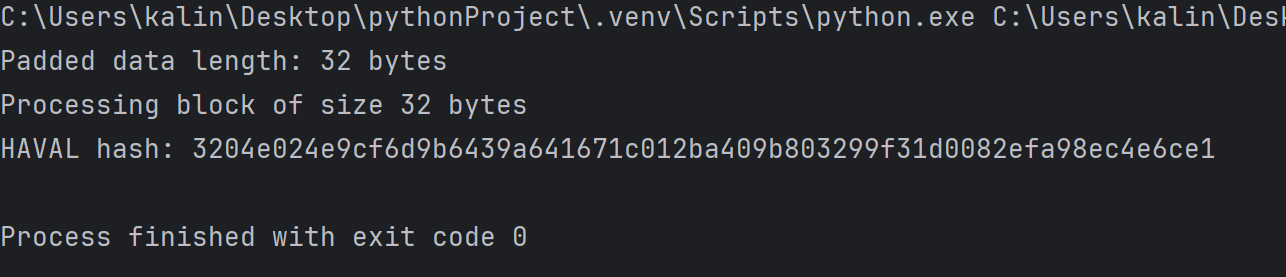
* **Тестовые данные**: В данном примере хешируется строка Hello, world!.
* **Вызов HAVAL**: Данные передаются в основную функцию haval(), которая возвращает хеш.
* **Вывод**: Итоговый хеш выводится в консоль.

### **Итог:**

Этот код выполняет следующие шаги:

1. Инициализирует состояние с константами.
2. Дополняет данные до нужного размера.
3. Разбивает данные на блоки по 32 байта и применяет функцию раунда для каждого блока.
4. Формирует окончательное хеш-значение, преобразуя переменные состояния в шестнадцатеричное представление.

Этот упрощённый пример не реализует все раунды и функции HAVAL, но даёт представление о базовых принципах работы алгоритма.



## **8. Заключение**

HAVAL — это мощная и гибкая хеш-функция, предоставляющая возможность выбора длины итогового хеша и количества раундов обработки, что делает её настраиваемой для различных нужд. Его конструкция обеспечивает высокую производительность и возможность адаптации к конкретным требованиям безопасности. Хотя современные алгоритмы, такие как SHA-256, получили более широкое распространение, HAVAL остаётся актуальным выбором для тех, кто ищет баланс между эффективностью, скоростью работы и гибкостью в настройках хеширования.