МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное

учреждение высшего образования

«Ярославский государственный университет им. П.Г. Демидова»

Кафедра компьютерной безопасности и математических методов обработки информации

Курсовая работа

**Некоторые вопросы защиты информации при использовании технологии распределенного реестра (блокчейн).**

Научный руководитель

канд. физ.-мат. наук, доцент

\_\_\_\_\_\_\_\_ Д.М. Мурин

«\_\_» \_\_\_\_\_\_\_\_\_ 2018 г.

Студент группы КБ 41–CО

\_\_\_\_\_\_\_\_А.А. Шарунцова

«\_\_» \_\_\_\_\_\_\_\_\_ 2018 г.

Ярославль 2018 г.

**Реферат**

Объем 37 с., 5 гл., 16 рис., 5 табл., 7 источников, 3 прил.

**Хеш-функции. Алгоритмы хеш-функций BLAKE и BLAKE2. Хеш-функции в блокчейне. Анализ**

Объектом исследования является хеш-функция BLAKE и ее модификации, применение этой хеш-функции в блокчейнах и поиск оптимальных параметров для применения ее в блокчейнах.

Цель работы – провести экспериментальное исследование зависимости времени, затрачиваемого на поиск значения хеш-функции с нулевым префиксом определенной длины от длины префикса в условиях выбора входа хеш-функции по определенным правилам.

Задачи:

* Обзор литературы.
* Реализация хеш-функции.
* Проведение эксперимента.
* Составление отчета по эксперименту.

Краткий теоретический обзор:

В данной работе содержатся сведения о проведении Национальным институтом стандартов и технологий США конкурса на новую хеш-функцию SHA-3. Далее, в главе 1 приведено подробное описание и алгоритм одного из финалистов конкурса, хеш-функции BLAKE. В главе 2 дано описание модификации данной хеш-функции, BLAKE2, а также основные ее отличия от своего предшественника. Затем, в главе 3 производится обзор безопасности и скорости работы хеш-функции BLAKE2 на основании официальных исследований в рамках конкурса.

В главе 4 описаны теоретические понятия, связанные с технологией блокчейна. В ней же дается описание работы программы, в которой реализован алгоритм хеш-функции BLAKE2. Ниже идут результаты эксперимента над работой данной программы с различными входными данными, а также соответствующие выводы к ним.

**Оглавление**

[Введение 5](#_Toc515622283)

[1. BLAKE 6](#_Toc515622284)

[1.1 Основные характеристики алгоритма BLAKE 6](#_Toc515622285)

[1.1.1 Основные принципы реализации 6](#_Toc515622286)

[1.1.2 Преимущества и недостатки 7](#_Toc515622287)

[1.2 Алгоритм BLAKE 8](#_Toc515622288)

[1.2.1 Краткое описание алгоритма 8](#_Toc515622289)

[1.2.2 Входные данные и константы 9](#_Toc515622290)

[1.2.3 Функция сжатия: 10](#_Toc515622291)

[1.2.4. Инициализация 10](#_Toc515622292)

[1.2.5 Раундовая функция 10](#_Toc515622293)

[1.2.6 Заключительный этап 12](#_Toc515622294)

[1.2.7 Хеширование сообщения 12](#_Toc515622295)

[2. BLAKE2b 14](#_Toc515622296)

[2.1 Разновидности алгоритма BLAKE2 14](#_Toc515622297)

[2.2 Основные отличия алгоритма BLAKE2 от BLAKE 14](#_Toc515622298)

[2.2.1 Меньшее количество раундов. 14](#_Toc515622299)

[2.2.2 Оптимизированные константы сдвига. 14](#_Toc515622300)

[2.2.3 Минимальное дополнение и финализационные флаги. 15](#_Toc515622301)

[2.2.4 Меньшее количество констант. 15](#_Toc515622302)

[2.2.5 Порядок бит от старшего к младшему. 16](#_Toc515622303)

[2.2.6 Счетчик байтов. 16](#_Toc515622304)

[2.2.7 Использование соли в хешировании. 17](#_Toc515622305)

[2.2.8 Блок-параметр. 17](#_Toc515622306)

[2.2.9 Дерево хешей. 18](#_Toc515622307)

[2.3 Механизм хеширования посредством хеш-дерева. 18](#_Toc515622308)

[3. Безопасность и скорость работы BLAKE2 19](#_Toc515622309)

[3.1 Безопасность 19](#_Toc515622310)

[3.2 Скорость 20](#_Toc515622311)

[4. Хеш-функция в блокчейне 22](#_Toc515622312)

[4.1 Применения хеш-функций 22](#_Toc515622313)

[4.2 Принцип работы блокчейна 22](#_Toc515622314)

[4.2.1 Основной механизм 22](#_Toc515622315)

[4.2.2 Доказательство работы (англ. «Proof of Work» – PoW) 23](#_Toc515622316)

[4.3 Моделирование работы хеш-функции в блокчейне 24](#_Toc515622317)

[Заключение 30](#_Toc515622318)

[Список литературы. 31](#_Toc515622319)

[Приложение А 32](#_Toc515622320)

[Приложение Б 33](#_Toc515622321)

[Приложение В 37](#_Toc515622322)

# Введение

Криптографическая хеш-функция – это отображение двоичной строки в двоичную строку фиксированной длины (хеш-сумму), следуя определенным параметрам безопасности. Алгоритмы хеширования можно использовать в формировании цифровых подписей, имитовставок, функциях формирования ключа, псевдослучайных функциях и других прикладных задачах информационной безопасности.

В 1993 Национальный институт стандартов и технологий США (НИСТ) опубликовал первый стандарт безопасного хеширования, алгоритм SHA-0, который два года спустя был вытеснен новой, улучшенной версией того же алгоритма, SHA-1. SHA-1 считался надежным алгоритмом вплоть до конца прошлого тысячелетия, когда исследования выявили, что SHA-1 и MD5 уязвимы к коллизионной атаке. Позже, в феврале 2005 года Ван Сяоюнь, Лиза Инь Ицюнь и Юй Хунбо предоставили алгоритм, производящий коллизионную атаку на SHA-1 менее чем за 269 операций. Тем временем, НИСТ представил новое семейство хеш-функций SHA-2, которые не были скомпрометированы до сих пор. Однако в 2006 году возникло предположение, что из-за роста мощности и производительности устройств, а также из-за появления новых методов криптоанализа в будущем надежность хеш-функции SHA-2 значительно. В связи с этим, 2 ноября 2007 года НИСТ объявил о конкурсе на новую криптографическую хеш-функцию, названную SHA-3, которая заменит в будущем стандарт SHA-2.

НИСТ выбрал 51 хеш-функцию в первый тур. 14 из них прошло во второй раунд, из которых было выбрано 5 финалистов.

В течение года институтом принимались дополнительные исследования всех заинтересованных сторон, посвященные исключительно пяти финалистам: BLAKE, Grostl, JH, Keccak и Skein. Создателям позволили вносить незначительные изменения в их алгоритмы вплоть до 16 января 2011 г. С этого времени алгоритмы считались полностью спроектированными и НИСТ выделил еще один год для их публичного рассмотрения. Победитель был объявлен в начале 2013 г. Им оказалась функция хеширования Keccak (автор Йоан Даймен).

Несмотря на то, что в качестве стандарта был выбран всего один алгоритм, все пять финалистов заслуживают особого внимания. Во-первых, потому, что в них предложены новые для криптографии подходы к построению современных функций хеширования. Во-вторых, потому что все пять финалистов являются "сильными" функциями и предположительно не уступают друг другу в стойкости.

Данная работа будет посвящена подробному изучению функции BLAKE, а также ее улучшенной версии BLAKE2.

# 1. BLAKE

### 1.1 Основные характеристики алгоритма BLAKE

Алгоритм BLAKE основан на ранее изученных компонентах. В связи с чем, можно выделить три основных аспекта внутренней реализации BLAKE:

1. В основе BLAKE лежит итеративный метод HAIFA, близкий по своей структуре к классическому, широко используемому методу построения криптографических хеш-функций Меркла-Дамгарда. Он обеспечивает устойчивость к атаке нахождения второго прообраза и поддерживает хеширования с использованием соли.
2. Внутренняя структура local wide-pipe обеспечивает защиту от коллизий.
3. Алгоритм сжатия BLAKE является модифицированной версией поточного шифра ChaCha, который был тщательно проанализирован. Кроме того, он является хорошо параллелизируемым.

1.1.1 Основные принципы реализации

Хеш-функция была сконструирована так, чтобы отвечать всем критериям, обозначенным НИСТ для SHA-3. Они включают в себя следующие критерии:

* Длина хеш-суммы может принимать значения 224, 256, 384 и 512 бит.
* Размер параметра такой же, как в SHA-2.
* Однопоточный режим передачи.
* Максимальная длина сообщения должна составлять как минимум 264-1.

Создатели алгоритма BLAKE добавили для своего алгоритма еще ряд требований:

* Поддержка хеширования с использованием соли. Это обеспечивает интерфейс с дополнительными входными данными во избежание использования небезопасных режимов и поддерживает идею использования рандомизированных алгоритмов хеширования.
* Поддержка параллелизма для мультиядерных или SIMD-процессоров. Это увеличивает скорость хеширования в несколько раз.
* Возможность адаптации к средам с меньшей пропускной способностью.

С другой стороны, они исключили достижение алгоритмом следующих целей:

* Наличие редукции к предположительно трудной задаче.
* Наличие гомоморфных и инкрементальных свойств.
* Наличие масштабируемого дизайна.
* Наличие спецификации для хеширования с переменной длиной хеша.

Для обоснования своего выбора разработчики приводят следующие доводы.

Те немногие преимущества, которыми обладают гомоморфные и инкрементальные хеш-функции, не покрывают их недостатков, и, что более важно, таковые свойства нежелательны в большинстве приложений. Масштабируемость дизайна в отношении различных размеров параметров не имеет реального преимущества на практике, а надежность масштабируемого дизайна трудно достижима. Наконец, представляется излишним усложнять функцию свойствами переменной длины хеша, поскольку пользователь может попросту урезать значения хеша для получения более короткого хеша, что может сказаться на стойкости хеш-функции.

Таким образом, целью разработчиков было создание максимально простой хеш-функции, в которой будут хорошо известные и надежные компоненты. Они постарались избежать избыточных функций, предоставив пользователю лишь то, в чем он действительно нуждается или будет иметь потребность в будущем (например, хеширование с солью).

1.1.2 Преимущества и недостатки

**Преимущества:**

* Простота алгоритма.
* Возможность хеширования с использованием соли.
* Быстродействие, как на аппаратном, так и на программном уровне.
* Параллелизм и адаптация к средам с меньшей пропускной способностью.
* Тщательно проанализированный компонент, лежащий в основе (алгоритм шифрования ChaCha).
* Устойчивость к общим атакам второго рода.
* Устойчивость к атакам по сторонним каналам.
* Устойчивость к атакам удлинением сообщения.

**Недостатки:**

* Максимальная длина сообщения 264 и 2128 для алгоритмов BLAKE-256 и BLAKE-512 соответственно.
* Устойчивость к мультиколлизионным атакам осуществляется таким же образом, что и в SHA-2.
* Нахождение фиксированных точек происходит за меньшее время, чем в идеальной функции.

### 1.2 Алгоритм BLAKE

Рассмотрим детально алгоритм BLAKE. Для дальнейшего сравнения его с алгоритмом BLAKE2 и выявления основных отличий рассмотрим самую «близкую» к нему версию алгоритма BLAKE – BLAKE-512.

Для начала введем основные обозначения.

Таблица 1

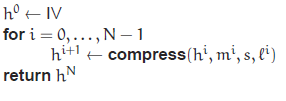
**Основные математические обозначения**

|  |  |
| --- | --- |
| **Символ** | **Значение** |
|  | Присваивание значения справа переменной слева |
|  | Сложение по модулю 232 или 264 |
|  | Сложение по модулю 2 |
|  | Сдвиг на k бит влево с добавлением первых k бит в конец |
|  | Сдвиг на k бит вправо с добавлением последних k бит в начало |
|  | Закодированное целое число l в k-битную последовательность |

1.2.1 Краткое описание алгоритма

1. На вход получаем сообщение M.
2. Дополнения исходного сообщения M до сообщения M' длиной кратной длине обрабатываемого блока (для BLAKE-512 это 1024).
3. Разбиение сообщения M' на блоки.
4. Инициализация вектора h0 с помощью инициализационного вектора IV (h0 ← IV).
5. Для каждого блока mi выполняем функцию сжатия, на вход которой подаются вектор h, блок сообщения mi, значение соли s и значение счетчика t.
   1. Инициализация текущего состояния V с использованием входных данных и констант.
   2. Запускается серия из n раундов (для BLAKE-512 n=16), где каждый раунд – это определенная операция над вектором V.
   3. Последним этапом вычисляем новое значение вектора h с использованием его старого значения, значения вектора состояния *V* и длины битов сообщения в блоке l.
6. Выходными данными будет новое значение вектора h.

Схематично процесс представлен на рис. 1.



**Рис. 1.** Схема алгоритма хеширования BLAKE

Далее рассмотрим каждый этап алгоритма более подробно.

1.2.2 Входные данные и константы

BLAKE-512 оперирует с 64-битными словами и возвращает 64-байтный хеш. Длина цепочки **h=h0,…,h7** – 512 бит, один блок сообщения – 1024 бит, соль – 256 бит, счётчик – 128 бит.

В алгоритме используются следующие константы:

1. Инициализационный вектор **IV** в BLAKE-512 такой же, как и в SHA-512. В таблице 2 представлены его значения.

Таблица 2

**Значения инициализационного вектора IV**

|  |  |
| --- | --- |
| IV0 = 6A09E667F3BCC908 | IV1 = B67AE8584CAA73B |
| IV2 = 3C6EF372FE94F82B | IV­3 = A54FF53A5F1D36F1 |
| IV4 = 510E527FADE682D1 | IV5 = 9B05688C2B3E6C1F |
| IV6 = F83D9ABFB41BD6B | IV7 = 5BE0CD19137E2179 |

1. BLAKE-512 использует также константы, указанные в таблице 3. Они образованы из первых цифр числа π.

Таблица 3

**Значения констант для алгоритма BLAKE**

|  |  |
| --- | --- |
| c0 = 243F6A8885A308D3  c1 = 13198A2E03707344  c2 = A4093822299F31D0  c3 = 082EFA98EC4E6C89  c4 = 452821E638D01377  c5 = BE5466CF34E90C6C  c6 = C0AC29B7C97C50DD  c7 = 3F84D5B5B5470917 | c8 = 9216D5D98979FB1B  c9 = D1310BA698DFB5AC  c10 = 2FFD72DBD01ADFB7  c11 = B8E1AFED6A267E96  c12 = BA7C9045F12C7F99  c13 = 24A19947B3916CF7  c14 = 0801F2E2858EFC16  c15 = 636920D871574E69 |

1. В алгоритме BLAKE-512 используются перестановки, указанные в таблице 4.

Таблица 4

**Перестановки, используемые в алгоритме BLAKE**

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

1.2.3 Функция сжатия:

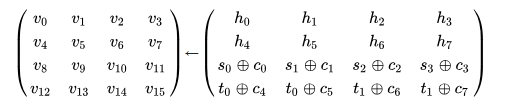
Функция сжатия алгоритма BLAKE-512 принимает на вход:

* Переменные цепочки **h = h0,...,h7** (8 слов);
* Блок сообщения **m = m0,...,m15**;
* Значение соли **s = s0,...,s3**;
* Значение счётчика **t = t0,t1**.

Таким образом, на вход ей подаётся 30 слов (8+16+4+2=30, 30\*64 = 1024 бит = 128 байт). Возвращает функция сжатия только новое значение переменных цепочки: **h' = h'0,...,h'7**. В дальнейшем будем обозначать **h'=compress(h, m, s, t).**

1.2.4. Инициализация

16 переменных, **v0,...,v15**, описывающих текущее состояние **v**, инициализируются начальными значениями в зависимости от входных данных и представлены в виде матрицы **4×4** на рис. 2:



**Рис. 2** Заполнение матрицы текущего состояния *v*

1.2.5 Раундовая функция

После того, как v инициализировано, запускается серия из 16 раундов. Раунд — это операция над состоянием v, которая производит вычисления, разбитые на блоки, указанные в таблице 5:

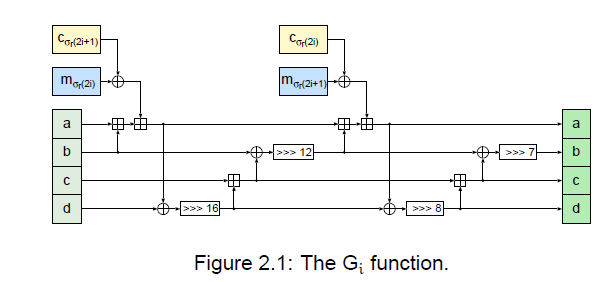
Таблица 5

**Раундовая фунция G**

|  |  |
| --- | --- |
| G0(v0, v4, v8, v12) | G4(v0, v5, v10, v15) |
| G1(v1, v5, v9, v13) | G5(v1, v6, v11, v12) |
| G2(v2, v6, v10, v14) | G6(v2, v7, v8, v13) |
| G3(v3, v7, v11, v15) | G7(v3, v4, v9, v14) |

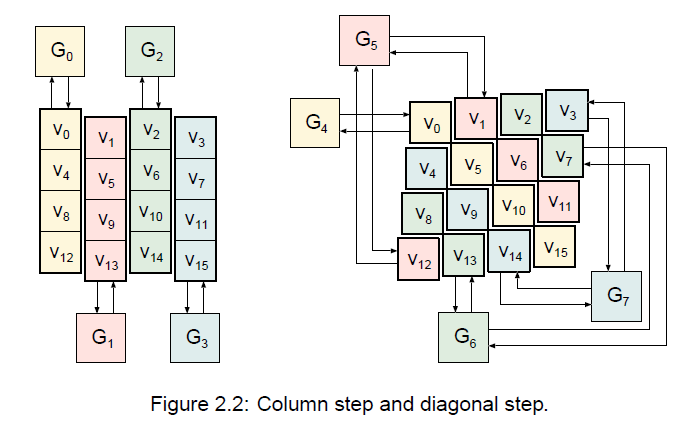
На r-ом раунде блок вычислений Gi(a, b, c, d) работает следующим образом:

На рисунке 3 работа функции **G** представлена в виде схемы.



**Рис. 3** Графическая иллюстрация работы блока вычислений Gi

Первые четыре блока **G0,...,G3** могут вычисляться параллельно, так как каждый изменяет свою определённую колонку переменных матрицы состояний. Назовём процедуру вычисления **G0,...,G3 column step**. Точно также могут быть параллельно вычислены G4,...,G7, но они в свою очередь изменяют каждый свою *диагональ* матрицы состояния v. Поэтому назовём процедуру вычисления **G4,...,G7** **diagonal step**. Возможность параллельного вычисления **Gi** представлена графически на рисунке 4.

На раундах, для номеров **r** больших 9, используется перестановка **σr%10**, например, на 13-том раунде используется **σ3**.

**Рис**.**4** Иллюстрация процесса распараллеливания выполнения функций Gi. Процедуры *column step* и *diagonal step*

1.2.6 Заключительный этап

После всех раундов новое значение переменных цепочки h'0,...,h'7 вычисляется из переменных *v­0,…,v15* матрицы состояния, входных переменных h и из соли s:

*h'0 ← h0 ⊕ s0 ⊕ v8*

*h'1 ← h1 ⊕ s1 ⊕ v9*

*h'2 ← h2 ⊕ s2 ⊕ v10*

*h'3 ← h3 ⊕ s3 ⊕ v11*

*h'4 ← h4 ⊕ s4 ⊕ v12*

*h'5 ← h5 ⊕ s5 ⊕ v13*

*h'6 ← h6 ⊕ s6 ⊕ v14*

*h'7 ← h7 ⊕ s7 ⊕ v15*

1.2.7 Хеширование сообщения

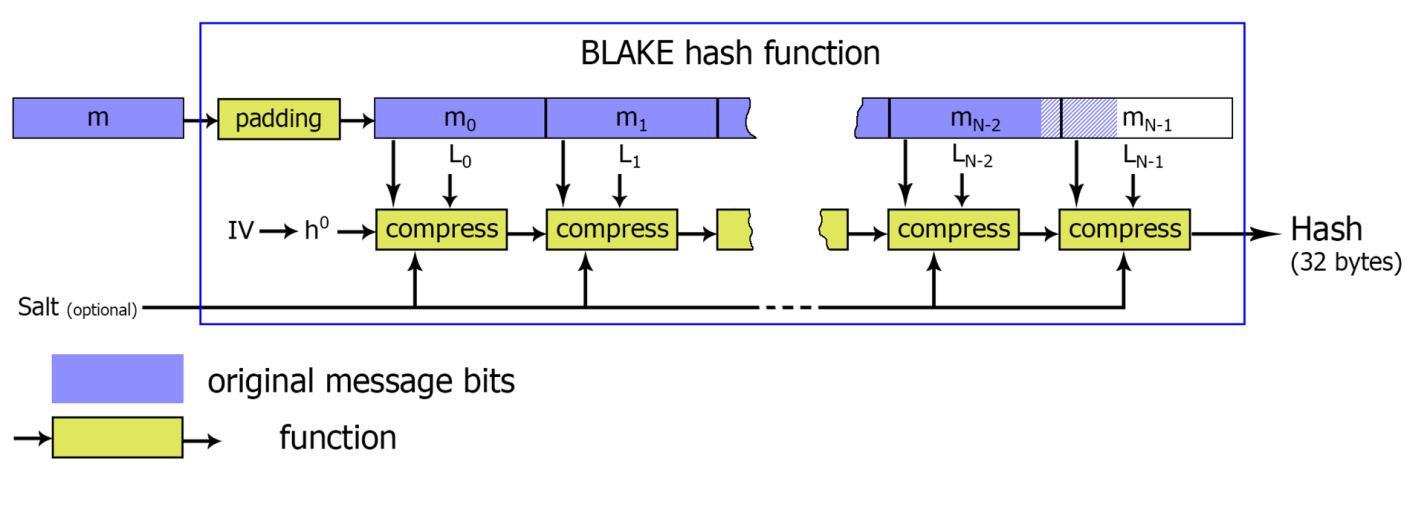
Опишем процесс хеширования сообщения **m** длиной **l<264** бит. Сначала сообщение дополняется функцией **padding** данными для кратности 1024 битам (128 байтам), затем, блок за блоком, его обрабатывает функция сжатия **compression function**.

Для BLAKE-512 дополнение до блока действует следующим образом: добавляем бит 1, затем добавляем столько нулей, сколько нужно, чтобы битовая длина сообщения была сравнима с 895 по модулю 1024. Затем снова добавляем бит 1, после чего 128-битовое беззнаковое представление битовой длины сообщения в порядке от старшего бита к младшему:

Такая процедура гарантирует, что длина дополненного сообщения будет кратна 1024.

Дополненное сообщение разбивается на блоки, каждый из которых состоит из 16 слов **m0,…,m15.** Пусть **Li** — количество бит исходного сообщения в блоках **m0,...,mi**, то есть исключая те биты, которые были добавлены в процедуре padding. Значение соли определяется пользователем или задаётся равным 0, если её не нужно использовать (**s0=s1=s2=s3=0**). Хеш-сумма сообщения таким образом вычисляется:

Процесс хеширования представлен наглядно на рисунке 5:



**Рис**.5 Блок-схема процесса хеширования.

# 2. BLAKE2b

### 2.1 Разновидности алгоритма BLAKE2

Blake2 имеет две модификации:

1. Алгоритм BLAKE2b (под названием BLAKE2 подразумевается именно эта модификация) оптимизирован для 64-разрядных платформ и способен выдавать хеш-сумму от 1 до 64 байт.
2. Алгоритм BLAKE2s оптимизирован для платформ от 8 до 32 разрядов и способен выдавать хеш-сумму от 1 до 32 байт.

Оба алгоритма считаются стойкими и имеют хорошее представление для любых платформ, программного и аппаратного обеспечения. Каждый из них подходит для любого ЦПУ, но работает вдвое быстрее на том ЦПУ, под который он оптимизирован. Например, на процессоре Tegra (SoC-платформа с 32-битной архитектурой ARMv7) алгоритм BLAKE2s должен работать вдвое быстрее, чем BLAKE2b, в то время как на процессоре AMD A10-5800K (с 64-разрядной архитектурой Piledriver) BLAKE2b должен работать в полтора раза быстрее, чем BLAKE2s.

Поскольку алгоритм BLAKE2 очень похож на своего предшественника, алгоритм BLAKE, достаточно будет рассмотреть ключевые различия между ними.

### 2.2 Основные отличия алгоритма BLAKE2 от BLAKE

2.2.1 Меньшее количество раундов.

BLAKE2b делает 12 раундов, BLAKE2s – 10 раундов вместо 16 и 14 для BLAKE. На основании исследований и предположений, какие технологии возможны в будущем, маловероятно, что 16 и 14 раундов обеспечат существенно большую безопасность, нежели 12 и 10 раундов. Однако такие изменения дают выигрыш в быстродействии алгоритма. На больших данных они обеспечивают прирост в скорости на 25% и 29%, соответственно.

2.2.2 Оптимизированные константы сдвига.

В алгоритме BLAKE-512 функция G производит четыре сдвига на 64-битовых словах в 32, 25, 16 и 11 битах, соответственно. BLAKE2b заменяет 25 на 24 и 11 на 63.

* Использование сдвига на 24 бита позволяет процессорам с поддержкой SSSE3 производит два сдвига параллельно с одной SIMD инструкцией. Это уменьшает арифметические затраты на выполнение функции G. В последних версиях процессоров Intel количество циклов выполнения инструкций уменьшается с 18 до 16, что дает 12-процентный выигрыш в скорости.
* Сдвиг на 63 бита может быть реализован как дублирование и сдвиг с последующим применением операции логического ИЛИ. Это обеспечивает небольшое увеличение скорости на платформах, где дублирование и сдвиг могут обрабатываться параллельно (в некоторых новых версиях процессоров Intel). К тому же, сдвиг вправо на 63 бита эквивалентен сдвигу влево на 1 бит, что ускоряет эту процедуру в некоторых архитектурах, где сдвиг на 1 бит реализован особенным образом.

2.2.3 Минимальное дополнение и финализационные флаги.

BLAKE2 дополняет последний блок сообщения нулевыми битами тогда и только тогда, когда это необходимо. Если длина сообщения кратна длине блока, то дополнение не производится.

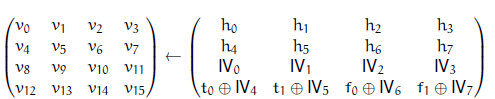
Алгоритм BLAKE2 вводит вспомогательные финализационные флаги f0 иf1, которые подаются на вход в функцию сжатия. Размер флагов – 64 бита для BLAKE2b и 32 бита для BLAKE2s.

* Назначение флага f0 – определение последнего блока. Он принимает значение ff…ff, если обрабатываемый блок последний, и 00…00 в другом случае.
* Флаг f1 служит сигналом того, что обрабатываемый блок является последним узлов определенного уровня в деревне хешей.
* Финализационные флаги участвуют в функции сжатия, как будет описано ниже.

2.2.4 Меньшее количество констант.

В то время как BLAKE в функции сжатия использует 8 констант в качестве инициализационного вектора и еще 16 дополнительных констант, BLAKE2 использует всего 8. Это позволяет использовать на 128 байт меньше как оперативной, так и постоянной памяти для реализации BLAKE2b и 64 байта для BLAKE2s.

16 векторов v текущего состояния инициализируется теперь иначе. Матрица инициализации представлена на рисунке 6.



**Рис. 6** Заполнение матрицы текущего состояния *v*

Заметим, что здесь используются финализационные флаги f вместо дублирования счетчика, как это было в BLAKE.

Функция G в алгоритме BLAKE2b определяется следующими операциями:

В алгоритме BLAKE2s функция G немного упрощена:

2.2.5 Порядок бит от старшего к младшему.

BLAKE2 так же, как SHA-1 и SHA-2 записывает данные блока в порядке от старшего разряда к младшему (big-endian). BLAKE2, как и MD5, записывает данные от младшего разряда к старшему (little-endian) в связи с тем, что большинство платформ работают именно так (процессоры AMD и Intel). Переход к такой записи дает небольшое увеличение скорости и зачастую упрощает реализацию.

2.2.6 Счетчик байтов.

Счетчик t считает байты, а не биты. Это упрощает реализацию и уменьшает риск ошибок, поскольку конечное приложения измеряют данные именно в байтах. Такое изменение увеличивает объем обрабатываемых данных в 8 раз в сравнении с BLAKE.

2.2.7 Использование соли в хешировании.

В предшественнике BLAKE2, хеш-функции LAKE было введено использование соли при хешировании для упрощения процесса рандомизации в проектировании цифровой подписи.

В BLAKE2 соль используется один раз при инициализации, действуя на вектор IV вместо того, чтобы действовать каждый раз на функцию сжатия. Это упрощает функцию сжатия и сберегает несколько байт оперативной памяти, потому что соль больше не требуется хранить.

2.2.8 Блок-параметр.

Блок-параметр в алгоритме BLAKE2 складывается по модулю 2 с вектором IV прежде чем начинается обработка первого блока сообщения.

Ниже описаны параметры, которые составляют данный блок.

Основные параметры:

* Длина хеш-значения в байтах (1 байт): целое число от 1 до 64 для BLAKE2b и от 1 до 32 для BLAKE2s.
* Длина ключа в байтах (1 байт): целое число от 0 до 64 для BLAKE2b и от 0 до 32 для BLAKE2s (значение 0 устанавливается, если ключ не используется).
* Соль (16 или 8 байт): случайная строка длины 16 байт для BLAKE2b и 8 для BLAKE2s.
* Персонализация (16 или 8 байт): случайная строка длины 16 байт для BLAKE2b и 8 для BLAKE2s (по умолчанию устанавливается null).

Параметры для хеширования посредством хеш-дерева:

* Коэффициент разветвления (1 байт): целое число от 0 до 255 (устанавливается 0, если коэффициент не определен и 1 в режиме последовательной обработки блоков).
* Максимальная глубина (1 байт): целое число от 1 до 255 (устанавливается 255, если не определено, и 1 в режиме последовательной обработки).
* Максимальная длина листьев в байтах (4 байта): целое число от 0 до 232-1, что достигает предела в 4 Гб (устанавливается 0, если не определено, либо в режиме последовательной обработки).
* Смещение узла (8 или 6 байт): целое число от 0 до 264-1 для BLAKE2b и до 248-1 для BLAKE2s (устанавливается 0 для первого, крайнего слева, листа или в режиме последовательной обработки).
* Глубина узла (1 байт): целое число от 0 до 255 (устанавливается 0 для листьев или в режиме последовательной обработки).
* Внутренняя длина хеш-значения в байтах (1 байт): целое число от 0 до 64 для BLAKE2b и до 32 для BLAKE2s (устанавливается 0 в режиме последовательной обработки).

В целом, размер параметра получается 50 байт для BLAKE2b и 32 байта для BLAKE2s. Любые байты, что остались, резервируются для дальнейшего использования и/или для особого использования приложением. Им присваивается значение null. Значения, содержащие больше, чем 1 байт записываются в порядке от младшего к старшему разряду (little-endian). Заметим, что к дереву хешей можно добавить ключ. В таком случае, хешируется то количество его байт, которое равно максимальной длине листа.

2.2.9 Дерево хешей.

Блок параметров поддерживает режимы хеширования посредством хеш-дерева - двоичного или троичного, обновляемого хеш-дерева случайной глубины, обновляемого хеш-дерева фиксированной глубины и т.д. Важно, что в отличие от иных функций BLAKE2 не ограничивает длину листа и коэффициент разветвления значениями, равными степеням 2.

### 2.3 Механизм хеширования посредством хеш-дерева.

Простым языком, при хешировании посредством хеш-дерева блоки данных, представляющие собой байты «длины листьев», обрабатываются независимо друг от друга, после чего объединяются соответствующие хеши, используя древовидную структуру, где каждый узел в качестве входных данных получает конкатенацию хешей «коэффициентов разветвления». Параметры смещения узла и глубины узла гарантируют, что при каждом обращении к хеш-функции (хешированию отдельного листа внутреннего узла) используется отдельная хеш-функция. Завершающий флаг f1 сигнализирует о том, что производится хеширование последнего узла на заданной глубине («последнего» в отношении счетчика смещения узла, как для листьев, так и для промежуточных узлов). Флаг f1 может иметь значение, отличное от нуля, лишь для последнего блока, а для корневого узла f1 всегда устанавливается значение ff...ff.

Механизм хеширования посредством хеш-дерева проиллюстрирован на рис. 2 и 3, на которых даны схемы деревьев с заданными различными параметрами и входными длинами. Восьмиугольниками на схеме обозначены листья (например, экземпляры хеш-функций, обрабатывающих вводные данные), узлы с двойной линией (включая листья) – это последние узлы слоя, что значит, установлен флаг f1. Метками «i:j» обозначены глубина узла (i) и его смещение (j). Отсылаем вас к [7], где приведен подробный анализ конструкций надежного дерева хешей.

# 3. Безопасность и скорость работы BLAKE2

### 3.1 Безопасность

Целью BLAKE2 является обеспечение самого высокого уровня безопасности как в практическом понимании – устойчивость к атакам нахождения (второго) прообраза и коллизионным атакам, - так и с точки зрения теоретических понятий – псевдослучайности или недифференцируемости.

С 2009 года по настоящий момент было проведено 14 криптоаналитических исследований сокращенной версии BLAKE. Самой серьезной атакой на BLAKE как на хеш-функцию, а не на ее отдельные компоненты является атака нахождения прообраза, проведенная Дзи (Ji) и Лянъюй (Liangyu), со сложностью 2241 и 2481 для BLAKE-256 и BLAKE-512 соответственно. На удивление большинством исследователей рассматривалась сокращенная версия функции сжатия в BLAKE, без учета ограничений, налагаемых IV. Приведем самые последние результаты подобного типа.

Разработчика алгоритма BLAKE2 считают, что уменьшение количества раундов и изменение констант сдвига вряд ли способны ощутимо сократить надежность BLAKE2 в сравнении с BLAKE. Приведем некоторые выводы о стойкости хеш-функции относительно других изменений:

**Функция сжатия без использования соли**. BLAKE2 добавляет соль к инициализационным векторам IV, а не при каждом сжатии. Это позволяет сохранить уникальность хеш-функции для любой определенной соли, но способствует теоретическим мультиколлизионным атакам, осуществляемым на основании автономных предвычислений. Тем не менее, так остается меньше «контролируемых» бит в начальном состоянии функции сжатия, что затрудняет нахождение фиксированных точек.

**Множество допустимых IV.** В силу большого количества допустимых блоков параметров в BLAKE2 возможны множественные допустимые значения начального вектора h. К примеру, если у атакующего есть оракул, который возвращает коллизии для случайного вектора h и сообщений, то шанс успешности атаки хеш-функции повышается так как есть множество допустимых целей, а не единственная. Однако такой сценарий подразумевает, что коллизии можно найти за относительно небольшое время, что означает, что хеш-функция уже взломана.

**Упрощенная процедура дополнения до блока**. Новая процедура дополнения не предусматривает, в отличие от BLAKE, включение в блок данных длины сообщения. Однако несложно заметить, что длина сообщения косвенно закодирована посредством счетчика и что дополнение до блока данных сохраняет явно выраженное кодирование, которое присутствовало в изначальной процедуре. Другими словами, упрощение процедуры дополнения блока данных не сказывается на надежности хеш-функции.

### 3.2 Скорость

BLAKE2 работает гораздо быстрее BLAKE – в основном в силу сокращенного числа раундов. На длинных сообщениях версии BLAKE2b и BLAKE2s, как ожидается, дадут прирост в скорости приблизительно на 25% и 29%, без учета экономии ресурсов в силу отсутствия констант, оптимизированных констант сдвигов и записи байт в младшего к старшему. От параллельных версий BLAKE2bp и BLAKE2sp на длинных сообщениях ожидается 4-х- и 8-кратное увеличение скорости в сравнении с BLAKE2b и BLAKE2s при реализации с использованием многопоточности на ЦПУ c 4 и более ядрами (как на большинстве пользовательских и серверных процессоров: AMD FX-8150, Intel Core i5-2400S и т.д.).

BLAKE2 и его параллельный вариант могут использовать преимущества таких архитектурных возможностей, или их комбинации, как параллелизм на уровне команд. Большинство современных процессоров являются суперскалярными, то есть, обладают способностью выполнения нескольких инструкций за цикл посредством конвейеризации, а также способностью исполнения команд с изменением их последовательности и выполнения прочих подобных методов. В BLAKE2 реализован параллелизм 4 инструкций внутри функции G; процессоры, которые в состоянии параллельно исполнять большее количество инструкций, могут реализовать свой потенциал в BLAKE2bp посредством вызова независимой функции сжатия. Среди процессоров, известных количеством параллельно исполняемых команд, Intel’s Core 2, i7, Itanium либо AMD’s K10, Bulldozer и Piledriver.

**Архитектура SIMD** (один поток команд - много потоков данных). Разработанные первоначально для ускорения выполнения мультимедийных задач, многие современные процессоры имеют векторные блоки, позволяющие осуществлять обработку данных с помощью архитектуры SIMD. И снова BLAKE2 может использовать преимущества векторных блоков не только в своих функциях G, но также в режиме хеширования посредством хеш-дерева, выполняя несколько сжатие в несколько потоков. Микроархитектуры с внедренной SIMD можно встретить в последних процессорах Intel и AMD, в SoC-платформа с архитектурой ARM, PowerPC и Cell.

**Многоядерность**. Технологические ограничения производства полупроводников, помимо ограничений параллелизма на уровне команд, подтолкнули производителей процессоров к еще одному виду параллелизма, где на один кристалл микросхемы устанавливается несколько независимых ЦПУ, таким образом предоставляя программисту условия для использования параллелизма на уровне потока. И пусть последовательный BLAKE2 не может воспользоваться данным преимуществом, параллельный режим и прочие древовидные режимы могут осуществлять промежуточное хеширование каждый в своем потоке. Среди кандидатов на подобный подход сейчас последние процессоры Intel и socAMD, IBM Cell, последние модели ARM, UltraSPARC и Loongson.

# 4. Хеш-функция в блокчейне

### 4.1 Применения хеш-функций

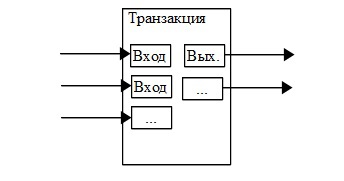
Область применения хеш-функций чрезвычайно широка, они успешно решают практически все задачи защиты компьютерной информации. Спектр областей применения хеш-функции включает в себя построение ассоциативных массивов, поиск дубликатов в сериях наборов данных, построение уникальных идентификаторов для наборов данных, вычисление контрольных сумм от данных (сигнала) для последующего обнаружения в них ошибок (возникших случайно или внесённых намеренно), возникающих при хранении и/или передаче данных, сохранение паролей в системах защиты в виде хеш-кода (для восстановления пароля по хеш-коду требуется функция, являющаяся обратной по отношению к использованной хеш-функции) и многие др.

Также хеш-функции играют важную роль в новой технологии, связанной с защитой информации - блокчейнах. Для того чтобы понять тонкости применения хеш-функции в блокчейне, рассмотрим более подробно, как он устроен на примере блокчейна в криптовалютах.

### 4.2 Принцип работы блокчейна

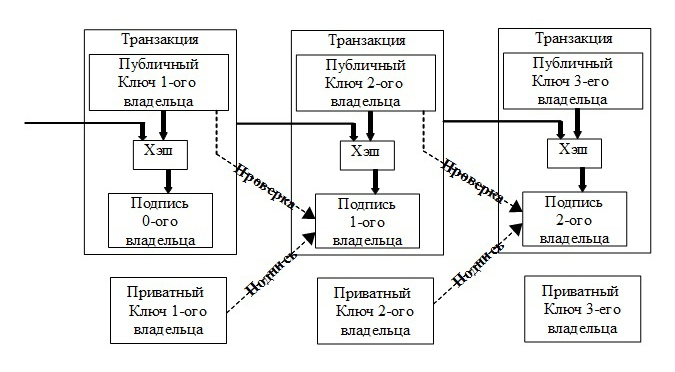
4.2.1 Основной механизм

В большинстве криптовалют состояние системы хранится в неподтвержденных входящих транзакциях (UTXOs – unspent transaction outputs). Пользователь указывает список входов (источников) транзакций и выходов (получателей). В качестве входов используются выходы предыдущих транзакций. Чтобы можно было делить и комбинировать суммы, транзакции содержат множество входов и выходов. Обычно они имеют либо один вход от большой предыдущей транзакции, либо множество входов, комбинирующих малые суммы, а также хотя бы два выхода: один для платежа и один для возврата сдачи, если она есть, обратно посылающему (рис.7).



**Рис.7** Входы и выходы транзакции

Чтобы ими воспользоваться, пользователю необходимо предоставить доказательство того, что именно он владеет этими средствами. Это осуществляется за счет цифровой подписи. Пользователь, пересылающий деньги следующему, добавляет в конец транзакции хеш предыдущей транзакции, подписанный своей электронной подписью, и публичный ключ получателя (рис.8). Таким образом, владелец монет может проследить цепочку их получения с самого первого их обладателя.



**Рис. 8** Цепочка транзакций

Транзакции объединяются в блоки, блоки – в цепочку блоков, называемую *блокчейном*, в которой каждый блок ссылается на предыдущий. Первый блок ни на что не ссылается, он, как правило, прописан в протоколе. Новые блоки генерируются в соответствии с определенными правилами, установленными протоколом. Важной функцией этих правил является противостояние атакам, а также достижение консенсуса (согласования) в случае появления нескольких версий блокчейна.

Существует несколько механизмов достижения консенсуса, один из которых будет описан ниже.

4.2.2 Доказательство работы (англ. «Proof of Work» – PoW)

Работу этого алгоритма будем рассматривать на основе Биткоина, как наиболее известной системы. Каждый блок в Биткоине состоит из двух частей:

* заголовок блока, содержащий такие параметры, как время создания, ссылка на предыдущий блок и список хешей транзакций.
* список транзакций в виде дерева Меркла (Приложение А)

Заголовок хешируется дважды с помощью алгоритма SHA-256. Результатом является число, лежащее в интервале от 0 до 2256-1. Чтобы учесть различные варианты реализации, будем рассматривать хеш-функцию в общем виде hash(\*) с разным количеством аргументов и значениям, лежащими в интервале [0, *M*]. Например, мы можем записать все параметры в одну строку и передать как один аргумент *B*.

Чтобы блок считался валидным, его хеш должен не превышать определенного значения:

|  |  |  |
| --- | --- | --- |
|  | , | (1) |

где *D* – целевая сложность, определяемая системой следующим образом: подсчитывается количество блоков, сгенерированных за определенное время и сравнивается с эталоном (в Биткоине 1 блок каждые 10 минут), соответственно, если количество сгенерированных блоков оказалось меньше, то значение *D* уменьшается, больше – увеличивается.

Нет никакого способа найти подходящее *B*, кроме перебора подряд всех вариантов.

### 4.3 Моделирование работы хеш-функции в блокчейне

Целью моего исследования было определение зависимости времени, затрачиваемого на поиск значения хеш-функции с нулевым префиксом определенной длины от длины префикса в условиях выбора входа хеш-функции по определенным правилам.

Для осуществления данной цели я написала программу (текст программы в Приложении Б-В). В программе был реализован алгоритм хеширования BLAKE2b (Приложение Б) и генератор одного блока данных (Приложение В). Согласно алгоритму, блок данных должен иметь длину 1024 бит. Генератор блока на вход получает число (<1024), равное длине постоянной части блока, остальное – переменная часть. Постоянная часть выполняет роль неизменяющихся данных для блока блокчейна (список транзакций, хеш предыдущего блока и т.д.), переменная часть представлена в виде счетчика, увеличивающегося на единицу при каждом следующем обращении к генератору блока.

Программа работает в нескольких режимах. Варианты работы отображены на снимках экрана в Приложении А. Первый режим вывод хеш любой бинарной строки, поданную на вход. В этом режиме наглядно показан «лавинный эффект» хеш-функции. Второй режим производит подсчет количества итераций и времени в миллисекундах на нахождение хеш-суммы с заданной длиной нулевого префикса. Он использует генератор блока данных для генерации исходного сообщения. Третий режим работы аналогичен второму, только он производит вычисления для заданного отрезка длин нулевого префикса.

Я провела эксперимент, подсчитав количество итераций и времени для нахождения хешей с длинами нулевого префикса от 1 до 25 с различными длинами переменной части блока.

Введем некоторые обозначения.

Пусть M – исходный блок. Он состоит из двух частей. X – переменная часть, Y – постоянная. M = X+Y;

При выполнении хеш-функции H(M), мы получаем цепочку бит h1,…,hn.

H(X,Y) = h1…hn.

L – длина нулевого префикса. Если h1= h2=…= hL=0, то блок считается сформированным (валидным).

|X| – длина части сообщения X.

Эксперименты были проведены на 4 вариантах длины переменной части блока:

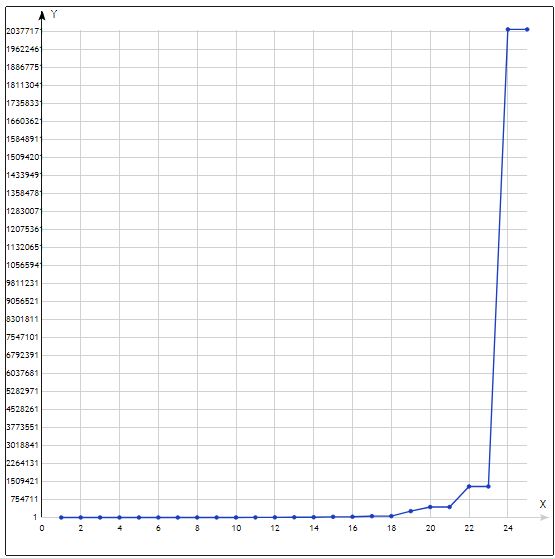
1) |X| = 512 (1/2 от длины блока)

2) |X| = 256 (1/4 от длины блока)

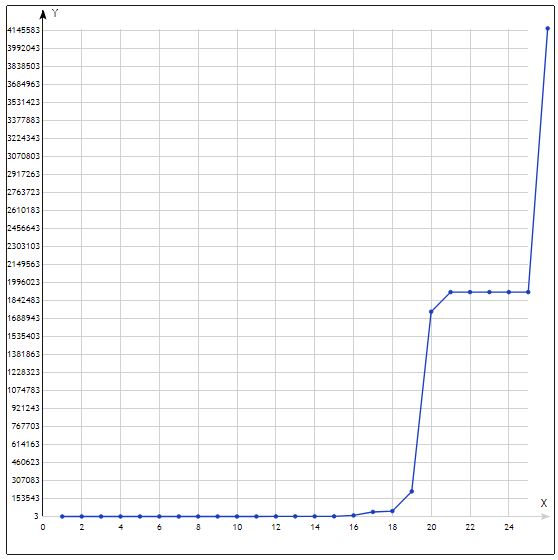
3) |X| = 128 (1/8 от длины блока)

4) |X| = 64 (1/16 от длины блока)

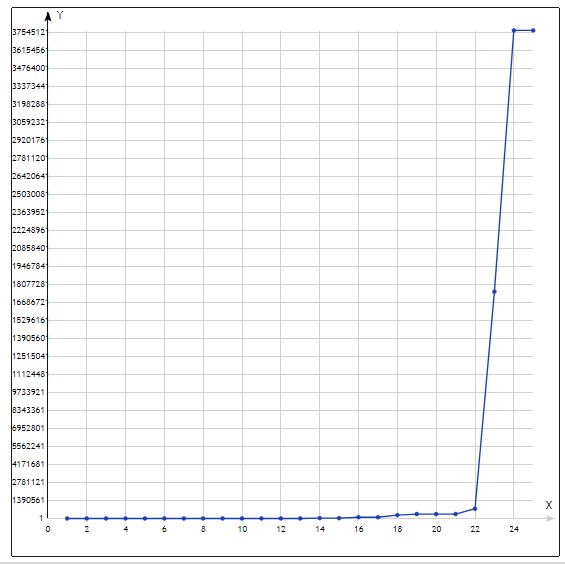
В результате запуска третьего режима работы программы, были получены данные, на основании которых построены графики зависимости количества итераций, необходимых для нахождения хеш-суммы с заданной длиной нулевого префикса, от длины префикса. Ниже на рисунках 9-12 показаны графики для длин переменной части блока 512, 256,128, 64, соответственно.



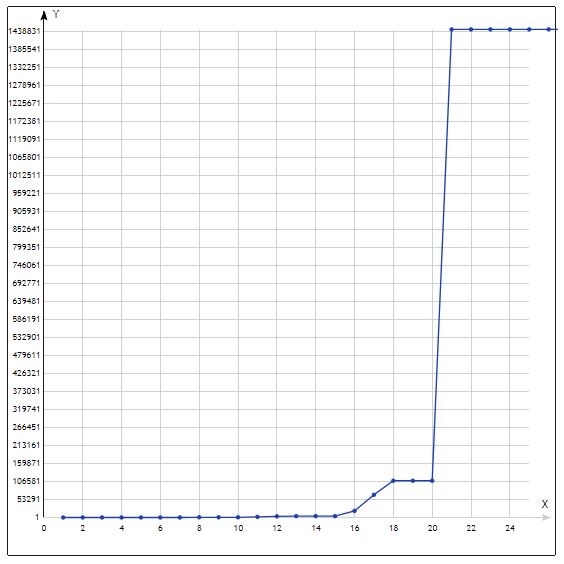
**Рис. 9** Зависимостьколичества итераций от длины префикса. |X| = 512



**Рис. 10** Зависимостьколичества итераций от длины префикса. |X| = 256



**Рис. 11** Зависимостьколичества итераций от длины префикса. |X| = 128

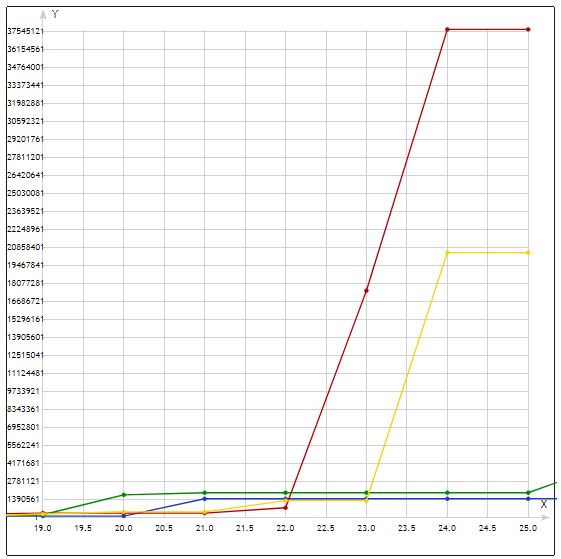


**Рис. 12** Зависимостьколичества итераций от длины префикса. |X| = 64

На графиках видно, что чем больше длина переменной части, тем позже происходит резкий скачок увеличения числа итераций для нахождения подходящего хеша, однако размер такого «скачка» мы сможем оценить чуть позже.

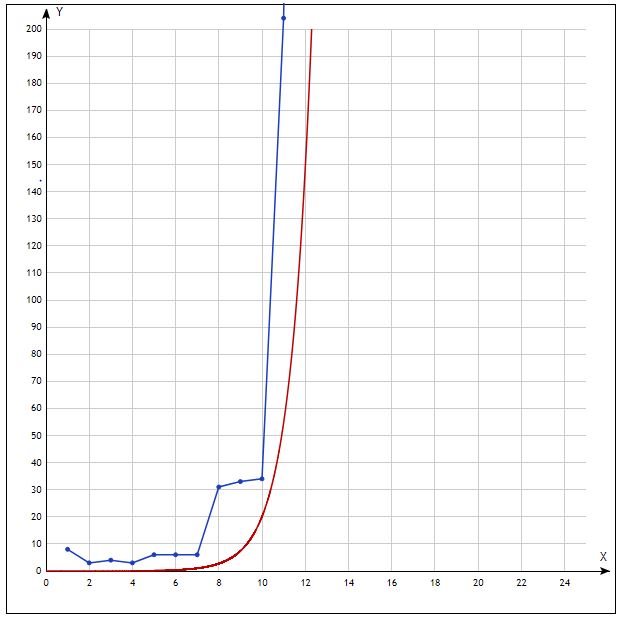
Большая длина переменной части гарантирует большее количество вариантов, что усложняет поиск коллизий, а соответственно, делает подделку блока в блокчейне более трудоемкой задачей. Однако при увеличении длины переменной части, уменьшается количество значимых данных в блоке, соответственно, при большом потоке данных, увеличивается количество блоков для обработки, что увеличивает трудоемкость работы программы.

Попробуем найти наиболее оптимальное соотношение длины переменной части к длине блока. Для этого выведем имеющиеся графики вместе (рис. 13).



**Рис. 13** Графики для длин переменной части блока 512 (жёлтый), 256 (красный), 128 (зеленый), 64 (синий)

На графике из рисунка 13 видно, что количество итераций для заданного префикса с переменными длинами в 256 бит и 128 бит увеличивается более равномерно. Создается ощущение, что они не изменяются совсем, однако это ложное ощущение. На рисунках 10 и 11 видно, что они увеличиваются, причем, экспоненциально. Для доказательства приведем сравнение графика с переменной частью 256 и графика экспоненциальной функции (рис. 14).

****

**Рис. 14**  Сравнение графика с переменной частью 256 и графика экспоненциальной функции

Большая разница между графиками происходит из-за того, что на красном и жёлтом графиках происходит слишком резкий скачок количества итераций.

Сложность нахождения подходящего хеш-значения D, определяемая в пункте 4.2.2, должна меняться в зависимости от количества блоков, сгенерированных за определенное время, которое сравнивается с эталоном. Учитывая развивающиеся мощности работы современных компьютеров, зависимость количества итераций на поиск значения хеш-функции с нулевым префиксом определенной длины от длины префикса должна быть более или менее равномерной для большей гибкости использования механизма достижения консесуса «Доказательство работы».

Соответственно, для использования хеш-функции BLAKE2b в описанном выше механизме, оптимальное соотношение длины переменой части блока данных от длины блока должно быть в диапазоне от 1/4 до 1/2, то есть для длины блока в 1024 бит – от 128 бит до 256 бит.

# Заключение

В ходе работы было рассмотрено несколько различных хеш-функций. Был реализован алгоритм хеш-функции BLAKE2, на основе которого была смоделирована работа хеш-функции в блокчейне. При помощи этого был проведен эксперимент, указанный в цели работы и проанализированы результаты эксперимента.

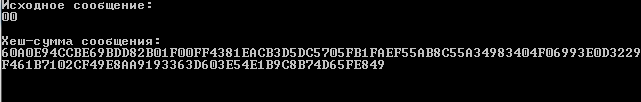
В дальнейшем хотелось бы провести более глубокий анализ безопасности данной хеш-функции и смоделировать некоторые типы атак для проведения анализа.

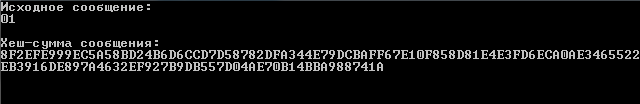
# Список литературы.

1. BLAKE2: simpler, smaller, fast as MD5. Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O’Hearn, Christian Winnerlein. 29 января 2013. 20 с.
2. SHA-3 proposal BLAKE. Версия 1.3. Jean-Philippe Aumasson, Luca Henzen, Willi Meier, Raphael C.-W. Phan. 16 декабря 2010. 79 с.
3. The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC). M-J. Saarinen, Ed. ISSN: 2070-1721. Ноябрь 2015. 30 с.
4. Proof of Stake vs. Proof of Work. White Paper, BitFury Group. 13 сентября 2015. 26 с.
5. Bitcoin: A Peer-to-Peer Electronic Cash System. Satoshi Nakamoto, [www.bitcoin.org](http://www.bitcoin.org). 9 с.
6. Веб-сайт <https://131002.net/blake/>.
7. Веб-сайт <https://blake2.net/>.

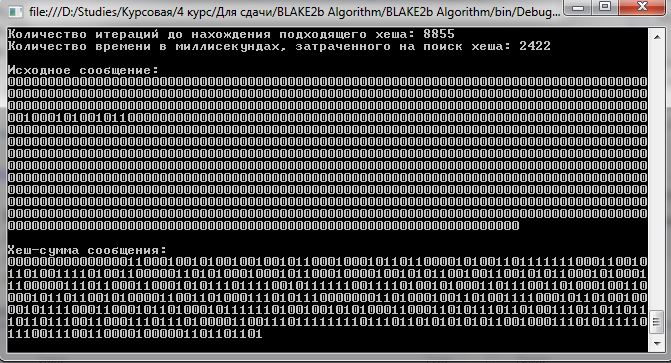
# Приложение А

**Примеры работы разных режимов программы.**





**Рис.15** Пример первого режима работы программы. Здесь можно увидеть «лавинный эффект» хеш-функции

****

**Рис. 16** Пример второго режим программы. Нахождения хеша с нулевым префиксом длины 15

# Приложение Б

class Hasher

{

#region Constant values

private readonly ulong[] IV = new ulong[8] {

0x6A09E667F3BCC908UL,

0xBB67AE8584CAA73BUL,

0x3C6EF372FE94F82BUL,

0xA54FF53A5F1D36F1UL,

0x510E527FADE682D1UL,

0x9B05688C2B3E6C1FUL,

0x1F83D9ABFB41BD6BUL,

0x5BE0CD19137E2179UL};

private readonly int[,] Permutation = new int[12, 16] {

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

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

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

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

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

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

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

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

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

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

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

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

#endregion

byte bytesInBlock = 128;

byte hashLenght = 64;

public string Hash (string message)

{

string result = "";

ulong[][] messageBlocks = ToUlongArray(message);

ulong[] hashBlocks = GetHashVectors(messageBlocks, (ulong)message.Length, 0, hashLenght);

foreach (var block in hashBlocks)

result += ToBinaryString(block);

return result;

}

#region Transformations

//преобразование бинарной строки в массив блоков длины 1024

//в случае, когда длина сообщения не кратна 1024, оно дополняется нулевыми битами

//каждый блок представляет собой массив из 16 64-битных слов

private ulong[][] ToUlongArray(string binaryMessage)

{

int messageLenth = binaryMessage.Length;

int numberOfWholeBlocks = messageLenth / 1024;

int numberOfBlocks = numberOfWholeBlocks;

if (messageLenth > numberOfWholeBlocks \* 1024)

numberOfBlocks++;

ulong[][] messageBlocks = new ulong[numberOfBlocks][];

for (int i = 0; i < numberOfWholeBlocks; i++)

{

messageBlocks[i] = new ulong[16];

for (int j = 0; j < 16; j++)

{

messageBlocks[i][j] = Convert.ToUInt64(binaryMessage.Substring(0, 64),2);

binaryMessage = binaryMessage.Remove(0, 64);

}

}

//формирование последнего блока, если сообщение не было кратно 1024

if (numberOfBlocks > numberOfWholeBlocks)

{

messageBlocks[numberOfBlocks-1] = new ulong[16];

for (int i = 0; i < 16; i++)

{

if (binaryMessage.Length < 64)

{

messageBlocks[numberOfBlocks-1][i] = Convert.ToUInt64(binaryMessage, 2);

break;

}

messageBlocks[numberOfBlocks-1][i] = Convert.ToUInt64(binaryMessage.Substring(0, 64),2);

binaryMessage = binaryMessage.Remove(0, 64);

}

}

return messageBlocks;

}

//преобразование 64-битного слова в бинарную строку

private string ToBinaryString(ulong hexNumber)

{

string result = "";

while (hexNumber != 0)

{

ulong nextDigit = hexNumber & 0x01;

result = nextDigit + result;

hexNumber = hexNumber >> 1;

}

if (result.Length < 64)

result = new string('0', 64 - result.Length) + result;

return result;

}

//преобразование 64-битного слова в шестнадцитиричную строку

//в данном алгоритме не используется. Нужна была для тестирования случайности хеша

private string ToHexString(ulong hexNumber)

{

string result = "";

result = string.Format("0x{0:X}", hexNumber);

return result;

}

#endregion

#region BLAKE2b algorithm functions

private ulong[] GetHashVectors(ulong[][] messageBlocks, ulong messageLenght, byte keyLenght, byte hashLenght)

{

ulong[] h = new ulong[8];

for (int i = 0; i < 8; i++)

h[i] = IV[i];

h[0] = h[0] ^ 0x01010000 ^ keyLenght ^ hashLenght;

if (messageBlocks.Length > 1)

{

for (int i = 0; i < messageBlocks.Length - 1; i++)

h = Compress(h, messageBlocks[i], (ulong)(i + 1) \* bytesInBlock, false);

}

if (keyLenght == 0)

h = Compress(h, messageBlocks[messageBlocks.Length - 1], messageLenght, true);

else h = Compress(h, messageBlocks[messageBlocks.Length - 1], messageLenght + bytesInBlock, true);

return h;

}

//функция сжатия

private ulong[] Compress(ulong[] h, ulong[] m, ulong counter, bool finalizationFlag)

{

ulong[] vectors = Inizializaton(h, counter, finalizationFlag);

for (int r = 0; r < 12; r++)

{

G\_Function(vectors, m, 0, 4, 8, 12, 0, r);

G\_Function(vectors, m, 1, 5, 9, 13, 1, r);

G\_Function(vectors, m, 2, 6, 10, 14, 2, r);

G\_Function(vectors, m, 3, 7, 11, 15, 3, r);

G\_Function(vectors, m, 0, 5, 10, 15, 4, r);

G\_Function(vectors, m, 1, 6, 11, 12, 5, r);

G\_Function(vectors, m, 2, 7, 8, 13, 6, r);

G\_Function(vectors, m, 3, 4, 9, 14, 7, r);

}

h = Finalization(h, vectors);

return h;

}

private ulong[] G\_Function(ulong[] vectors, ulong[] messageBlock, int a, int b, int c, int d, int i, int r)

{

vectors[a] += vectors[b] + messageBlock[Permutation[r, 2 \* i]];

vectors[d] ^= vectors[a];

vectors[d] = ((vectors[d] >> 32) | (vectors[d] << 32));

vectors[c] += vectors[d];

vectors[b] = vectors[b] ^ vectors[c];

vectors[b] = ((vectors[b] >> 24) | (vectors[b] << 40));

vectors[a] = vectors[a] + vectors[b] + messageBlock[Permutation[r, 2 \* i + 1]];

vectors[d] ^= vectors[a];

vectors[d] = ((vectors[d] >> 16) | (vectors[d] << 48));

vectors[c] += vectors[d];

vectors[b] ^= vectors[c];

vectors[b] = ((vectors[b] >> 63) | (vectors[d] << 1));

return vectors;

}

//инициализация вектора текущего состояния v

private ulong[] Inizializaton(ulong[] h, ulong counter, bool finalizationFlag)

{

ulong[] vectors = new ulong[16];

for (int i = 0; i < 8; i++)

vectors[i] = h[i];

for (int i = 8; i < 15; i++)

vectors[i] = IV[i-8];

vectors[12] = vectors[12] ^ counter;

vectors[13] = vectors[13] ^ counter;

if (finalizationFlag == true)

{

vectors[14] = vectors[14] ^ 0xFFFFFFFFFFFFFFFF;

vectors[15] = vectors[15] ^ 0xFFFFFFFFFFFFFFFF;

}

return vectors;

}

//заключительный этап хеширования

private ulong[] Finalization(ulong[] h, ulong[] v)

{

for (int i = 0; i < 8; i++)

h[i] ^= v[i] ^ v[i + 8];

return h;

}

#endregion

}

# Приложение В

class MessageGenerator

{

int \_lenght;

int \_constantPartLenght;

int \_variablePartLenght;

int \_counter;

string \_messageConstant;

string \_messageVariable;

// по умолчанию генерируется блок длиной 1024 с постоянной частью 768

public MessageGenerator()

{

\_lenght = 1024;

\_constantPartLenght = 768;

\_variablePartLenght = 256;

\_counter = 0;

\_messageVariable = "";

\_messageConstant = new string('0', 768);

}

public MessageGenerator(int lenght, int constantPart)

{

if (lenght < constantPart)

throw new Exception("Входные параметры некорректны.");

\_lenght = lenght;

\_constantPartLenght = constantPart;

\_variablePartLenght = lenght - constantPart;

\_counter = 0;

\_messageVariable = "";

\_messageConstant = new string ('0', constantPart);

}

public int GetCurrentIteration()

{

return \_counter;

}

public string Generate ()

{

string temp = Convert.ToString(\_counter, 2);

\_messageVariable = new string('0', \_variablePartLenght - temp.Length) + temp;

\_counter++;

//return \_messageConstant+\_messageVariable; //разный порядок постоянной и переменной части сообщения

return \_messageVariable + \_messageConstant;

}

}