МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования   
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий

Кафедра Информационных систем и технологий

Специальность 1– 40 01 02 «Информационные системы и технологии»

Направление специальности 1– 40 01 02 03 Информационные системы

и технологии (издательско– полиграфический комплекс)

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**КУРСОВОГО ПРОЕКТА:**

по дисциплине «Защита информации и надежность информационных систем»

Тема: «Реализация и исследование алгоритма хэширования Blake2b»

Исполнитель

Студент 3 курса группы 1 Почта К.А.

(Ф.И.О.)

Руководитель работы ассистент Копыток Д.В. (учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой

Председатель Копыток В.О.

(подпись, дата)

**Содержание**

[Введение 3](#_Toc102934492)

[1 Аналитический обзор литературы 4](#_Toc102934493)

[1.1 Что делает алгоритм Blake2b 4](#_Toc102934494)

[1.2 Особенность алгоритма 4](#_Toc102934495)

[1.3 Быстродействие и реализация 6](#_Toc102934496)

[1.4 Криптостойкость 6](#_Toc102934497)

[1.5 Алгоритм 6](#_Toc102934498)

[1.5.1 Константы 7](#_Toc102934499)

[1.5.2 Функция сжатия 7](#_Toc102934500)

[1.5.3 Инициализация 8](#_Toc102934501)

[1.5.4 Раундовая функция 8](#_Toc102934502)

[1.5.5 Последний шаг 10](#_Toc102934503)

[1.5.6 Хеширование сообщения 10](#_Toc102934504)

[1.5.7 Отличие от quarterround алгоритма ChaCha 11](#_Toc102934505)

[2 Обзор аналогов 12](#_Toc102934506)

[3 Проектирование 18](#_Toc102934507)

[4 Описание средств, используемых для разработки 19](#_Toc102934508)

[5 Тестирование 27](#_Toc102934509)

[6 Руководство пользователя 28](#_Toc102934510)

[Заключение 31](#_Toc102934511)

[Список используемых источников 32](#_Toc102934512)

[ПРИЛОЖЕНИЕ А 33](#_Toc102934513)

[ПРИЛОЖЕНИЕ Б 35](#_Toc102934514)

# **Введение**

С хешированием мы сталкиваемся едва ли не на каждом шагу: при работе с браузером (список Web– ссылок), текстовым редактором и переводчиком (словарь), языками скриптов (Perl, Python, PHP и др.), компилятором (таблица символов). По словам Брайана Кернигана, это «одно из величайших изобретений информатики». Заглядывая в адресную книгу, энциклопедию, алфавитный указатель, мы даже не задумываемся, что упорядочение по алфавиту является не чем иным, как хешированием.

Термин хеширование появился сравнительно недавно, в 1967 году. Хеширование есть разбиение множества ключей (однозначно характеризующих элементы хранения и представленных, как правило, в виде текстовых строк или чисел) на непересекающиеся подмножества (наборы элементов), обладающие определенным свойством. Это свойство описывается функцией хеширования, или хеш– функцией, и называется хеш– адресом. Решение обратной задачи возложено на хеш– структуры (хеш– таблицы): по хеш– адресу они обеспечивают быстрый доступ к нужному элементу

В данной курсовой работе рассматривается одна из функций хеширования криптовалюты – Blake2b. Криптографическая хэш– функция — это специальный класс хэш– функций, который имеет различные свойства, необходимые для криптографии: детерминированное, быстрое вычислимость, сложность обратного вычисления и др.

В целом, в криптовалюте есть такое понятие как блокчейн, это такая цепочка, состоящая из последовательно идущих, друг за другом блоков, сами же блоки содержат информацию, например о транзакции между пользователями системы, это секретные данные, которые могут украсть злоумышленники, поэтому существуют специальные алгоритмы, которые позволяют эту информацию хешировать.

Исходя из вышеописанного целью курсовой работы является реализация одного из алгоритмов хеширования Blake2b, изучение его сильный и слабых сторон, а также особенности его реализации. Для реализации данного алгоритма будет разработано программное средство, демонстрирующее его работу, и руководство по его использованию.

# **Аналитический обзор литературы**

## **1.1 Что делает алгоритм Blake2b**

Главная функция, которую выполняет любой криптографический алгоритм – это преобразование (хеширование) произвольного массива данных в строку определенной длины (битовую строку). Функция, которая реализовывает алгоритм и выходное преобразование, называется функцией свёртки (или хеш– функцией).

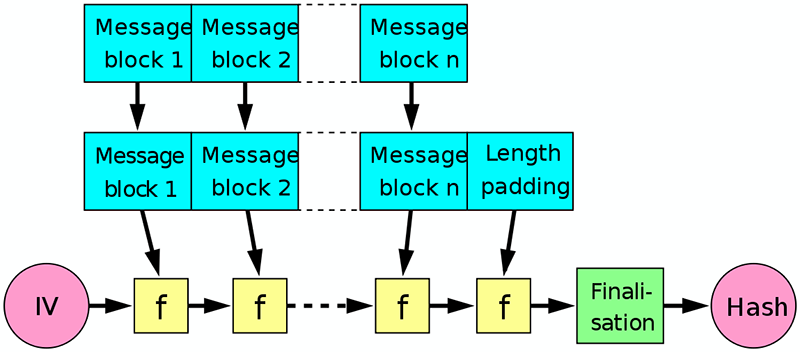


Рисунок 1 – Графическое представление работы алгоритма

В блокчейне все транзакции фиксируются в децентрализованном реестре. Они формируются в блоки определенного размера, установленного сетью, и связываются с предыдущими блоками посредством цифровой подписи. Вот эту подпись и хеширует криптографический алгоритм, а выполняет все это компьютер майнера.

* 1. **Особенность алгоритма**

BLAKE2b[[1](#_Список_используемых_источников_1)] – один из двух алгоритмов, основанных на родительской функции BLAKE2 (второй – BLAKE2s). Он оптимизирован для 64– разрядных платформ, тогда как Blake2s оптимизирован для 8– 32– разрядных платформ.

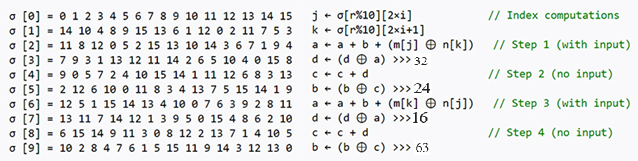
[](#_Список_используемых_источников)

Рисунок 2 – Поэтапная работа алгоритма

Он основан на поточном шифре, известном как ChaCha, разработанном Даниэлем Дж. Бернштейном и нацелен на максимально возможную защиту хеш– функции. И действительно, ни одна атака не была обнаружена на BLAKE2 с момента ее публикации в 2012 году.

Хеш– функция BLAKE имеет три главных методов устройства:

* итеративную организацию обработки данных в режиме HAIFA;
* внутренние строение local– wide– pipe;
* улучшенную версию криптошифра ChaCha, который прекрасно параллезируется и уже показал высокую безопасность.

Новая усовершенствованная версия протокола BLAKE под названием BLAKE2 появилась в конце 2012 года. Это криптографическая хеш– функция, которая работает оперативнее, чем MD– 5, SHA1, SHA2 и SHA3, и такая же безопасная, как и последний стандарт SHA3. BLAKE2 был принят многими проектами благодаря высокой скорости, безопасности и простоте.

Он был разработан командой экспертов по криптоанализу, внедрению и криптографической технике:

* Жан– Ф**илипп Аумассон**: главный дизайнер BLAKE, автор научных работ по криптоанализу;
* С**эмюэл Невес**: автор исследовательских работ по эффективным реализациям;
* Z**ooko Wilcox– O’Hearn**: один из разработчиков защищенной распределенной системы хранения Tahoe– LAFS;
* C**hristian Winnerlein**: автор ряда проектов на C#.

Алгоритм BLAKE2 имеет две вариации:

* BLAKE2b (или просто BLAKE2) оптимизирован для 64– разрядных платформ, включая ARM с поддержкой NEON, и производит дайджесты любого размера от 1 до 64 байт;
* BLAKE2s оптимизирован для 8– и 32– разрядных платформ и производит дайджесты любого размера от 1 до 32 байт.

BLAKE2 включает в себя 4– полосный параллельный BLAKE2bp и 8– позиционный параллельный BLAKE2sp, предназначенный для повышения производительности на многоядерных или SIMD– процессорах.

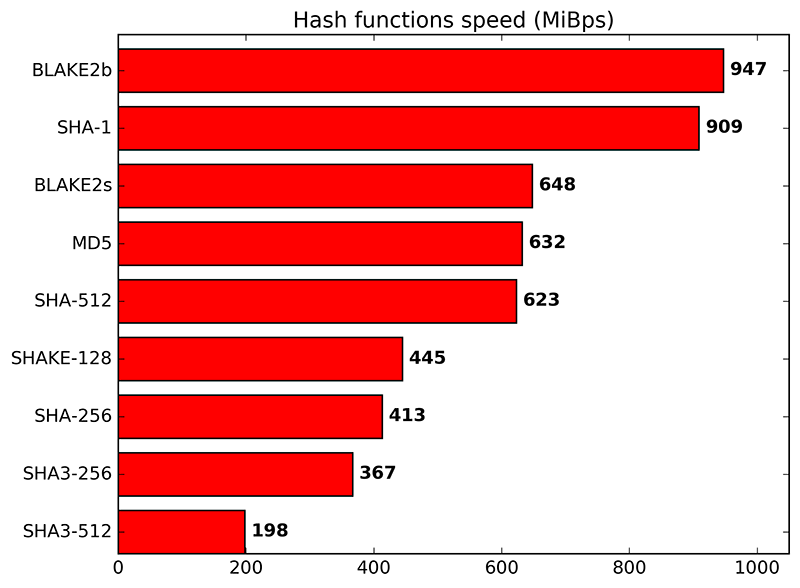


Рисунок 3 – Сравнение быстродействия других алгоритмов

BLAKE2 показывает очень хорошие результаты на 64– битных процессорах: на Intel Core i5– 6600 (микроархитектура Skylake, 3310 МГц) BLAKE2b может выдавать скорость передачи 1 GiB в секунду или 3,08 циклов на байт.

**1.3 Быстродействие и реализация**

Сравним быстродействие алгоритма на двух различных процессорах:

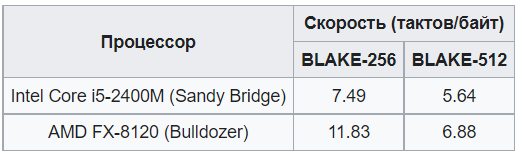


Рисунок 4 – Сравнение быстродействия алгоритмов на разных процессорах

Алгоритмы возможно реализовать на базе следующих микропроцессоров:

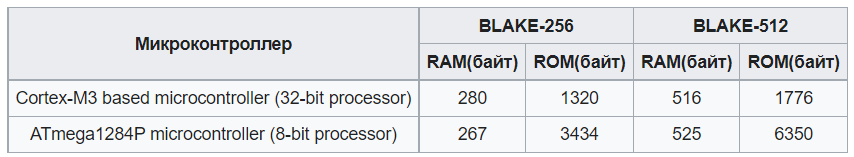


Рисунок 5 – Сравнение работы алгоритмов на разных микроконтроллерах

Как можно заметить, BLAKE– 512, имеет скорость, сравнительно большую чем BLKAE– 256, на обоих микропроцессорах.

* 1. **Криптостойкость**

Так как в основе алгоритма лежит шифр ChaCha[[1](#_Список_используемых_источников_1)], этот код направлен на улучшение системы от стронного воздействия. Согласно официальным заявлениям компании SIA, с момента релиза этого алгоритма на систему не было произведено ни одной успешной атаки.

## **Алгоритм**

Как упоминалось ранее, хеш– функция BLAKE построена из трёх ранее перечисленных компонентов:

* режим итерации HAIFA[[2](#_Список_используемых_источников_1)];
* внутренняя структура local wide– pipe;
* алгоритм сжатия для BLAKE, является модифицированной версией хорошо параллелизуемого поточного шифра ChaCha, чья безопасность тщательно проанализирована.

Рассмотрим алгоритм BLAKE– 256.

BLAKE– 256 оперирует с 32– битными словами и возвращает 32– байтный хеш.

**1.5.1 Константы**

Существуют начальные константы[[2](#_Список_используемых_источников_1)], INITIAL VALUES (IV):

IV0 = 6A09E667 IV1 = BB67AE85

IV2 = 3C6EF372 IV3 = A54FF53A

IV4 = 510E527F IV5 = 9B05688C

IV6 = 1F83D9AB IV7 = 5BE0CD19

16 констант (первые числа числа ПИ):

c0  = 243F6A88 c1  = 85A308D3

c2  = 13198A2E c3  = 03707344

c4  = A4093822 c5  = 299F31D0

c6  = 082EFA98 c7  = EC4E6C89

c8  = 452821E6 c9  = 38D01377

c10 = BE5466CF c11 = 34E90C6C

c12 = C0AC29B7 c13 = C97C50DD

c14 = 3F84D5B5 c15 = B5470917

Перестановки {0..15}, использующиеся во всех Blake функциях:

σ0 = 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

σ1 = 14 10 4 8 9 15 13 6 1 12 0 2 11 7 5 3

σ2 = 11 8 12 0 5 2 15 13 10 14 3 6 7 1 9 4

σ3 = 7 9 3 1 13 12 11 14 2 6 5 10 4 0 15 8

σ4 = 9 0 5 7 2 4 10 15 14 1 11 12 6 8 3 13

σ5 = 2 12 6 10 0 11 8 3 4 13 7 5 15 14 1 9

σ6 = 12 5 1 15 14 13 4 10 0 7 6 3 9 2 8 11

σ7 = 13 11 7 14 12 1 3 9 5 0 15 4 8 6 2 10

σ8 = 6 15 14 9 11 3 0 8 12 2 13 7 1 4 10 5

σ9 = 10 2 8 4 7 6 1 5 15 11 9 14 3 12 13 0

**1.5.2 Функция сжатия**

Функция сжатия[[2](#_Список_используемых_источников_1)] алгоритма BLAKE2b 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\*4 = 120 байт = 960 бит). Возвращает функция сжатия только новое значение переменных цепочки: h' = h'0,…,h'7. В дальнейшем будем обозначать h'=compress(h, m, s, t).

### **1.5.3 Инициализация**

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

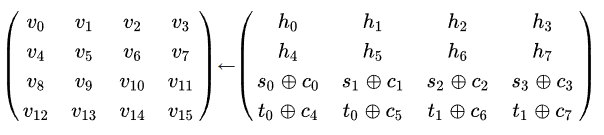


Рисунок 6 – Переменные состояния

Во время работы функции сжатия, ей удобнее работать с конкретными переменными, они же являются входными состояниями, состоящими из счетчика, соли, сообщения и переменных цепочки.

### **1.5.4 Раундовая функция**

После того, как состояние V инициализировано, запускается серия из 14 раундов. Раунд – это операция над состоянием V, которая производит вычисления, разбитые на следующие блоки:

G0(v0, v4, v8 , v12) G1(v1, v5, v9 , v13) G2(v2, v6, v10, v14) G3(v3, v7, v11, v15)

G4(v0, v5, v10, v15) G5(v1, v6, v11, v12) G6(v2, v7, v8 , v13) G7(v3, v4, v9 , v14)

На r– ом раунде, блок вычислений Gi[[2](#_Список_используемых_источников_1)]{a,b,c,d) работает следующем образом:

j ← σr%10[2×i]

k ← σr%10[2×i+1]

a ← a + b + (mj ⊕ ck)

d ← (d ⊕ a) >>> 32

c ← c + d

b ← (b ⊕ c) >>> 24

a ← a + b + (mk ⊕ cj)

d ← (d ⊕ a) >>> 16

c ← c + d

b ← (b ⊕ c) >>> 63

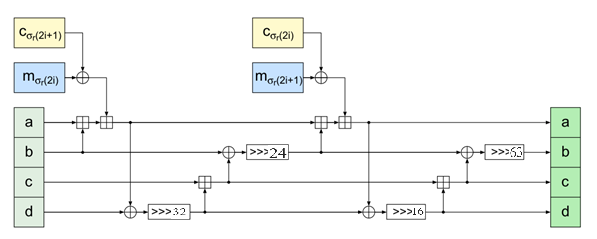


Рисунок 7 – Графическое представление работы блока вычислений Gi

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 раундах, номера r которых больше 9, используется перестановка σr%10, например на 13– том раунде используется σ3.

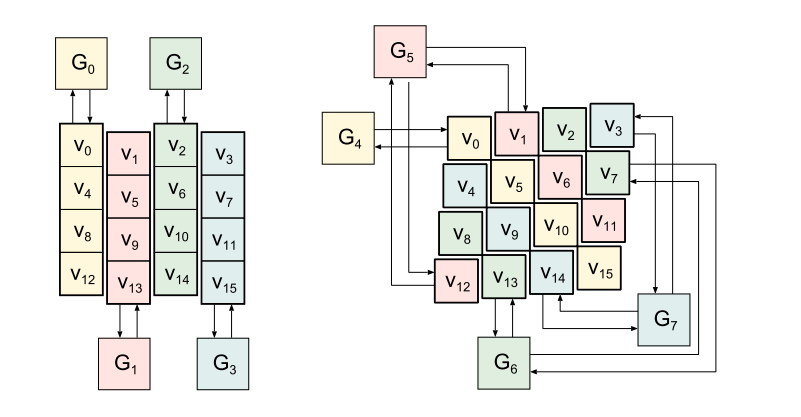


Рисунок 8 – Колонный и диагональный сдвиг

Как можно заметить функция G, выполняется несколько раз для определенных входных состояний.

### **1.5.5 Последний шаг**

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

### **1.5.6 Хеширование сообщения**

Опишем процесс хеширования сообщения m длиной l<2^64 бит. Сначала сообщение дополняется функцией padding (заключается в добавлении ничего не значащих данных к зашифрованной информации, для повышения криптостойкости) данными для кратности 512 битам (64 байтам), затем, блок за блоком, его обрабатывает функция сжатия compression function (преобразование данных с целью уменьшения его объёма).

В функции padding сообщение сначала дополняется битами, так, что его длина становится по модулю 512 равной 447: сначала добавляется 1, затем необходимое количество нолей. После этого прибавляется ещё одна 1 и 64– битное представление длины сообщения l от старшего бита к младшему. Таким образом, длина сообщения становится кратной 512. Padding гарантирует, что длина сообщения станет кратной 512 битам. Добавление ничего не значащих данных к зашифровываемой информации, нацеленное на повышение криптостойкости. Различные техники дополнения применялись в классической криптографии. [⇨], обширное применение техники дополнений нашли в компьютерных системах шифрования.

Чтобы высчитать хеш сообщения, результат функции padding делится на блоки из 16 слов m0,…,mN– 1. Пусть Li — количество бит исходного сообщения в блоках m0,…,mi, то есть исключая те биты, которые были добавлены в процедуре padding. Например, если сообщение имеет длину 600 бит, то после процедуры padding оно будет иметь длину 1024 бита и будет разделено на два блока: m0 и m1. Притом L0=512, L1=600. В некоторых случаях последний блок не содержит бит оригинального сообщения. Например, если в исходном сообщении 1020 бит, то в результате процедуры padding оно будет иметь длину 1536 бит и в m0 будет 512 бит исходного сообщения, в m1 — 508, а в m2 — 0. Выставим L0=512, L1=1020, а L2=0. То есть правило следующее: если в последнем блоке нет бит оригинального сообщения, то выставим счётчик LN– 1 равным 0. Это гарантирует, что если i ≠ j, то Li ≠ Lj. Значение соли определяется пользователем или задаётся равным 0, если её не нужно использовать (s0=s1=s2=s3=0). Хеш сообщения таким образом вычисляется:

h0 ← IV

for i=0,...,N– 1

hi+1 ← compress(hi,mi,s,li)

return hN.

Процесс хеширования представлен на блок– схеме ниже:

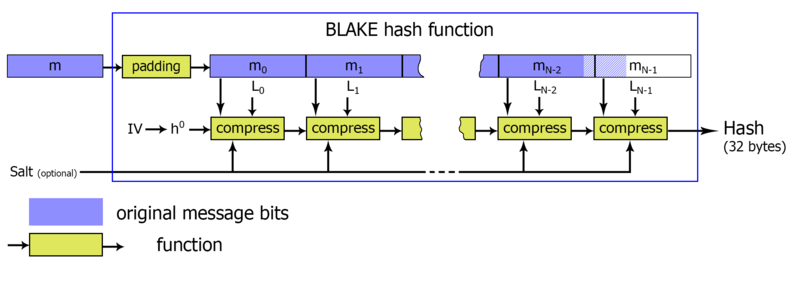


Рисунок 9 – Процесс хеширования

Видно, что сообщение разбивается на блоки и каждому блоку отдельно применяется функция сжатия, на выходе получаем хеш.

### **1.5.7 Отличие от quarterround алгоритма ChaCha**

* Добавление констант[[2](#_Список_используемых_источников_1)] к сообщению.;
* Измененное направление[[2](#_Список_используемых_источников_1)] сдвига.

# **2 Обзор аналогов**

В качестве аналогов Blake2b приведены следующие алгоритмы хеширования:

1 SHA– 256

SHA– 256 — это безопасный алгоритм шифрования, который приобрел популярность после его использования в коде биткоина. Аббревиатура SHA — это Secure Hash Algorithm, а 256 означает, что алгоритм криптовалюты генерирует 256– битный хеш, т.е. строку (дайджест) размером 256 бит. Хешрейт для криптовалют, работающих на основе SHA– 256, вычисляется в единицах Gigahash в секунду (GH/s). На создание блока уходит от шести до десяти минут.

Особенности

Алгоритм SHA– 256 был изобретен Агентством Национальной Безопасности США в 2001 году. Он входит в семейство алгоритмов SHA и сейчас является единственным алгоритмом криптовалют из данного семейства, который прошел тест на устойчивость к таким видам атак как нахождение коллизий и нахождение прообраза, что имеет решающее решение для безопасности криптовалют, работающих на основе данного алгоритма.

Помимо криптовалют, SHA– 256 также широко используется в некоторых других технологиях.

Например, работа протоколов безопасности, таких как TLS, SSL, PGP, SSH, построена на SHA– 256.

В 2009 году, когда Bitcoin был известен лишь очень ограниченному кругу людей, для майнинга использовали обычные компьютеры, которые производили вычисления при помощи центрального процессора. Позже начали применять более мощные графические процессоры.

Однако сейчас, когда популярность биткоина просто огромная, экономически выгодным стало использование лишь ASIC– майнеров — специальных устройств, обладающих большой вычислительной мощностью.

Монеты алгоритма SHA– 256

На алгоритме SHA– 256 работает Bitcoin, а также ряд других криптовалют, которые в основном являются форками биткоина.

Криптовалюты алгоритма SHA– 256:

* Bitcoin (BTC).
* Bitcoin Cash (BCH).
* Bitcoin SV (BSV).
* Emercoin (EMC).

2 Equihash

Equihash — это анонимный алгоритм криптовалют, увидивший свет в 2016 году.  Первой криптовалютой, которая использовала Equihash в качестве базы, была Zcash. Создание блоков занимает 150 секунд, а хешрейт измеряется в Megahash в секунду (MH/s). В основе данного алгоритма лежит хеш– функция, которая построена на принципе «Парадокса дней рождений» — это математическая закономерность, которая используется для расчета вероятности. Правило гласит:

Если в комнате находится 23 человека, то вероятность того, что день рождения как минимум двух из них приходится на один и тот же день, составляет 50%. Исходя из этой закономерности, вероятность нахождения числа nonce в процессе майнинга равняется 2, вознесенное в степень N и разделенное на 2.

Особенности

Этот алгоритм криптовалют был разработан Александром Бирюковым и Дмитрием Ховратовичом — учеными Университета Люксембурга, которые входят в исследовательскую группу CryptoLUX. В 2016 году разработка была представлена широкому круга.

Equihash требователен к объему оперативной памяти, а не к скорости обработки математических вычислений.

Это делает майнинг «айсикоустойчивым», а сеть более децентрализованной. Для майнинга криптовалют, работающих на Equihash, используются видеокарты, обладающие минимальным объемом памяти в 2 Гб. Самые лучшие результаты показало применение графических процессоров бренда NVidia.

Однако для майнинга Equihash также были разработаны устройства ASIC. Сегодня самыми популярными являются две модели: Antminer Z9 mini от Bitmain и A9 ZMaster от менее известной компании Innosilicon. В отличие от команды Monero, разработчики Zcash не предприняли никаких действий для защиты своей криптовалюты от централизации майнинга, а лишь высказали свое огорчение.

Монеты алгоритма Equihash

Самые популярные криптовалюты, работающие на алгоритме Equihash:

* Zcash (ZEC).
* Bitcoin Gold (BTG).
* Komodo (KMD).
* ZClassic (ZCL).
* ZenCash (ZEN).
* Bitcoin Private (BTCP).

3 Алгоритм Ethash (Dagger Hashimoto)

Ethash (Dagger Hashimoto) — это алгоритм криптовалют, разработанный специально для майнинга Ethereum. В основу работы данного алгоритма хеширования положены два разных алгоритма: Dagger, созданный Виталиком Бутериным, и Hashimoto (название состоит из слов hash, shift, and modulo), разработанный программистом Thaddeus Dryja. Хейшрейт алгоритма Ethash измеряется в Megahash в секунду (MH/s).

Особенности

Dagger — это алгоритм криптовалют, который предъявляет жесткие требования к памяти видеокарты. Принцип его работы похож на Scrypt, однако его производительность выше, особенно это заметно в условиях повышения сложности сети.

Но Dagger имеет некоторые уязвимости, поэтому он эффективен только в паре с Hashimoto.

Алгоритм Hashimoto работает с операциями ввода/вывода в особом режиме. В частности, он ограничивает скорость добычи, из– за того, что количество памяти для записывания и считывания информации не является бесконечным.

Hashimoto — это алгоритм криптовалют, который требует большой объем памяти, за счет чего не получается выполнить большое количество операций ввода/вывода, т.е. для расшифровки не получится использовать метод бесконечного подбора случайных значений. Это и являлось основной причиной того, почему устройства ASIC не подходили для майнинга Ethereum (об этом ниже).

Для майнинга монет, работающих на основе Ethash, используется вычислительная мощность GPU– процессоров (видеокарт). Наибольшую эффективность показали видеокарты бренда AMD. Однако и применение видеокарт Nvidia 10– й серии приносят неплохой профит майнерам.

Основное требование — это высокий показатель оперативной памяти, который постоянно увеличивается из– за роста сложности сети.

Как мы говорили выше, устройства ASIC не подходили для добычи монет, работающих на основе Ethash, однако все изменилось летом 2018 года, когда крупнейший китайский производитель майнеров Bitmain выпустил в продажу модель Innosilicon A10 ETHMaster, т.е. ASIC для Ethereum.

Монеты алгоритма Ethash

Изначально Ethash был создан для Ethereum, однако этот алгоритм также используют и другие монеты. Криптовалюты, в которых используется алгоритм Ethash:

* Ethereum (ETH).
* Ethereum Classic (ETC).
* Ubiq (UBQ).
* Expanse (EXP).
* Pirl (PIRL).
* Musicoin (MUSIC).
* QuarkChain (QKC).

4 Scrypt

Scrypt — это алгоритм майнинга криптовалют, который раньше был интересен многим одиночным майнерам в виду его устойчивости к так называемой «аппаратной атаке». Скорость создания блоков в блокчейне, работающем на базе Scrypt, составляет около 30 секунд. Хешрейт как и у Ethash измеряется в Megahash в секунду (MH/s). Scrypt, прежде всего, стал популярным благодаря его применению в криптовалюте Litecoin.

Особенности

История создания этого алгоритма началась с того, что вскоре стало ясно, что майнинг биткоина очень легко монополизировать, из– за того что простота функции SHA– 256 позволяет автоматизировать процесс майнинга.

Поэтому основной задачей при создании Scrypt было усложнение механизма генерации блоков за счет повышенных требований к ресурсам, используемым для вычислительных операций. В частности, решающее значение имеет объем оперативной памяти (как и в Ethash), при этом требования к энергопотреблению и вычислительной мощности намного ниже чем в случае SHA– 256.

Изначально для добычи криптовалют на основе Scrypt применялись центральные и графические процессоры, однако алгоритм не мог противостоять майнинг– корпорациям и в 2014 году был создан первый ASIC для Scrypt– монет.

В ответ на это крипто энтузиасты создали усовершенствованный алгоритм под названием Scrypt– N, который впервые был применен в монете Vertcoin. Отличается он тем, что требования к оперативной памяти постоянно растут.

Монеты алгоритма Scrypt

Алгоритм майнинга Scrypt положен в основу работы следующих криптовалют:

* Litecoin (LTC).
* Dogecoin (DOGE).
* Redcoin (RED).
* Monacoin (MONA).
* Viacoin (VIA).
* Syscoin (SYS).

5 Алгоритм функций X11

X11 — это алгоритм шифрования, в котором вместо одной функции используются одинадцать. Это означает, что данная технология может обеспечить высокую степень безопасности, ведь чтобы нанести вред системе, злоумышленнику придется взломать все 11 функций, а это очень маловероятно, потому что внесенные изменения будут видны уже после взлома первой функции, и у разработчиков будет масса времени защитить систему, прежде чем хакер дойдет до одиннадцатой функции. Алгоритм криптовалют X11 был создан специально для майнинга Dash.

Особенности

Данный алгоритм шифрования был разработан создателем одной из топовых криптовалют (Dash, в прошлом Darkcoin) — Эваном Даффилдом. Все началось с того, что он хотел улучшить анонимность и взаимозаменяемость биткоина, однако сообщество не одобрило его идею. Тогда Эвану не оставалось ничего, кроме как создать свою собственную криптовалюту. По словам Даффилда на разработку ушли всего лишь одни выходные.

Еще одним достоинством X11 является то, что майнинг криптовалют, созданных на его основе, является очень экономным в плане потребления энергоресурсов. Следует сказать, что энергоэффективность — это всего лишь приятный «бонус», так как Эван не ставил за цель создать алгоритм, который бы не требовал больших энергозатрат. Данное свойство заметили майнеры.

Проведенные эксперименты показали, что добыча Dash на видеокартах потребляет на 30– 50% меньше электроэнергии, чем майнинг Bitcoin на ASIC– устройствах.

Для добычи монет, которые работают на основе алгоритма X11, можно использовать не только GPU, но и CPU. Это один из немногих сегодня алгоритмов, который допускает майнинг на процессорах, что делает его привлекательным для одиночных майнеров и небольших компаний. Конечно, ASIC для добычи криптовалют на основе алгоритма X11 также существуют.

Как признался сам создатель X11, он не старался создать «айсикоустойчивый» алгоритм, однако он сделал все возможное, чтобы производителям пришлось «хорошенько потрудиться».

На данный момент X11 — не единственный в своем роде, который использует больше одной хеш– функции.

По его примеру были созданы другие алгоритмы, которые также построены на интеграции нескольких хеш– функций: X12, X13, X14, X14, X15, X16 и даже X17.Однако X11 до сих пор является самым популярным и самым используемым.

Монеты алгоритма  X11

* Dash (DASH).
* Pura (PURA).
* Tao (XTO).
* Synergy (SNRG).
* Enigma (ENG).
* CannabisCoin (CANN).

6 CryptoNight

CryptoNight — это алгоритм для анонимных криптовалют, в основе которого лежат такие технологии как кольцевые подписи и одноразовые адреса.  Он стал, прежде всего, известен благодаря криптовалюте Monero (XMR). Генерация блока занимает около 60 секунд.

Особенности

Применение CryptoNight позволяет создать конфиденциальную, защищенную и взаимозаменяемую цифровую валюту. Эти свойства достигаются благодаря применению протокола CryptoNote, который был создан в 2012 году блокчейн– разработчиком (или группой разработчиков) под псевдонимом Николас ван Саберхаген. До сих пор неизвестно, кто скрывается за этим именем.

Высокая степень анонимности алгоритма CryptoNight достигается благодаря использованию двух технологий:

* Кольцевые подписи (шифруют отправителя);
* Одноразовые адреса (шифруют получателя).

Из– за того, что транзакции являются абсолютно анонимными, все монеты, построенной на алгоритме криптовалют CryptoNight, являются взаимозаменяемыми.

Это означает, что они не имеют «темного прошлого», как, например, Bitcoin, блокчейн которого позволяет отследить движение определенного коина и его участие в «грязных делах».

Еще одна очень важная особенность данного алгоритма состоит в том, что для майнинга наиболее эффективными является использование CPU в паре с GPU, хотя добыча только на центральном процессоре тоже дает положительные результаты. И, напротив, применение устройств ASIC является экономически невыгодным.

Секрет в том, что CryptoNight требовательный к оперативной памяти (для правильной работы нужно минимум 2 Гб), а не к скорости проведения математических вычислений, в чем сильны ASIC– майнеры.

Процесс поиска правильного nonce для генерации блока состоит в том, что CryptoNight создает случайные блоки, которые хранит в памяти вычислительного устройства, а не отправляет непрерывные частые обращения к майнеру с целью проведения математических операций, как это происходит в случае использования алгоритма SHA– 256 (используемого для Bitcoin).

Это позволяет построить более децентрализованную сеть и не допустить возникновения ситуации, когда больше половины хешрейта принадлежит одному узлу, и существует высокая вероятность проведения атаки 51%.

Для эффективного майнинга криптовалют, работающих на основе CryptoNight, стоит использовать центральные процессоры Intel Core поколений i5 и i7.

Особенностью которого является очень высокая степень конфиденциальности благодаря затемнению кода по каждой из транзакций. Анонимность в CryptoNight реализована за счёт использования кольцевых подписей (скрывают отправителя) и одноразовых адресов (скрывают получателя).

Также хорошие результаты показывает применение оборудования бренда Radeon HD (новые модели). Кроме того, для майнинга криптовалют на основе CryptoNight можно использовать ASIC– майнеры.

На данный момент для этого подходят такие модели: Antminer X3, Baikal Giant– N, PinIdea RR– 200, DragonMint X1/X2.

Однако стоит отметить, что Monero каждый год вносит изменения в протокол, которые помогают защитить добычу монеты от использования ASIC. Но другие монеты, работающие на CryptoNight, например, Bytecoin, Aeon, без проблем добываются при помощи ASIC.

Монеты алгоритма CryptoNight

На анонимном алгоритме CryptoNight созданы следующие криптовалюты:

* Monero (XMR).
* Bytecoin (BCN).
* DigitalNote (XDN).
* Electroneum (ETN).
* Aeon (AEON).
* CryptoNoteCoin.

В итоге можно сказать, что вышеперечисленные алгоритмы хеширование ничуть не уступают Blake2b, каждый из них имеет свои преимущества над другими алгоритмами, некоторые плохи в том, в чем хороши другие и наоборот, что выбирать для использования. Но стоит отметить быстроту работы алгоритма Blake2b, она весьма велика, по сравнению со своими аналогами[[1](#_Список_используемых_источников_1)].

# **3 Проектирование**

Решение проекта «Blake2b» представлено на рисунке ниже:

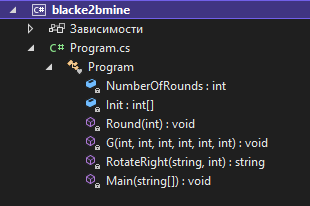


Рисунок 10 – Структура проекта

Решение представлено одним проектом «Blake2bmine», имеющим следующую структуру: имеется файл Program.cs, который содержит в себе ряд методов необходимых для реализации алгоритма, метод Main является точкой запуска приложения, которая открывает консоль при запуске приложения.

Весь функционал разработанного приложения представлен в классе Program.cs.

**4 Описание средств, используемых для разработки**

Было разработано 2 программы, первая для показа работа алгоритма на всех раундах, операции, производимые с входными состояниями, вторая, собственно, для генерации хеша.

Первая программа будет написан на современном объектно– ориентированном и тип безопасным языке программирования C#, позволяющим создавать разные типы безопасных и надежных приложений, выполняющихся в .NET. Версия .NET 6.0, в среде разработки Microsoft Visual Studio 2022, по шаблону «Консольное приложение».

Вторая программа написано в среде разработке IntelliJ IDEA, на языке Go, в результате будет представлять команду в командной строке, для генерации хеша.

Основные элементы, позволяющие реализовать алгоритм, располагаются в program.cs, структура которого представлена на рисунке 11.

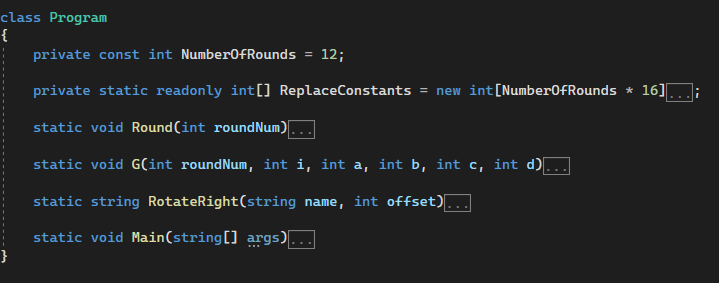


Рисунок 11 – Структура program.cs

Переменная NumberOfRounds задает количество раундов выполнения алгоритма, по правилам Blake2b имеет 12 раундов.



Рисунок 12 – NumberOfRounds

Целочисленный массив ReplaceConstants, это стандарт для всех BLAKE подобных алгоритмов хеширования, он задает перестановки {0,…,15}

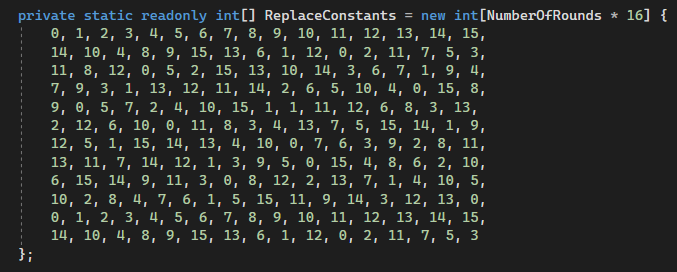


Рисунок 13 – Массив ReplaceConstants

Статический метод Round, возвращает void, метод принимает целочисленный параметр “r”, параметр олицетворяет номер раунда. Метод Round итерируется 12 раз, в соответствии с BLAKE2b, каждый раунд включает в себя трансформацию состояний V, основанную на одноядерной (функция выполняющаяся 1 раз за такт, так как работает в одном потоке) функции G. Важно понимать что операции G0– G3, выполняются над колонками и эта процедура называется column step или же колонный шаг, G4– G7 – диагональный шаг, каждая G имеет свою матрицу состояний V, производит над ними операции.

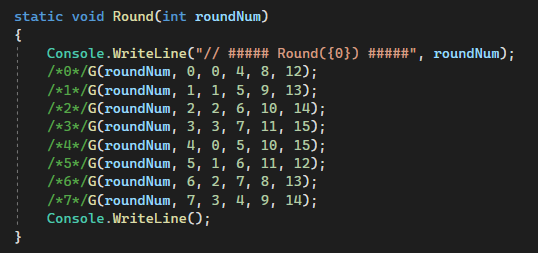


Рисунок 14 – Функция Round

Алгоритм BLAKE2B, использует одноядерную функцию – Gфункцию. Вначале вычисляется p, p0, p1, их вычисление необходимо для понимания какое семейство перестановок нужно использовать. Далее в соответствии со схемой, представленной на Рисунок. 7 Происходят преобразования входных состояний.

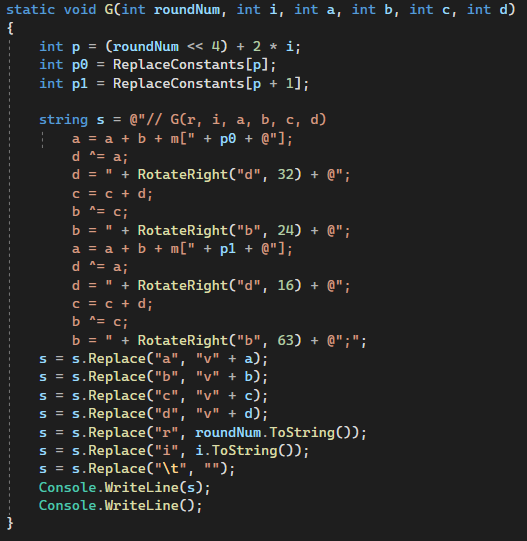


Рисунок 15 – Одноядерная функция G

Как можно заметить в функции G применяется функция RotateRight, как можно догадаться она меняет места определённое количество бит, что наглядно показано на Рисунок. 17 ниже. Кроме этого используется встроенная функция класса string – Replace, она нужна для замены непонятных a, b ,c ,d на ключевые состояния V[0– 15].

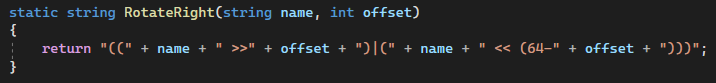


Рисунок 16 – Метод RotateRight

Метод RotateRight[[2](#_Список_используемых_источников_1)], служит бля изменения положения битов, применяется он 4 раза. Принимает на вход какое– нибудь состояние, и offset (сколько битов берется для разворота), сама функция возвращает строку, в которой происходит конкатенация входного состояния и битов для разворота. Ниже показано как происходит «разворот» битов.

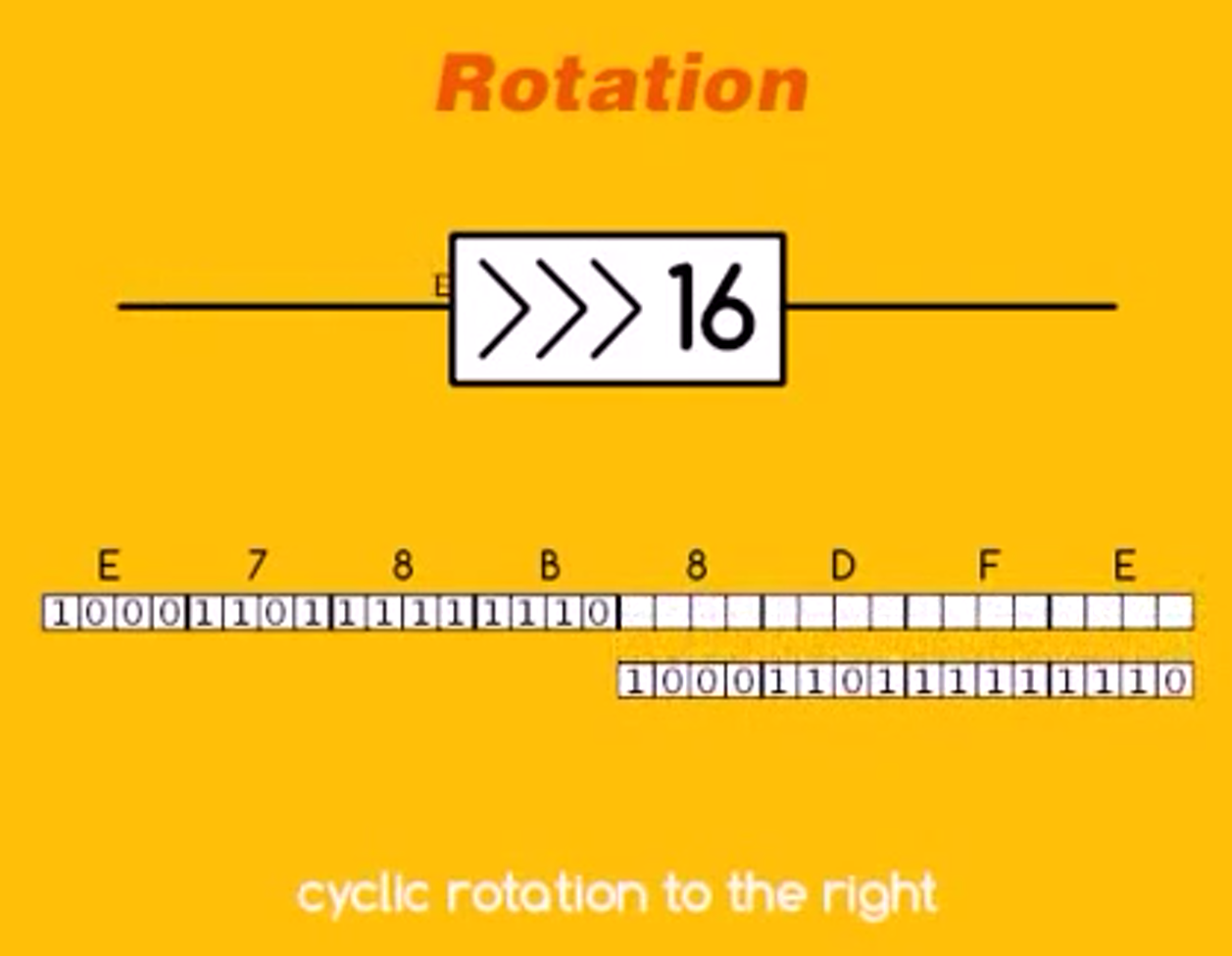


Рисунок 17 – Визуализация работы метода RotateRight

Как можно увидеть, при развороте на 16 бит, берется последние 16 бит состояния, и меняются местами с остальными битами, так они становятся вперед, соответсвенно на выходе получает другой битовоый вектор.

Ну и наконец метод Main, который служит точкой входа, содержащий метод Round, о котором говорилась выше, находящийся в цикле, выполняющийся по NumberOfRounds, а то есть 12 раз, как и предписано в документации blake2b.

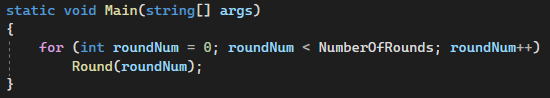


Рисунок 18 – Метод Main

Вторая программа как упоминалось выше написана на языке GoLang, рассмотрим все компоненты используемые для генерации хеша.



Рисунок 19 – Импорт пакетов

Весь код организуется в пакеты. Пакеты представляют удобную организацию разделения кода на отдельные части или модули. Модульность позволяет определять один раз пакет с нужной функциональностью и потом использовать его многкратно в различных программах.Вначале импортируются необходимые пакеты для работы, а именно flag ( этот пакет позволит добавить флаги в команду, в командной строке), fmt (этот пакет позволит прочитывать данные из файла, для последущего его хеширования), hash ( пакет для работы с хешом).

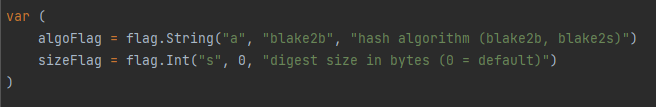


Рисунок 20 – Флаги для взаимодействия с программой

Можно увидеть 2 флага, 1 отвечает за алгоритм хеширования, второй за размер хеша в битах.

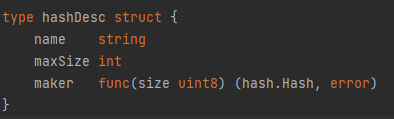


Рисунок 21 – Тип hashDesc

Типы данных определяют виды значений, которые сохраняются определенными переменными при написании программы Тип понадобится в дальнейшем, а именно его составляющие, name для, хранения пути к файлу, maxSize для указания размера хеш сообщения, и функция maker для генерации хеша.

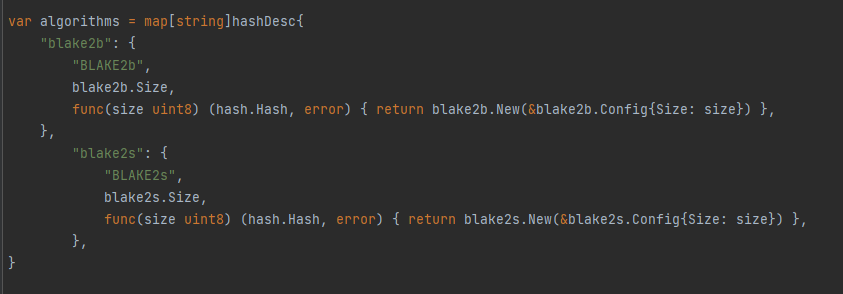


Рисунок 22 – Отображение algrothms

Algorithms это отображение которое представляет собой хеш– талицу map[K]V, ключ имеет тип string, значение тип hashDesc. Имеется два объекта, blake2b/s соответсвенно, два виды хеширования, которые можно выбрать, оба имеет функцию, возвращающую хеш.

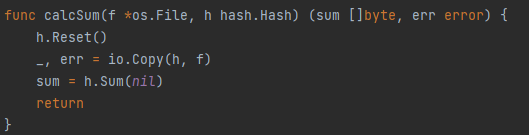


Рисунок 23 – Функция clcSum

Функция calcSum, имеет тип func \*os.file, hash.Hash и возврает массив байтов и, если возникает ошибка, ошибку. Используется для генерации хеша исходя из содержимого файла.

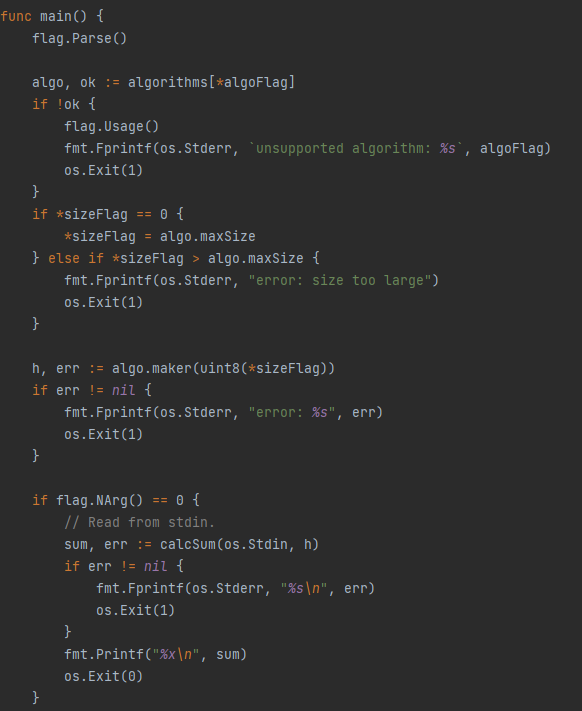


Рисунок 24 – Первая часть функции main.

Вначале происходить синтаксическая проверка вводимого флага с помощью функции Parse. В основном здесь происходит проверка вводимых данных и при возникновении непредвиденных ситуаций выводится сообщение для уведомления пользователя.

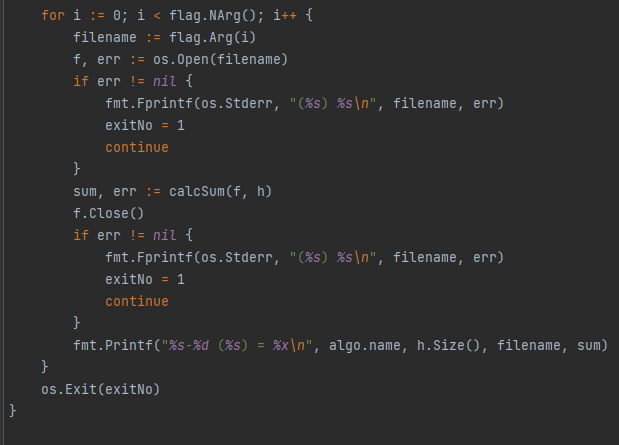


Рисунок 25 – Вторая часть функции main

Вторая часть метода main, представляет собой цикл, который выполняет до flag.NArg(), далее перменная filename, хранящая путь к файлу, которая затем испольщуется в метде Open, в качестве входного аргумента. После проверки, на существование файла, содержимое файла передается в метод calcSum, файл закрывается. После проверки, в выходной поток выводится название алгоритма, размер, имя файла и сам хеш.

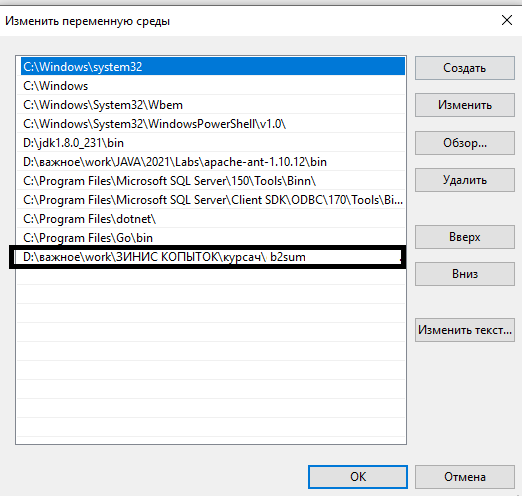


Рисунок 26 – Настройка переменных среды

Для корректной работы необходимо указать путь к проекту в перменной среды «PATH», после этого станет доступно использование программы в консоли.

# **5 Тестирование**

Приложение 1, представляет собой простой вывод работы алгоритма, саму его суть, все этапы его работы, тестировать там нечего.

Приложение 2 использует условные операторы if– else, использовавшиеся в реализации, была учитана возможность ввода некорректных данных, пример уведомления пользователя об ошибки приведен ниже.



Рисунок 27 – Ошибка при вводе размера хеша.

Как можно заметить, при вводе слишком большого значения для размера хеша, выводится уведомление.

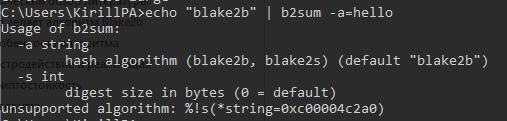


Рисунок 28 – Неккоректный алгоритм

Ввод неккоректного алгоритма в флаг, за которым последовало уведомления пользователя.

По результатам тестирования можно сказать, что приложение предусматривает исключительные ситуации.

# **6 Руководство пользователя**

С результатом работы приложения 1 можно ознакомиться запустив исполняемый файл, полученный после сборки проекта, результат можно увидеть ниже

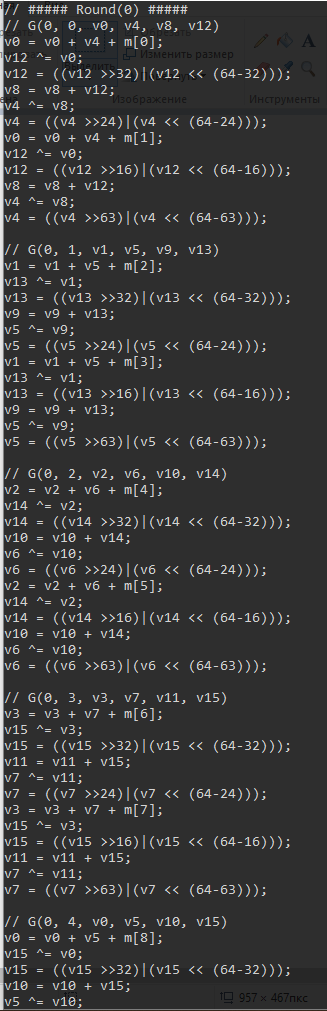
.

Рисунок 29 – Результат работы 0 раунда

Можно заметить, что на вход функции G подаются входные состояния v[0– 15] с которыми собственно и происходят взаимодействие, функция по умолчанию принимает определённые заранее[[2](#_Список_используемых_источников_1)] состояния.

Пример работы приложения 2, можно увидеть ниже. Сначала вводитися стандартная команда Windows echo, далее палка | и ключевое слово b2sum, в качестве флагов команды выступают – a=SIZE, которые будет задавать количество бит хеша, fileName1, fileName2, примеры можно увидеть на рисунках ниже.

Простая генерация хеша, по введенному сообщению.



Рисунок 30 – Пример генерации хеша без параметров.

Пример с генерацией хеша, с ограничением размера в 56 бит.



Рисунок 31 – Пример генерации хеша с ограничим в 56 бит

Далее рассмотрим генерацию хеша исходя из содержимого файла «text.txt»

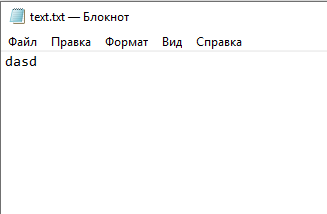


Рисунок 32 – Содержимое файла “text.txt”

Результат хеширование данных из файла приведен ниже.



Рисунок 33 – Пример с хешированием содержимого файла.

Также можно указывать два файла для хеширования, добавим файл “text1.txt”.

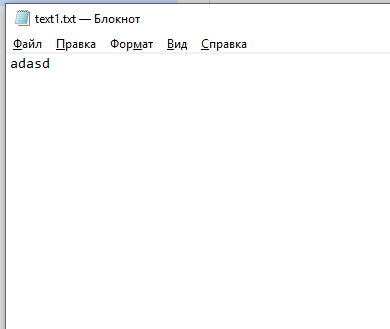


Рисунок 34 – Содержимое файла “text1.txt”

Как можно заметить, содержимое двух файлов отличается. Ниже же представлен пример использования с двумя файлами, через пробел указываем путь до следующего файла.

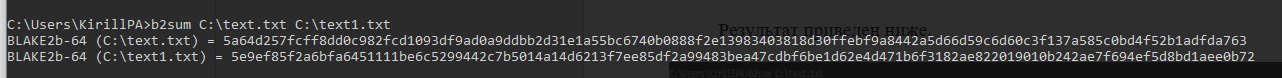


Рисунок 35 – Хеширование двух файлов.

На самом деле, при необходимости можно хешировать и 3 файла одновременно, пример приведен ниже.

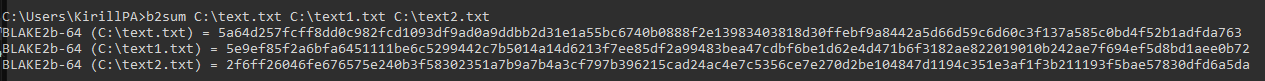


Рисунок 36 – Хеширование трех файлов.

Также команда обладает дополнительным флагом, а котором не упоминалось, а именно – a=ALROTITHM, пример использования можно увидеть ниже.



Рисунок 37 – Хеширование с флагом – a

Этот флаг задает алгоритм хеширования, в данном случае BLKAE2S, от рассматриваемого ранее он отличается тем что, заточен под 8– 32 платформы, в то время как BLAKE2B, используется на 64– ых битных.

# **Заключение**

В ходе выполнения данного проекта были разработаны два приложения для одно для просмотра всех этапов работы алгоритма, другое для генерации хешей, на основе криптографической хеш– функции BLAKE2b, семейства BLAKE. Для разработки данного приложения были использованы такие ЯП как язык Cрешетка и Go, в виду своей мощной базы для работы с битами. Был проведён анализ данного семейства алгоритмов, в результате чего выяснилась ценность данного алгоритма, которая заключается в том, что данный алгоритм гораздо быстрее своих аналогов. Как и у других криптографических алгоритмов, важнейшим достоинством хеш– функций семейства Blake2b перед своими конкурентами, является высокая скорость хеширования, соизмеримая со скоростью поступления входной информации; поэтому, обеспечивается хеширование практически в реальном масштабе времени вне зависимости от объема и разрядности потока преобразуемых данных, конечно это в теории, в наших реалиях на вход может подаваться такой объём данных что предыдущее утверждение может быть ложным в некоторых, исключительных ситуациях.

# **Список используемых источников**

1 Jean– Philippe Aumasson, Samuel Neves , Zooko Wilcox– O’Hearn, & Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5: [Электронный ресурс]. – Virginia: Science: 2013.01.29. URL: https://www.blake2.net/blake2.pdf – Дата доступа 08.05.2022.

2 Википедия [Электронный ресурс] / Wikimedia Foundation Inc. – 2022. – Режим доступа: https://en.wikipedia.org/wiki/BLAKE\_(hash\_function) – Дата доступа: 08.05.2022.

# **ПРИЛОЖЕНИЕ А**

Program.cs

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  class Program  {  private const int NumberOfRounds = 12;    private static readonly int[] ReplaceConstants = new int[NumberOfRounds \* 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, 1, 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  };  static void Round(int roundNum)  {  Console.WriteLine("// ##### Round({0}) #####", roundNum);  /\*0\*/G(roundNum, 0, 0, 4, 8, 12);  /\*1\*/G(roundNum, 1, 1, 5, 9, 13);  /\*2\*/G(roundNum, 2, 2, 6, 10, 14);  /\*3\*/G(roundNum, 3, 3, 7, 11, 15);  /\*4\*/G(roundNum, 4, 0, 5, 10, 15);  /\*5\*/G(roundNum, 5, 1, 6, 11, 12);  /\*6\*/G(roundNum, 6, 2, 7, 8, 13);  /\*7\*/G(roundNum, 7, 3, 4, 9, 14);  Console.WriteLine();  }  static void G(int roundNum, int i, int a, int b, int c, int d)  {  int p = (roundNum << 4) + 2 \* i;  int p0 = ReplaceConstants[p];  int p1 = ReplaceConstants[p + 1];  string s = @"// G(r, i, a, b, c, d)  a = a + b + m[" + p0 + @"];  d ^= a;  d = " + RotateRight("d", 32) + @";  c = c + d;  b ^= c;  b = " + RotateRight("b", 24) + @";  a = a + b + m[" + p1 + @"];  d ^= a;  d = " + RotateRight("d", 16) + @";  c = c + d;  b ^= c;  b = " + RotateRight("b", 63) + @";";  s = s.Replace("a", "v" + a);  s = s.Replace("b", "v" + b);  s = s.Replace("c", "v" + c);  s = s.Replace("d", "v" + d);  s = s.Replace("r", roundNum.ToString());  s = s.Replace("i", i.ToString());  s = s.Replace("\t", "");  Console.WriteLine(s);  Console.WriteLine();  }  static string RotateRight(string name, int offset)  {  return "((" + name + " >>" + offset + ")|(" + name + " << (64– " + offset + ")))";  }  static void Main(string[] args)  {  for (int roundNum = 0; roundNum < NumberOfRounds; roundNum++)  Round(roundNum);  Console.WriteLine();  }  } |

# **ПРИЛОЖЕНИЕ Б**

B2SUM.GO

|  |
| --- |
| package main  import (  "flag"  "fmt"  "hash" )  var (  algoFlag = flag.String("a", "blake2b", "hash algorithm (blake2b, blake2s)")  sizeFlag = flag.Int("s", 0, "digest size in bytes (0 = default)") )  type hashDesc struct {  name string  maxSize int  maker func(size uint8) (hash.Hash, error) }  var algorithms = map[string]hashDesc{  "blake2b": {  "BLAKE2b",  blake2b.Size,  func(size uint8) (hash.Hash, error) { return blake2b.New(&blake2b.Config{Size: size}) },  },  "blake2s": {  "BLAKE2s",  blake2s.Size,  func(size uint8) (hash.Hash, error) { return blake2s.New(&blake2s.Config{Size: size}) },  }, }  func calcSum(f \*os.File, h hash.Hash) (sum []byte, err error) {  h.Reset()  \_, err = io.Copy(h, f)  sum = h.Sum(nil)  return }  func main() {  flag.Parse()   algo, ok := algorithms[\*algoFlag]  if !ok {  flag.Usage()  fmt.Fprintf(os.Stderr, `unsupported algorithm: %s`, algoFlag)  os.Exit(1)  }  if \*sizeFlag == 0 {  \*sizeFlag = algo.maxSize  } else if \*sizeFlag > algo.maxSize {  fmt.Fprintf(os.Stderr, "error: size too large")  os.Exit(1)  }   h, err := algo.maker(uint8(\*sizeFlag))  if err != nil {  fmt.Fprintf(os.Stderr, "error: %s", err)  os.Exit(1)  }   if flag.NArg() == 0 {  // Read from stdin.  sum, err := calcSum(os.Stdin, h)  if err != nil {  fmt.Fprintf(os.Stderr, "%s\n", err)  os.Exit(1)  }  fmt.Printf("%x\n", sum)  os.Exit(0)  }  exitNo := 0  for i := 0; i < flag.NArg(); i++ {  filename := flag.Arg(i)  f, err := os.Open(filename)  if err != nil {  fmt.Fprintf(os.Stderr, "(%s) %s\n", filename, err)  exitNo = 1  continue  }  sum, err := calcSum(f, h)  f.Close()  if err != nil {  fmt.Fprintf(os.Stderr, "(%s) %s\n", filename, err)  exitNo = 1  continue  }  fmt.Printf("%s– %d (%s) = %x\n", algo.name, h.Size(), filename, sum)  }  os.Exit(exitNo) } |