

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ  
федеральное государственное автономное образовательное учреждение высшего образования  
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

ДОПУСТИТЬ К ЗАЩИТЕ

Заведующий кафедрой № 14

К.т.н., доцент

должность, уч. степень, звание

  
2.06.23

подпись, дата

В. Л. Оленов

инициалы, фамилия

БАКАЛАВРСКАЯ РАБОТА

на тему Разработка криптовалюты с применением технологии CryptoNote

выполнена Клейменовым Никитой Александровичем

фамилия, имя, отчество студента в творительном падеже

по направлению подготовки 09.03.01 Информатика и вычислительная техника

код

наименование направления

направленности

01

код


Автоматизированные системы обработки

наименование направленности

информации и управления

наименование направленности

Студент группы № 1941

  
02.06.23

подпись, дата

Н.А. Клейменов

инициалы, фамилия

Руководитель

доц., канд. техн. наук

должность, уч. степень, звание

  
02.06.23

подпись, дата

К.А. Курицын

инициалы, фамилия

Санкт-Петербург 2023

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ  
федеральное государственное автономное образовательное учреждение высшего образования  
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

УТВЕРЖДАЮ

Заведующий кафедрой № 14

К. Г. Н., доцент

должность, уч. степень, звание

 04.04.23

подпись, дата

В.А. Оленев

инициалы, фамилия

ЗАДАНИЕ НА ВЫПОЛНЕНИЕ БАКАЛАВРСКОЙ РАБОТЫ

студенту группы

1941

номер

Клейменову Никите Александровичу

фамилия, имя, отчество

на тему Разработка криптовалюты с применением технологии CryptoNote

утвержденную приказом ГУАП от

05.04.2023

№ 11-369/23

Цель работы: Разработать криптовалютный блокчейн, приложения клиента, приложение криптографического узла.

Задачи, подлежащие решению: Разработать блокчейн с применением технологии CryptoNote, реализовать сетевую библиотеку для взаимодействия с блокчейном,

Реализовать приложение клиента и криптографического узла, создать метод защищенной генерации ключей пользователей

Содержание работы (основные разделы): Проектирование информационной системы, анализ алгоритмов и технологий криптовалюты, реализация программного комплекса

Срок сдачи работы « 16 » июля 2023

Руководитель

доц., канд. техн. наук

должность, уч. степень, звание

 04.04.23

подпись, дата

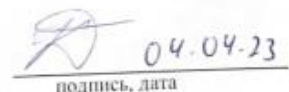
К.А. Курицын

инициалы, фамилия

Задание принял(а) к исполнению

студент группы №

1941

 04.04.23

подпись, дата

Н.А. Клейменов

инициалы, фамилия

# ОГЛАВЛЕНИЕ

<u>ВВЕДЕНИЕ</u> .....	4
<u>1 Проектирование информационной системы</u> .....	6
<u>1.1 Язык программирования Golang</u> .....	6
<u>1.2 Кроссплатформенная база данных SQLite</u> .....	7
<u>2 Анализ алгоритмов и технологий криптовалюты</u> .....	8
<u>2.1 Протокол консенсуса Proof-of-Work (PoW)</u> .....	8
<u>2.2 Алгоритм Cryptonight</u> .....	10
<u>2.3 Алгоритм CryptoNote</u> .....	11
<u>2.4 Алгоритм кольцевых подписей</u> .....	12
<u>2.5 Хэш-функция Blake2b</u> .....	14
<u>2.6 Эллиптическая кривая 25519</u> .....	15
<u>3 Реализация программного комплекса</u> .....	17
<u>3.1 Реализация блокчейна криптовалюты</u> .....	17
<u>3.2 Реализация приложения пользователя криптовалюты</u> .....	52
<u>3.3 Реализация приложения узла криптовалюты</u> .....	59
<u>3.4 Реализация сетевой библиотеки для работы с блокчейном</u> .....	63
<u>ЗАКЛЮЧЕНИЕ</u> .....	68
<u>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ</u> .....	69
<u>ПРИЛОЖЕНИЕ А</u> .....	70
<u>ПРИЛОЖЕНИЕ Б</u> .....	74
<u>ПРИЛОЖЕНИЕ В</u> .....	92
<u>ПРИЛОЖЕНИЕ Г</u> .....	99

## ВВЕДЕНИЕ

В современном мире криптовалюты стали одним из самых актуальных и быстро развивающихся секторов финансовой индустрии. Одним из самых популярных типов криптовалют являются криптовалюты с протоколом Proof-of-Work (PoW), которые используются для майнинга и обеспечения безопасности сети. В рамках данного дипломного проекта рассмотрена разработка криптовалюты с протоколом PoW на основе технологии CryptoNote.

CryptoNote представляет собой протокол, который обеспечивает высокий уровень анонимности и безопасности транзакций за счет использования уникальных алгоритмов шифрования.

Разработка криптовалюты на основе CryptoNote представляет собой актуальную и интересную задачу, которая позволит создать новый продукт, обладающий высоким уровнем безопасности и анонимности, что в свою очередь повысит доверие пользователей к данной криптовалюте и увеличит ее популярность на рынке.

Для того чтобы обеспечить высокий уровень безопасности и анонимности, в разработке криптовалюты на основе CryptoNote будет использовано шифрование на эллиптических кривых. Это позволит обеспечить более высокую степень защиты от криптоанализа и взлома, чем при использовании классических алгоритмов шифрования.

Для обеспечения безопасности транзакций в сети будет использоваться протокол I2P (Invisible Internet Project) – это анонимная сеть, которая позволяет пользователям обмениваться информацией, не раскрывая своей личности и местоположения.

Кроме того, интерес к криптовалютам и блокчейн технологиям в целом, в последние годы растет и в России. В 2019 году Государственная Дума России приняла закон "О цифровых финансовых активах", который

легализовал криптовалюты в стране и определил их статус как цифровые финансовые активы.

Это открыло новые возможности для развития криптовалютной индустрии в России и привлекло внимание предпринимателей и инвесторов. Российские компании и стартапы также активно занимаются разработкой и внедрением блокчейн-решений в различных сферах, от финансов до государственного управления. В связи с этим, разработка криптовалюты на основе CryptoNote может быть интересна для российского рынка и представлять собой перспективный инвестиционный проект.

# **1 Проектирование информационной системы**

## **1.1 Язык программирования Golang**

Go (или Golang) – это язык программирования, разработанный в Google в 2007 году. Он был создан с целью обеспечения высокой производительности, простоты и удобства использования. Go является компилируемым языком, который поддерживает параллельное и конкурентное программирование. Он также имеет сборщик мусора, что облегчает процесс разработки и уменьшает количество ошибок.

Go имеет множество преимуществ, которые делают его идеальным языком для разработки криптовалют. Некоторые из них:

1. **Высокая производительность:** Go имеет очень высокую производительность благодаря своей компиляции и эффективному использованию ресурсов компьютера. Это делает его идеальным для работы с криптографическими алгоритмами и другими вычислительно интенсивными задачами;

2. **Параллельное и конкурентное программирование:** Go имеет удобную модель параллельного и конкурентного программирования, которая позволяет многопоточным приложениям работать более эффективно. Это особенно важно для разработки криптовалют, где многие задачи должны выполняться одновременно;

3. **Простота и удобство использования:** Go имеет простой и лаконичный синтаксис, который снижает количество ошибок и упрощает процесс разработки. Это также делает его более доступным для новичков в программировании;

4. **Большое сообщество разработчиков:** Go имеет большое сообщество разработчиков, которые создают множество библиотек и инструментов для упрощения разработки. Это делает его идеальным языком для разработки криптовалют, которые часто требуют использования множества библиотек и инструментов.

В целом, Go – это отличный язык программирования для разработки криптовалют благодаря своей высокой производительности, удобной модели параллельного и конкурентного программирования, простоте и удобству использования, а также большому сообществу разработчиков.

## **1.2 Кроссплатформенная база данных SQLite**

SQLite – это легковесная база данных, которая может хранить данные локально на устройстве, а также обеспечивает быстрый и эффективный доступ к этим данным. Использование SQLite для хранения блокчейна при разработке криптовалют имеет ряд преимуществ:

1. Эффективность: SQLite быстро и эффективно обрабатывает большие объемы данных, что делает его отличным выбором для хранения блокчейна;
2. Надежность: SQLite имеет высокую степень надежности, поскольку она использует транзакционный подход к хранению данных. Это означает, что в случае сбоя в системе, данные не будут утеряны;
3. Простота: SQLite является простой в использовании базой данных, которая хранит данные в виде файла на устройстве. Это позволяет упростить процесс разработки и управления блокчейном;
4. Низкие затраты: SQLite – бесплатная база данных, что делает ее доступной для использования при разработке криптовалют;
5. Масштабируемость: SQLite легко масштабируется, что позволяет расширять блокчейн при необходимости.

В целом, использование SQLite для хранения блокчейна при разработке криптовалют является хорошим выбором, поскольку она обеспечивает эффективную, надежную и простую в использовании базу данных, что необходимо для блокчейн-технологий.

## **2 Анализ алгоритмов и технологий криптовалюты**

### **2.1 Протокол консенсуса Proof-of-Work (PoW)**

Proof of Work (PoW) – это протокол консенсуса, используемый в блокчейне, который позволяет подтверждать транзакции и создавать новые блоки. Он был изобретен Сатоши Накамото для Bitcoin и стал основой для большинства криптовалют.

Как работает PoW:

1. Майнеры решают математические задачи, чтобы создать новый блок;
2. Эти задачи очень сложны и требуют больших вычислительных мощностей;
3. Когда майнер решает задачу, он создает новый блок и получает вознаграждение в криптовалюте.

Преимущества PoW:

1. Безопасность: PoW обеспечивает высокий уровень безопасности блокчейна. Каждый блок может быть создан только после решения сложной математической задачи, что делает его невозможным для подделки;
2. Децентрализация: PoW позволяет любому желающему стать майнером и участвовать в создании новых блоков. Это обеспечивает децентрализацию блокчейна и предотвращает монополизацию майнинга;
3. Экономическая стимуляция: PoW предоставляет майнерам вознаграждение за создание новых блоков. Это стимулирует майнеров продолжать работу и поддерживать блокчейн.





Рисунок 1 – Защищённая блокчейн сеть

Существует несколько протоколов консенсуса, используемых в блокчейне, но PoW имеет несколько преимуществ перед другими протоколами:

1. **Безопасность:** PoW обеспечивает высокий уровень безопасности блокчейна. Каждый блок может быть создан только после решения сложной математической задачи, что делает его невозможным для подделки. Это отличает PoW от других протоколов консенсуса, таких как Proof of Stake (PoS), где участники могут ставить свою ставку на подтверждение транзакций и создание новых блоков, что может привести к возможности атаки на блокчейн;

2. **Децентрализация:** PoW позволяет любому желающему стать майнером и участвовать в создании новых блоков. Это обеспечивает децентрализацию блокчейна и предотвращает монополизацию майнинга. В

то же время, другие протоколы, такие как PoS, могут быть склонны к централизации майнинга в руках небольшого числа крупных участников;

3. Экономическая стимуляция: PoW предоставляет майнерам вознаграждение за создание новых блоков. Это стимулирует майнеров продолжать работу и поддерживать блокчейн. В других протоколах, таких как Delegated Proof of Stake (DPoS), вознаграждения распределяются между участниками блокчейна, что может не стимулировать их на работу.

## **2.2 Алгоритм Cryptonight**

Cryptonight – это алгоритм хэширования, который используется в криптовалютах, таких как Monero, Bytecoin и другие. Он был разработан для обеспечения конфиденциальности и анонимности транзакций, а также для повышения устойчивости к ASIC-майнингу.

Преимущество Cryptonight заключается в том, что он использует алгоритмы, которые способны эффективно работать на центральных процессорах (CPU) и графических процессорах (GPU), а не на специализированных майнерах ASIC. Это позволяет обеспечить более децентрализованную сеть, поскольку майнинг может быть выполнен на обычных компьютерах, а не только на дорогостоящих специализированных устройствах.

Кроме того, Cryptonight обеспечивает высокую степень конфиденциальности и анонимности транзакций. Это достигается путем использования различных методов, таких как ринговые подписи и скрытые адреса, которые делают транзакции невозможными для отслеживания. В целом, использование Cryptonight позволяет создавать более безопасные, конфиденциальные и устойчивые криптовалюты, которые могут быть добыты на обычных компьютерах.

## 2.3 Алгоритм CryptoNote

Алгоритм CryptoNote является алгоритмом шифрования, который используется в криптовалюте Monero и других подобных монетах. Он разработан для обеспечения максимальной анонимности пользователей и защиты от атак 51%.

Сам алгоритм использует механизмы шифрования, которые делают транзакции не отслеживаемыми. Каждая транзакция в сети CryptoNote защищена от прослушивания и анализа, что позволяет пользователям сохранять конфиденциальность своих операций.

Для того чтобы сделать транзакцию, пользователь генерирует новый адрес и отправляет свои деньги на этот адрес. При этом, вся информация о транзакции и адресе получателя шифруется, что делает ее нечитаемой для посторонних.

Кроме того, все транзакции в сети CryptoNote используют кольцевую подпись. Это означает, что в каждой транзакции участвуют несколько адресов, из которых нельзя определить, какой именно отправитель.

Для того, чтобы обеспечить безопасность сети, алгоритм CryptoNote использует доказательство выполнения работы (PoW). Это означает, что для создания новых блоков в сети нужно потратить некоторый вычислительный ресурс. Это делает сеть более защищенной от атак 51%.

В целом, алгоритм CryptoNote обеспечивает максимальную анонимность и безопасность пользователям, что делает его одним из самых популярных алгоритмов в криптовалютной индустрии.

Алгоритм CryptoNote работает следующим образом:

1. Создание уникального адреса. Пользователь генерирует свой уникальный адрес, который и будет использоваться для отправки и получения криптовалюты в сети CryptoNote;
2. Защита данных о транзакции. Когда пользователь отправляет транзакцию, данные о ней шифруются при помощи криптографических алгоритмов, что делает их нечитаемыми для посторонних;

3. Кольцевая подпись. Каждая транзакция в сети CryptoNote содержит кольцевую подпись, благодаря которой нельзя определить, какой именно адрес отправил деньги. В каждой кольцевой подписи участвует несколько адресов, выбранных случайным образом;

4. Проверка транзакции. Каждая транзакция в сети CryptoNote проходит проверку на соответствие правилам протокола. Это включает в себя проверку наличия необходимых средств на счету отправителя, а также наличие корректных подписей;

5. Доказательство выполнения работы (PoW). Для создания новых блоков в сети CryptoNote используется алгоритм доказательства выполнения работы. Это означает, что майнеры должны потратить значительное количество вычислительной мощности для создания нового блока, что делает сеть более защищенной от атак 51%;

6. Обновление блокчейна. Когда новый блок создан, он добавляется в блокчейн, который хранит всю историю транзакций в сети CryptoNote. Обновление блокчейна происходит автоматически и не требует дополнительных действий от пользователей;

7. Получение криптовалюты. Когда пользователь получает криптовалюту в сети CryptoNote, он может использовать ее для оплаты товаров и услуг, а также для отправки другим пользователям.

## **2.4 Алгоритм кольцевых подписей**

Алгоритм кольцевых подписей (Ring Signatures) является криптографическим методом, который позволяет доказать подпись документа или транзакции, не раскрывая личность подписавшего. Это достигается путем создания группы подписантов, из которых только один является подписавшим. Но при этом невозможно определить, кто именно из группы создал подпись, что делает алгоритм кольцевых подписей особенно полезным для разработки криптовалют.

Алгоритм кольцевых подписей используется в криптовалютах для обеспечения конфиденциальности транзакций. Он позволяет скрыть отправителя и получателя транзакции, а также сумму перевода. Это делает криптовалютную транзакцию более анонимной и защищенной от взлома.

Процесс создания кольцевой подписи включает следующие шаги:

1. Генерация ключей. Каждый участник создает свой ключ и передает его другим участникам группы;
2. Создание подписи. Подписывающий выбирает случайного участника из группы и создает подпись, используя свой собственный ключ и публичные ключи других участников;
3. Проверка подписи. Получатель транзакции может проверить подпись, используя публичные ключи всех участников группы.

Преимущества использования алгоритма кольцевых подписей для разработки криптовалют:

1. Анонимность. Алгоритм позволяет скрыть личность отправителя и получателя транзакции, что делает ее более анонимной;
2. Защита от взлома. Кольцевые подписи защищают транзакцию от взлома, так как злоумышленник не может определить, кто именно из группы создал подпись;
3. Безопасность. Алгоритм обеспечивает безопасность транзакций, так как позволяет проверить подпись без раскрытия личности подписавшего;
4. Эффективность. Алгоритм кольцевых подписей является эффективным способом обеспечения конфиденциальности и безопасности транзакций в криптовалютах.

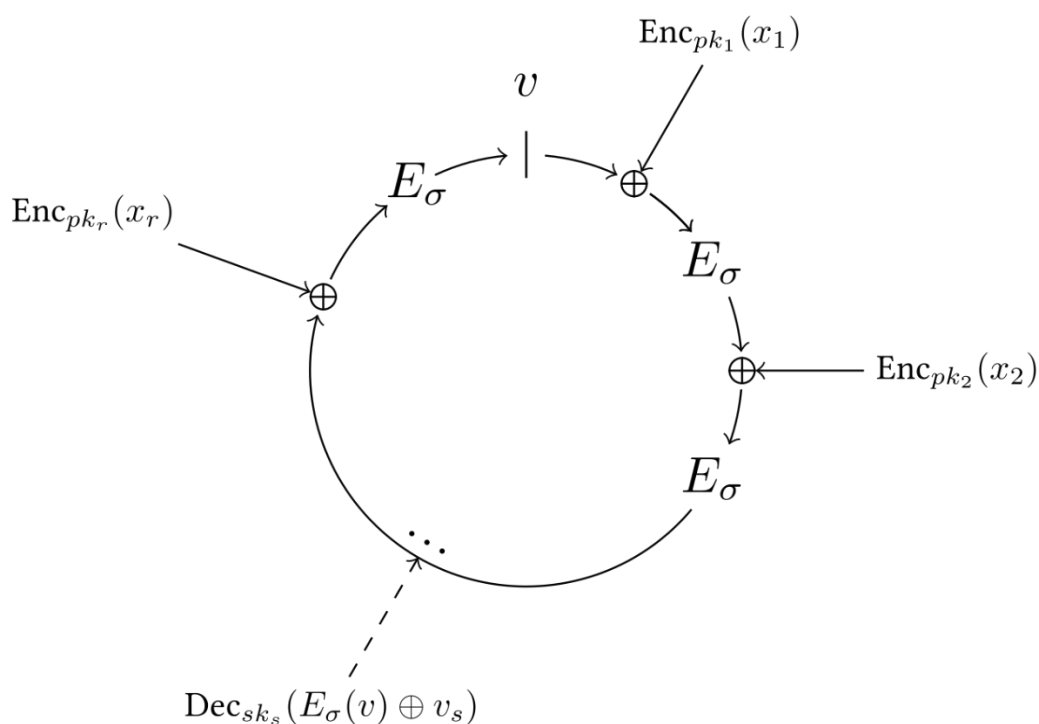


Рисунок 2 – Схема кольцевой подписи

Таким образом, алгоритм кольцевых подписей является надежным и эффективным инструментом, который обеспечивает конфиденциальность и безопасность транзакций в криптовалютах. Он позволяет создать анонимную транзакцию, которая защищена от взлома и проверена на безопасность.

## 2.5 Хэш-функция Blake2b

Blake2b – это хэш-функция, которая используется в криптографии и криптовалютах. Она является улучшенной версией алгоритма Blake, который был разработан в 2011 году и имел проблемы с безопасностью. Blake2b был разработан, чтобы решить эти проблемы и обеспечить безопасность и эффективность в криптографии.

Одним из преимуществ Blake2b является его высокая скорость хэширования. Он может обрабатывать данные на скорости более 1 GB в секунду на современных процессорах. Это позволяет использовать Blake2b для быстрой проверки целостности данных, а также для различных задач, связанных с безопасностью и шифрованием.

Кроме того, Blake2b является безопасным алгоритмом, который обеспечивает высокую степень защиты от атак. Он использует 64-битное слово и 12 раундов шифрования, что делает его очень сложным для взлома. Blake2b также имеет высокую стойкость к коллизиям, что означает, что вероятность обнаружения двух разных сообщений, дающих одинаковый хэш-код, крайне мала.

## **2.6 Эллиптическая кривая 25519**

Эллиптическая кривая 25519 – это криптографический алгоритм, который используется для шифрования и подписи данных. Он базируется на эллиптических кривых, которые являются математическими объектами, определенными на плоскости, и представляют собой множество точек на этой плоскости, удовлетворяющих определенному уравнению.

Одним из преимуществ эллиптической кривой 25519 является их высокая степень безопасности. Они обеспечивают высокую степень защиты от различных видов атак, включая атаки на основе линейных и дифференциальных криптоанализов. Кроме того, эллиптические кривые 25519 обеспечивают высокую степень стойкости к атакам на основе комбинирования, что делает их идеальным выбором для различных криптографических задач.

Еще одним преимуществом эллиптической кривой 25519 является их высокая скорость работы. Они могут быть использованы для шифрования и подписи данных с высокой скоростью, что делает их идеальным выбором для разработки криптовалют.

Эллиптическая кривая 25519 также обеспечивают высокую степень конфиденциальности и анонимности транзакций в криптовалютах. Она может быть использована для создания уникальных ключей шифрования для каждой транзакции, что делает их невозможными для отслеживания и взлома.

В целом, эллиптическая кривая 25519 являются безопасным и эффективным инструментом в криптографической экосистеме.



### 3 Реализация программного комплекса

#### 3.1 Реализация блокчейна криптовалюты

Блокчейн – это распределенный реестр, который использует криптографические методы для записи и подтверждения транзакций. Это означает, что информация в блокчейне хранится на нескольких компьютерах, которые работают вместе, чтобы подтвердить транзакции и обеспечить безопасность.

Блокчейн используется для создания децентрализованных приложений, которые не нуждаются в централизованном управлении, таких как криптовалюты, смарт-контракты и другие приложения. Блокчейн также позволяет создавать устойчивые, прозрачные и безопасные системы для хранения и передачи данных и информации.

Для построения блокчейна были описаны 3 структуры

```
type Blockchain struct {  
    DB *sql.DB  
}
```

Рисунок 3 – Структура для хранения блокчейна

В данной структуре содержится указатель на базу данных для удобной работы с ней и выполнения таких операций как запись и чтение.

```

type Block struct { 25 usages
    Nonce          uint64
    Difficulty      uint8
    CurrHash        []byte
    PrevHash        []byte
    Transactions    []Transaction
    Mapping          map[string]uint64
    Miner           string
    Signature        []byte
    TimeStamp       string
}

```

Рисунок 4 – Структура блока

В данной структуре описывается блок цепи:

- Nonce – Результат подтверждения работы
- Difficulty – Сложность блока
- CurrHash – Хеш текущего блока
- PrevHash – Хеш предыдущего блока
- Transactions – Транзакции пользователей
- Mapping – Состояние баланса пользователей
- Miner – Пользователь замайнивший блок
- Signature – Подпись майнера указывающая на CurrHash
- TimeStamp – Метка времени создания блока

```
type Transaction struct {
    RandBytes []byte
    PrevBlock []byte
    Sender     string
    Receiver   string
    Value      uint64
    CurrHash   []byte
    Signature  []byte
    Emission   uint64
}
```

Рисунок 5 – Структура транзакции

В данной структуре описывается структура транзакций, совершаемых внутри блокчейна:

- RandBytes – Случайные байты
- PrevBlock – Хеш последнего блока в блокчейне
- Sender – Имя отправителя
- Receiver – Имя получателя
- Value – Количество переводимых денег получателю
- CurrHash – хеш текущей транзакции
- Signature – подпись отправителя
- Emission – показатель эмиссии

```

func NewChain(filename, receiver string) error { 1 usage
    file, err := os.Create(filename)
    if err != nil { err }
    file.Close()
    db, err := sql.Open( driverName: "sqlite3", filename)
    if err != nil { err }
    defer db.Close()
    _, err = db.Exec(CREATE_TABLES)

    chain := &Blockchain{
        DB: db,
    }
    genesis := &Block{
        PrevHash: []byte(GENESIS_BLOCK),
        Mapping:   make(map[string]uint64),
        Miner:     receiver,
        Timestamp: time.Now().Format(time.RFC3339),
    }

    genesis.Mapping[receiver] = GENESIS_REWARD
    genesis.CurrHash = genesis.hash()
    chain.AddBlock(genesis)
    return nil
}

```

Рисунок 6 – Функция создание новой цепи блокчейн

Функция NewChain получает на вход имя файла для сохранения и получателя блока создаёт новую блокчейн цепочку и сохраняет её в файле с указанным именем.

Функция возвращает ошибку, если что-то пошло не так. Сначала функция создает файл с указанным именем.

Затем создается новая база данных SQLite с именем файла и открыта с помощью sql.Open. Если возникает ошибка, то функция возвращает её. После выполнения операции базы данных закрывается с помощью defer.

Затем функция создает таблицы в базе данных с помощью запроса CREATE\_TABLES. Если возникает ошибка при выполнении запроса, то функция возвращает её.

```
const ( 1 usage
    CREATE_TABLES = ` 1 usage
    CREATE TABLE Blockchain (
        Id INTEGER PRIMARY KEY AUTOINCREMENT,
        Hash VARCHAR(44) UNIQUE,
        Block TEXT
    );
`
)
```

Рисунок 7 – Запрос” CREATE\_TABLES”

Далее функция создает новый экземпляр блокчейн цепочки и экземпляр генезис-блока. Генезис-блок прописывается вручную и является первым блоком блокчейна. Он нужен для начала работы цепи. Временная метка показывает время создания блока в формате RFC3339.

Затем генезис-блок добавляется в блокчейн цепочку, после чего функция возвращает nil, если всё прошло успешно.

```
func (chain *Blockchain) AddBlock(block *Block) { 1 usage
    chain.DB.Exec( query: "INSERT INTO Blockchain (Hash, Block) VALUES ($1, $2)",
        Base64Encode(block.CurrHash),
        SerializeBlock(block),
    )
}
```

Рисунок 8 – Функция добавления блока в блокчейн

Функция AddBlock добавляет новый блок в блокчейн.

Первым аргументом передается указатель на объект блокчейн, к которому будет добавлен блок. Вторым аргументом передается указатель на добавляемый блок.

Функция использует метод Exec объекта DB, который выполняет SQL-запрос INSERT INTO для добавления новой записи в таблицу Blockchain.

В качестве значений для добавления используются две переменные:

– Base64Encode(block.CurrHash) – хеш текущего блока, закодированный в формате Base64;

```
func SerializeBlock(block *Block) string { 2 usages
    jsonData, err := json.MarshalIndent(*block, prefix: "", indent: "\t")
    if err != nil {
        return ""
    }
    return string(jsonData)
}
```

Рисунок 9 – Функция сериализации данных

Функция `SerializeBlock` принимает указатель на объект типа `Block` и сериализует его в формат JSON.

Сначала функция вызывает функцию `MarshalIndent` из пакета `encoding/json`, которая преобразует объект `Block` в строку JSON с отступами для удобного чтения. В качестве аргументов функции передаются указатель на объект `Block`, пустая строка для префикса и символ табуляции для отступов.

Если при выполнении функции происходит ошибка, то возвращается пустая строка.

В конце функция возвращает строку, содержащую сериализованный объект `Block` в формате JSON.

```
func Base64Encode(data []byte) string { 5 usages
    return base64.StdEncoding.EncodeToString(data)
}
```

Рисунок 10 – Функция кодирование данных в строку

Функция `Base64Encoded` принимает на вход массив байтов (`data`), который нужно закодировать в строку в формате Base64.

Для этого функция использует стандартную кодировку base64.StdEncoding, которая предоставляет функционал для кодирования и декодирования строк в формате Base64.

```
func LoadChain(filename string) *BlockChain { 3 usages
    db, err := sql.Open(driverName: "sqlite3", filename)
    if err != nil : nil
    chain := &BlockChain{
        DB: db,
    }
    return chain
}
```

Рисунок 11 – Функция загрузки блокчейна

Функция LoadChain загружает цепочку блоков из базы данных SQLite, расположенной в файле с именем, указанным в аргументе 'filename'.

Сначала функция открывает подключение к базе данных SQLite с помощью функции sql.Open(). Если при этом происходит ошибка, функция возвращает значение 'nil'.

Затем функция создает экземпляр структуры BlockChain, используя указатель на открытое подключение к базе данных, и возвращает его в качестве результата.

```
func NewBlock(miner string, prevHash []byte) *Block {
    return &Block{
        Difficulty: DIFFICULTY,
        PrevHash:   prevHash,
        Miner:     miner,
        Mapping:    make(map[string]uint64),
    }
}
```

Рисунок 12 – Функция создание нового блока

Функция `NewBlock` создает новый блок в блокчейне с заданными параметрами.

Первый аргумент функции – `miner`, является строкой, и представляет собой адрес майнера, который добыл данный блок.

Второй аргумент функции – `prevHash`, является массивом байтов, и представляет собой хэш предыдущего блока в цепочке блоков.

```
func NewTransaction(user *User, lasthash []byte, to string, value uint64, size string) *Transaction {
    sizeInt, _ := strconv.ParseUint(size, base: 10, bitSize: 64)
    tx := &Transaction{
        RandBytes: GenerateRandomBytes(RAND_BYTES),
        PrevBlock: lasthash,
        Sender:    user.Address(),
        Receiver:  to,
        Value:     value,
        Emission:  emission(sizeInt),
    }
    tx.CurrHash = tx.hash()
    tx.Signature = tx.sign(user.Private())
    return tx
}
```

Рисунок 13 – Функция создание новой транзакции

Функция `NewTransaction` создает новую транзакцию (`Transaction`) в блокчейне. Входными параметрами функции являются указатель на объект пользователя (`User`), хеш предыдущего блока (`lasthash`), адрес получателя (`to`), количество единиц (`value`) и размер (`size`) транзакции в строковом формате.

Функция начинается с преобразования размера транзакции из строкового формата в беззнаковое целое число (`sizeInt`) с помощью функции `strconv.ParseUint`. Затем создается новый объект транзакции (`tx`) со следующими полями:

- `RandBytes`: случайные байты, которые используются для создания хеша транзакции;
- `PrevBlock`: хеш предыдущего блока;
- `Sender`: адрес отправителя (адрес пользователя);



- Receiver: адрес получателя;
- Value: количество единиц, которые будут переданы от отправителя получателю;
- Emission: количество единиц, которые будут выпущены в результате транзакции (вычисляется на основе размера транзакции).

Затем вычисляется хеш транзакции (tx.CurrHash) и создается подпись (tx.Signature) с использованием закрытого ключа пользователя (user.Private()).

В конце функция возвращает созданный объект транзакции (tx).

```
func emission(BlockChainSize uint64) uint64 {
    if BlockChainSize <= 1024 {
        return 1000
    } else if BlockChainSize >= 9216 {
        return 100
    } else {
        return 1000 - (BlockChainSize/1024)*100
    }
}
```

Рисунок 14 – Функция расчета эмиссии

Функция emission реализует расчет эмиссии монет (стоимость за создание блока) первые 1024 блока награда 1000, затем каждые 1024 блока она будет снижаться на 100. Когда в блокчейне станет 9216 блоков, награда станет постоянной и будет равна 100 монетам

```
type User struct {
    PublicKey *ed25519.PublicKey
    PrivateKey *ed25519.PrivateKey
}
```

Рисунок 15 – Структура пользователя

Структура User описывает поля пользователя криптовалюты в ней содержится 2 поля PublicKey и PrivateKey в первом содержится публичный ключ пользователя, а во втором приватный.

```
func GenerateRandomBytes(max uint) []byte {  
    var slice []byte = make([]byte, max)  
    _, err := rand.Read(slice)  
    if err != nil : nil  
    return slice  
}
```

Рисунок 16 – Функция генерации случайный байтов

Функция GenerateRandomBytes генерирует случайную последовательность байтов заданной длины max и возвращает ее в виде среза []byte для повышения уникальности и безопасности транзакции.

```
func (user *User) Address() string {  
    return StringPublic(user.Public())  
}
```

Рисунок 17 – Функция возврата публичного адреса пользователя

Функция Address возвращает адрес пользователя, который является публичным ключом.

```
func (user *User) Private() *ed25519.PrivateKey {  
    return user.PrivateKey  
}
```

Рисунок 18 – Функция возврата приватного ключа

Функция Private возвращает адрес пользователя, который является публичным ключом.

```
func (user *User) Public() *ed25519.PublicKey {
    return user.PublicKey
}
```

Рисунок 19 – Функция возврата приватного ключа

Функция Public возвращает публичный ключ пользователя.

```
func (tx *Transaction) hash() []byte {
    return CnFastHash(bytes.Join(
        [][]byte{
            tx.RandBytes,
            tx.PrevBlock,
            []byte(tx.Sender),
            []byte(tx.Receiver),
            ToBytes(tx.Value),
        },
        []byte{}),
    ))
}
```

Рисунок 20 – Функция вычисления хэш-суммы транзакции

Функция hash представляет собой метод, который вычисляет хэш-сумму транзакции. Для вычисления хэш-суммы используется функция CnFastHash. В качестве аргумента функции передается результат объединения пяти массивов байтов.

Первый массив – это случайные байты, сгенерированные при создании транзакции. Второй массив – это хэш-сумма предыдущего блока в блокчейне. Третий массив – это адрес отправителя транзакции, преобразованный в байты. Четвертый массив – это адрес получателя транзакции, также преобразованный в байты. Пятый массив – это числовое значение, представляющее сумму переводимых средств, преобразованное в байты.

В итоге, данная функция вычисляет уникальную хэш-сумму для каждой транзакции в блокчейне.

```
func (tx *Transaction) sign(priv *ed25519.PrivateKey) []byte {
    return Sign(priv, tx.CurrHash)
}
```

Рисунок 21 – Функция подписи транзакции

Функция `sign` используется для подписи текущей транзакции приватным ключом, что обеспечивает ее подлинность и целостность. Для подписи используется функция `Sign()`, которая принимает приватный ключ и хэш, и возвращает подпись в виде среза байтов.

```
func StringPublic(pub *ed25519.PublicKey) string {
    keyBytes := make([]byte, ed25519.PrivateKeySize)
    copy(keyBytes[:], *pub)
    return base64.StdEncoding.EncodeToString(keyBytes)
}
```

Рисунок 22 – Функция преобразования публичного ключа в строку

Функция `StringPublic` создает срез байтов `keyBytes` длиной `ed25519.PrivateKeySize` (равной 32 байтам) с помощью функции `make`. Затем она копирует первые 32 байта `ed25519.PrivateKeySize` указателя `pub` в срез `keyBytes` с помощью функции `copy`.

Далее функция кодирует срез байтов `keyBytes` в формате `base64` и возвращает его в виде строки с помощью метода `EncodeToString`.

В итоге функция `StringPublic` возвращает строковое представление открытого ключа в формате `base64`.

```
func CnFastHash(data []byte) []byte {
    out := blake2b.Sum256(data)
    result := make([]byte, 4)
    binary.LittleEndian.PutUint32(result, binary.LittleEndian.Uint32(out[0:4]))
    return result
}
```

Рисунок 23 – Функция вычисления хеша с помощью алгоритма CryptoNight

Функция CnFastHash вычисляется хэш с помощью алгоритма CryptoNight, который использует функцию blake2b для вычисления хэша и дополнительно преобразует результат с помощью функции binary.LittleEndian. В конечном итоге, программа выводит хэш в шестнадцатеричном формате.

```
func ToBytes(num uint64) []byte { 7 usages
    var data = new(bytes.Buffer)
    err := binary.Write(data, binary.BigEndian, num)
    if err != nil : nil
    return data.Bytes()
}
```

Рисунок 24 – Функция преобразования числа в последовательность байт

Функция ToBytes принимает на вход беззнаковое 64–битное целое число типа uint64 и возвращает его представление в виде среза байтов ([]byte).

Для этого функция создает новый буфер данных типа bytes.Buffer и записывает в него полученное число в формате BigEndian (порядок следования байтов, при котором старший байт записывается первым).

Если при записи произошла ошибка, функция возвращает nil.

Затем функция возвращает срез байтов, полученный из буфера данных, содержащий представление входного числа в виде последовательности байтов.

```

func Sign(privKey *ed25519.PrivateKey, data []byte) []byte { 2 usages
    ringKeys := make([]ed25519.PrivateKey, 5)
    for i := 0; i < 5; i++ {
        randKey := make([]byte, 32)
        rand.Read(randKey)
        ringKeys[i] = ed25519.NewKeyFromSeed(randKey)
    }
    var pubKeys []ed25519.PublicKey
    for i := 0; i < 5; i++ {
        pubKey := ed25519.PrivateKey(ringKeys[i].Seed()).Public().(ed25519.PublicKey)
        pubKeys = append(pubKeys, pubKey)
    }
    var signature []byte
    for i := range pubKeys {
        tmpKeys := make([]ed25519.PrivateKey, 5)
        copy(tmpKeys, ringKeys)
        tmpKeys[i] = *privKey
        if len(tmpKeys[i]) != ed25519.PrivateKeySize {
            continue
        }
        tmpSignature := ed25519.Sign(tmpKeys[i], data)
        signature = append(signature, tmpSignature[:]...)
    }
    return signature
}

```

Рисунок 25 – Функция создания кольцевой подписи

Функция Sign создает кольцевую подпись для переданных данных с использованием алгоритма Ed25519. Входными параметрами является указатель на приватный ключ и данные, которые нужно подписать.

В функции создается массив из 5 приватных ключей, генерируемых с помощью случайных значений. Затем для каждого приватного ключа создается соответствующий ему публичный ключ, которые добавляются в массив pubKeys.

Далее происходит подписание данных с использованием всех публичных ключей из массива pubKeys. Для этого в цикле проходится по каждому публичному ключу и создается временный массив из 5 приватных ключей. В этот массив копируются все сгенерированные ранее приватные

ключи, за исключением текущего, который заменяется на переданный в функцию приватный ключ.

Если длина текущего приватного ключа не соответствует длине ключа Ed25519, то происходит пропуск текущей итерации цикла. Если длина ключа корректна, то с помощью функции `ed25519.Sign` создается временная подпись для данных, используя текущий приватный ключ. Эта подпись добавляется в массив `signature`.

По завершении цикла возвращается массив `signature`, содержащий все временные подписи, созданные с помощью всех публичных ключей. Таким образом, создается кольцевая подпись для переданных данных, которая позволяет скрыть конкретный приватный ключ, используемый для подписи.

```
func (block *Block) AddTransaction(chain *BlockChain, tx *Transaction) error {
    if tx == nil {
        return errors.New( text: "tx is null")
    }
    if tx.Value == 0 {
        return errors.New( text: "tx value = 0")
    }
    if tx.Emission != emission(chain.Size()) {
        return errors.New( text: "emission value is invalid")
    }
    if !bytes.Equal(tx.PrevBlock, chain.LastBlockHash()) {
        return errors.New( text: "prev block in tx /= last hash in chain")
    }
    balanceInChain := chain.Balance(tx.Sender, chain.Size())
    balanceInTX := tx.Value + tx.Emission
    if balanceInTX > balanceInChain {
        return errors.New( text: "insufficient funds")
    }
    block.Mapping[tx.Sender] = balanceInChain - balanceInTX
    block.addBalance(chain, tx.Receiver, tx.Value)
    block.addBalance(chain, tx.Receiver, tx.Emission)
    block.Transactions = append(block.Transactions, *tx)
    return nil
}
```

Рисунок 26 – Функция добавления транзакции в блок

Данная функция предназначена для добавления транзакции в блок.

Функция принимает указатель на блок (block) и указатели на цепочку блоков (chain) и транзакцию (tx).

В начале функция проверяет, что переданная транзакция не является пустой и имеет ненулевое значение). Также проверяется, что значение эмиссии в транзакции соответствует эмиссии текущей цепочки блоков. Если хотя бы одно из этих условий не выполняется, то функция возвращает ошибку.

Далее функция проверяет, что хэш предыдущего блока в транзакции соответствует хэшу последнего блока в цепочке блоков. Если не выполняется, то функция возвращает ошибку.

Затем функция проверяет, достаточно ли средств у отправителя для осуществления транзакции. Если нет, то функция возвращает ошибку.

Если все проверки пройдены успешно, то функция обновляет баланс отправителя и получателя в блоке. Затем транзакция добавляется в список транзакций блока.

В конце функция возвращает nil, если все прошло успешно, или ошибку, если хотя бы одна проверка не прошла.

```
func (chain *BlockChain) LastBlockHash() []byte { 1 usage
    var lastBlockHash []byte
    err := chain.DB.QueryRow( query: "SELECT Hash FROM BlockChain ORDER BY Id DESC LIMIT 1").Scan(&lastBlockHash)
    if err != nil {
        log.Panic(err)
    }
    return lastBlockHash
}
```

Рисунок 27 – Функция добавления транзакции в блок

Функция LastBlockHash возвращает хеш последнего блока в блокчейне.

Сначала определяется переменная lastBlockHash типа []byte, которая будет хранить хеш последнего блока. Затем выполняется запрос к базе данных, который выбирает хеш из таблицы BlockChain, сортирует записи по убыванию Id (идентификатор блока) и выбирает только одну запись с



помощью LIMIT 1. Результат запроса записывается в переменную lastBlockHash.

Если при выполнении запроса возникает ошибка, то программа останавливается.

Наконец, функция возвращает переменную lastBlockHash, содержащую хеш последнего блока в блокчейне.

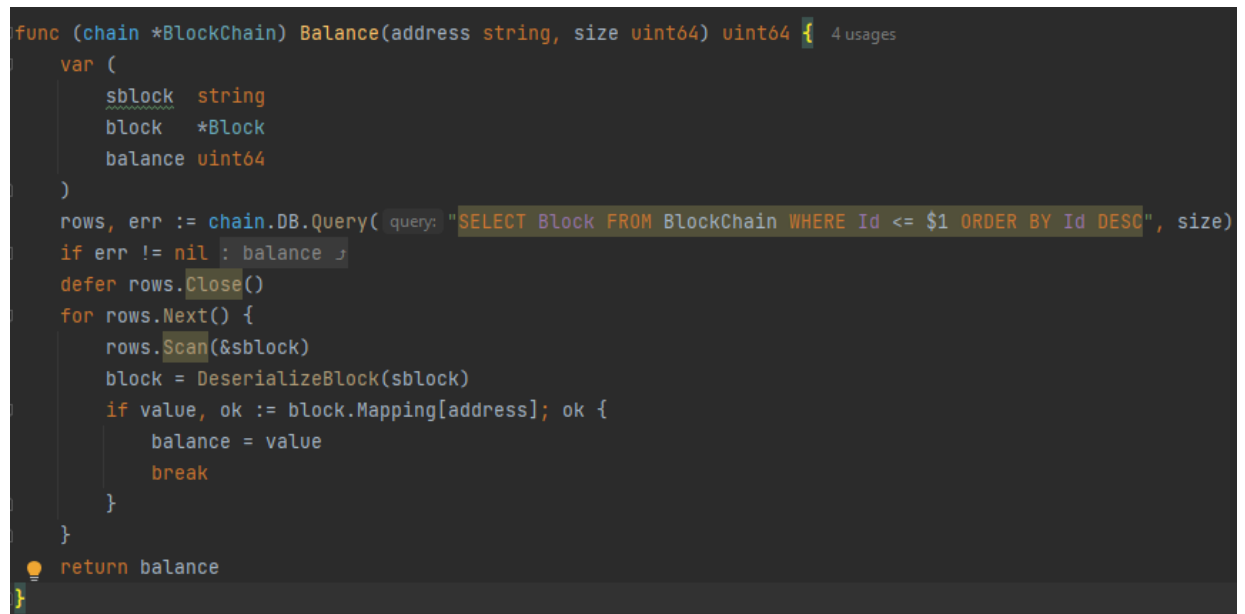
A screenshot of a code editor showing the implementation of a Go function named `Balance`. The function signature is `func (chain *BlockChain) Balance(address string, size uint64) uint64`. Inside the function, a `var` block declares `sblock string`, `block *Block`, and `balance uint64`. The code then executes a database query: `rows, err := chain.DB.Query(query: "SELECT Block FROM BlockChain WHERE Id <= $1 ORDER BY Id DESC", size)`. It checks for errors and uses `defer rows.Close()`. A `for` loop with `rows.Next()` iterates through the results. Inside the loop, `rows.Scan(&sblock)` is used to scan the data into `sblock`, followed by `block = DeserializeBlock(sblock)`. An `if` statement checks if the block contains the address: `if value, ok := block.Mapping[address]; ok {`. If so, it updates `balance = value` and breaks the loop. Finally, it returns `balance`.

Рисунок 28 – Функция анализа баланса пользователя

Функция `Balance` направляет запросы в блокчейн, затем происходит итерация по строкам результата запроса. Для каждой строки вызывается метод `Scan`, который записывает значение поля `Block` в переменную `sblock`. Затем вызывается функция `DeserializeBlock`, которая десериализует блок из строки `sblock` и записывает его в переменную `block`. Если в блоке есть запись для указанного адреса (`address`), то значение баланса (`value`) записывается в переменную `balance` и происходит выход из цикла.

В конце функция возвращает текущее значение переменной `balance`.

```
func (block *Block) addBalance(chain *BlockChain, receiver string, value uint64) {
    var balanceInChain uint64
    if v, ok := block.Mapping[receiver]; ok {
        balanceInChain = v
    } else {
        balanceInChain = chain.Balance(receiver, chain.Size())
    }
    block.Mapping[receiver] = balanceInChain + value
}
```

Рисунок 29 – Функция добавление суммы на баланс получателя

Сначала функция addBalance ищет текущий баланс получателя в блоке block с помощью поиска по ключу receiver в map block.Mapping. Если он найден, то значение баланса сохраняется в переменной balanceInChain. Если не найден, то баланс получателя ищется в цепочке блоков chain с помощью вызова метода Balance с аргументами receiver и chain.Size(), который возвращает текущий баланс получателя.

Затем в map block.Mapping по ключу receiver сохраняется сумма текущего баланса и добавляемой суммы value.

Таким образом, функция обновляет баланс получателя в блоке block и сохраняет эту информацию в map block.Mapping для последующего использования.

```
func (chain *BlockChain) Size() uint64 {
    var size uint64
    row := chain.DB.QueryRow( query: "SELECT Id FROM BlockChain ORDER BY Id DESC")
    row.Scan(&size)
    return size
}
```

Рисунок 30 – Функция получения размера блокчейна

Функция Size делает запрос к базе данных и возвращает id последнего блока тем самым узнавая размер всей цепочки

```
func DeserializeBlock(data string) *Block { 5 usages
    var block Block
    err := json.Unmarshal([]byte(data), &block)
    if err != nil : nil
    return &block
}
```

Рисунок 31 – Функция десериализации данных

Функция `DeserializeBlock` принимает на вход строку с данными в формате JSON, которые представляют собой информацию о блоке. Затем функция десериализует (преобразует из JSON в объект) эту строку и заполняет поля структуры `Block` с помощью функции `json.Unmarshal()`.

```
func (block *Block) Accept(chain *BlockChain, user *User, ch chan bool) error {
    if !block.transactionsIsValid(chain, chain.Size()) {
        return errors.New(text: "transactions is not valid")
    }
    block.AddTransaction(chain, &Transaction{
        RandBytes: GenerateRandomBytes(RAND_BYTES),
        PrevBlock: chain.LastHash(),
        Sender:    "",
        Receiver:  user.Address(),
        Value:     chain.Size(),
    })
    block.Timestamp = time.Now().Format(time.RFC3339)
    block.CurrHash = block.hash()
    block.Signature = block.sign(user.Private())
    block.Nonce = block.proof(ch)
    return nil
}
```

Рисунок 32 – Функция подтверждения транзакции

Функция `Асерт` принимает на вход указатель на блок (`Block`), указатель на цепочку блоков (`BlockChain`), указатель на пользователя (`User`) и канал (`ch`) типа `bool`. Функция выполняет следующие действия:

- Проверяет, являются ли транзакции в блоке действительными, с помощью метода `transactionsIsValid()`.
- Если транзакции не действительны, то функция возвращает ошибку `"transactions is not valid"`.

- Если транзакции действительны, то функция добавляет новую транзакцию в блок с помощью метода `AddTransaction()`. Новая транзакция содержит случайно сгенерированные байты (`RandBytes`), хэш предыдущего блока (`PrevBlock`), адрес отправителя (`Sender`), адрес получателя (`Receiver`) и размер цепочки блоков (`Value`).
- Функция устанавливает время создания блока (`TimeStamp`) на текущее время в формате `RFC3339`.
- Функция вычисляет хэш блока (`CurrHash`) с помощью метода `hash()`.
- Функция подписывает блок (`Signature`) с помощью приватного ключа пользователя (`user.Private()`) с помощью метода `sign()`.
- Функция вычисляет значение `Nonce` (`Nonce`) с помощью метода `proof()`, который использует алгоритм `Proof of Work` для вычисления `Nonce`.
- Функция возвращает `nil`, если все действия были выполнены успешно.

```
func (block *Block) transactionsIsValid(chain *BlockChain, size uint64) bool { 2 usages
    lentxs := len(block.Transactions)
    chainSize := chain.Size()
    em := emission(chainSize)
    str := strconv.FormatUint(em, base 10)
    plusStorage := 0
    for i := 0; i < lentxs; i++ {
        if block.Transactions[i].Sender == str {
            plusStorage = 1
            break
        }
    }
    if lentxs == 0 || lentxs > TXS_LIMIT+plusStorage : false }
    for i := 0; i < lentxs-1; i++ {
        for j := i + 1; j < lentxs; j++ {
            if bytes.Equal(block.Transactions[i].RandBytes, block.Transactions[j].RandBytes) : false }
            if block.Transactions[i].Sender == str &&
               block.Transactions[j].Sender == str : false }
        }
    }
    for i := 0; i < lentxs; i++ {
        tx := block.Transactions[i]
        if tx.Sender == str {
            if tx.Receiver != block.Miner || tx.Value != emission(chainSize) : false }
        } else {
            if !tx.hashIsValid() : false }
            if !tx.signIsValid() : false }
        }
        if !block.balanceIsValid(chain, tx.Sender, size) : false }
        if !block.balanceIsValid(chain, tx.Receiver, size) : false }
    }
    return true
}
```

Рисунок 33 – Функция подтверждения всех транзакций в блоке

Функция `transactionsIsValid` проверяет, являются ли все транзакции в блоке действительными и соответствующими правилам блокчейна. Входными параметрами являются указатель на блок и указатель на блокчейн, а также размер блока. Функция возвращает `true`, если все транзакции в блоке действительны, и `false` в противном случае.

Функция начинается с определения количества транзакций в блоке и размера блокчейна, а также с расчета эмиссии на текущий размер блокчейна. Затем проверяется, есть ли в блоке транзакция от эмитента (т.е. транзакция, которая отправляет монеты создателю блока). Если такая транзакция есть, то увеличивается счетчик `plusStorage` на 1.

Далее функция проверяет, что количество транзакций в блоке не превышает лимит, определенный константой `TXS_LIMIT`, увеличенный на `plusStorage`. Если количество транзакций превышает этот лимит, функция возвращает `false`.

Затем функция проверяет каждую транзакцию в блоке. Сначала проверяется, есть ли транзакция, отправленная от эмитента, и если есть, то проверяется, что она отправляет монеты только создателю блока и что ее значение соответствует текущей эмиссии. Затем проверяется, что транзакция имеет правильный хеш и подпись, если она не отправлена от эмитента. Затем проверяется, что баланс отправителя и получателя транзакции достаточен для проведения этой транзакции.

Если все проверки пройдены успешно, то функция возвращает `true`, иначе возвращает `false`.

```

func (block *Block) hash() []byte { 3 usages
    var tempHash []byte
    for _, tx := range block.Transactions {
        tempHash = CnFastHash(bytes.Join(
            [][]byte{
                tempHash,
                tx.CurrHash,
            },
            []byte{},
        ))
    }
    var list []string
    for hash := range block.Mapping {
        list = append(list, hash)
    }
    sort.Strings(list)
    for _, hash := range list {
        tempHash = CnFastHash(bytes.Join(
            [][]byte{
                tempHash,
                []byte(hash),
                ToBytes(block.Mapping[hash]),
            },
            []byte{},
        ))
    }
    return CnFastHash(bytes.Join(
        [][]byte{
            tempHash,
            ToBytes(uint64(block.Difficulty)),
            block.PrevHash,
            []byte(block.Miner),
            []byte(block.TimeStamp),
        },
        []byte{},
    ))
}

```

Рисунок 34 – Функция подтверждения всех транзакций в блоке

Функция (block \*Block) hash вычисляет хеш блока.

Переменная tempHash инициализируется как пустой слайс байтов. Затем происходит итерация по транзакциям блока. Для каждой транзакции создается хеш, который добавляется в tempHash вместе с предыдущим значением.

Затем создается слайс строк list, в который добавляются все хеши, на которые ссылается блок. Слайс сортируется, а затем для каждого хеша из

него создается хеш, который добавляется в tempHash вместе с предыдущим значением.

Наконец, создается итоговый хеш блока, который состоит из tempHash, текущей сложности блока в виде байтов, хеша предыдущего блока, имени майнера, который создал блок, и временной метки создания блока. Все эти значения объединяются вместе при помощи bytes.Join и хешируются при помощи функции CnFastHash.

Таким образом, данная функция вычисляет хеш блока, используя информацию о транзакциях, ссылках на другие блоки и другие параметры.

```
func (block *Block) balanceIsValid(chain *BlockChain, address string, size uint64) bool { 2 usages
    if _, ok := block.Mapping[address]; !ok { return false }
    lentxs := len(block.Transactions)
    balanceInChain := chain.Balance(address, size)
    balanceSubBlock := uint64(0)
    balanceAddBlock := uint64(0)
    for j := 0; j < lentxs; j++ {
        tx := block.Transactions[j]
        if tx.Sender == address {
            balanceSubBlock += tx.Value + tx.Emission
        }
        if tx.Receiver == address {
            balanceAddBlock += tx.Value
        }
    }
    if (balanceInChain + balanceAddBlock - balanceSubBlock) != block.Mapping[address] { return false }
    return true
}
```

Рисунок 35 – Функция подтверждения баланса адреса

Функция balanceIsValid проверяет, является ли баланс адреса в данном блоке верным. Входными параметрами функции являются указатель на блок (block), указатель на цепочку блоков (chain), адрес (address) и размер (size).

Сначала функция проверяет, есть ли данный адрес в маппинге блока. Если его нет, то функция возвращает false, так как баланс невозможно проверить.

Далее функция вычисляет баланс адреса в цепочке блоков) с помощью метода Balance. Затем функция начинает итерироваться по транзакциям блока и вычисляет суммы, которые нужно добавить или вычесть из баланса

блока. Если адрес является отправителем транзакции, то сумма, которую нужно вычесть из баланса блока, равна сумме отправленных монет и эмиссии. Если адрес является получателем транзакции, то сумма, которую нужно добавить к балансу блока, равна сумме полученных монет.

После того, как функция прошла по всем транзакциям блока и вычислила суммы для изменения баланса, она сравнивает общий баланс с балансом адреса, указанным в маппинге блока. Если они не совпадают, то функция возвращает false. В противном случае, функция возвращает true, что означает, что баланс адреса в блоке верный.

```
func ProofOfWork(blockHash []byte, difficulty uint8, ch chan bool) uint64 { 1 usage
    var (
        Target = big.NewInt(x: 1)
        intHash = big.NewInt(x: 1)
        nonce   = uint64(mrand.Intn(math.MaxUint32))
        hash    []byte
    )
    Target.Lsh(Target, 256-uint(difficulty))
    for nonce < math.MaxUint64 {
        select {
        case <-ch:
            if DEBUG {
                fmt.Println()
            }
            return nonce
        default:
            hash = CnFastHash(bytes.Join(
                [][]byte{
                    blockHash,
                    ToBytes(nonce),
                },
                [][]byte{},
            ))
            if DEBUG {
                fmt.Printf("\rMining: #{Base64Encode(hash)}")
            }
            intHash.SetBytes(hash)
            if intHash.Cmp(Target) == -1 {
                if DEBUG {
                    fmt.Println()
                }
                return nonce
            }
            nonce++
        }
    }
    return nonce
}
```

Рисунок 36 – Функция реализация алгоритма консенсуса Proof-of-Work



Функция `ProofOfWork` реализует алгоритм "Proof of Work" для генерации новых блоков в блокчейне. Она принимает на вход хеш предыдущего блока, сложность добычи (`difficulty`) и канал (`ch`), через который может быть отправлен сигнал для прерывания генерации нового блока.

В начале функция создает переменные `Target`, `intHash`, `nonce` и `hash`. `Target` – это целое число типа `big.Int`, равное 1, которое будет использоваться для сравнения с хешем, генерируемым в процессе добычи нового блока. `intHash` – это также целое число типа `big.Int`, которое будет использоваться для преобразования сгенерированного хеша в целочисленный вид для сравнения с `Target`. `nonce` – это случайное число типа `uint64`, которое будет использоваться для генерации разных хешей при каждой попытке добычи блока. `hash` – это байтовый массив, в который будут записываться хеши, генерируемые при каждой попытке добычи блока.

Затем функция вычисляет значение `Target`, используя операцию сдвига влево (`Lsh`) на  $256 - \text{uint}(\text{difficulty})$  битов. Это значение будет использоваться для сравнения с `intHash`, чтобы проверить, удовлетворяет ли сгенерированный хеш сложности добычи.

Далее функция запускает цикл `while`, который будет продолжаться, пока `nonce` меньше максимального значения `uint64`. Внутри цикла функция проверяет, был ли отправлен сигнал через канал `ch`, чтобы прервать генерацию нового блока. Если сигнал был отправлен, функция возвращает значение `nonce`.

Если сигнал не был отправлен, функция генерирует новый хеш, объединяя хеш предыдущего блока и `nonce` с помощью функции `CnFastHash`. Затем она проверяет, удовлетворяет ли сгенерированный хеш сложности добычи, сравнивая `intHash` и `Target`. Если сгенерированный хеш удовлетворяет сложности добычи, функция возвращает значение `nonce`.

Если сгенерированный хеш не удовлетворяет сложности добычи, функция увеличивает `nonce` на единицу и повторяет процесс генерации хеша.

Если nonce достиг максимального значения uint64, функция возвращает значение nonce.

```
func Verify(pub *ed25519.PublicKey, data, sign []byte) error { 2 usages
    if len(*pub) != ed25519.PublicKeySize : errors.New("invalid public key size")
    if !ed25519.Verify(*pub, data, sign) : errors.New("signature verification failed")
    return nil
}
```

Рисунок 37 – Функция проверки подписи

Функция Verify проверяет подпись данных, используя открытый ключ и алгоритм Ed25519.

Принимает на вход указатель на открытый ключ (pub), сами данные (data) и подпись (sign) в виде срезов байт.

Сначала функция проверяет, что длина открытого ключа соответствует размеру, определенному для алгоритма Ed25519. Если длина не соответствует, то возвращается ошибка "invalid public key size".

Затем функция вызывает функцию Verify, передавая ей открытый ключ, данные и подпись. Если верификация подписи не проходит, то возвращается ошибка "signature verification failed".

Если верификация проходит успешно, то функция возвращает nil, т.е. ошибок не возникает.

```
func ParsePublic(pubData string) *ed25519.PublicKey { 3 usages
    keyBytes, err := base64.StdEncoding.DecodeString(pubData)
    if err != nil {
        fmt.Println("Error decoding public key:", err)
        return nil
    }
    if len(keyBytes) != ed25519.PublicKeySize*2 {
        fmt.Println("Public key has unexpected length:", len(keyBytes))
        return nil
    }
    pubKey := ed25519.PublicKey(keyBytes[:ed25519.PublicKeySize])
    return &pubKey
}
```

Рисунок 38 – Функция преобразование строки в публичный ключ

Функция ParsePublic принимает строку pubData, которая представляет собой открытый ключ в формате base64. Сначала функция декодирует строку из base64 в байты и проверяет на наличие ошибок. Затем она проверяет, что длина байтов соответствует размеру открытого ключа ed25519. Если длина не соответствует, функция выдаст ошибку и вернет nil. Если длина верна, она создает новый открытый ключ ed25519 из первых PublicKeySize байтов и возвращает указатель на этот ключ. Если при декодировании произошла ошибка, функция выдаст ошибку и вернет nil.

```
func Base64Decode(data string) []byte { 2 usages
    result, err := base64.StdEncoding.DecodeString(data)
    if err != nil : nil
    return result
}
```

Рисунок 39 – Функция екодирования данных

Функция Base64Decode принимает в качестве аргумента строку data, которая содержит закодированные в формате Base64 данные. Функция декодирует эти данные и возвращает результат в виде среза байтов.

```
func GeneratePrivate() (*ed25519.PublicKey, *ed25519.PrivateKey) {
    pubKey, privKey, err := ed25519.GenerateKey(rand.Reader)
    if err != nil : nil, nil
    return &pubKey, &privKey
}
```

Рисунок 40 – Функция генерации ключей

Функция GeneratePrivate генерирует публичный и приватный ключ на основе эллиптической кривой Curve 25519.

```
func StringPrivate(priv *ed25519.PrivateKey) string {
    keyBytes := make([]byte, ed25519.PrivateKeySize)
    copy(keyBytes[:], *priv)
    return base64.StdEncoding.EncodeToString(keyBytes)
}
```

Рисунок 41 – Функция преобразования приватного ключа в строку

Функция StringPrivate преобразует приватный ключ пользователя в строковое представление.

```
func ParsePrivate(privData string) *ed25519.PrivateKey { 1 usage
    keyBytes, err := base64.StdEncoding.DecodeString(privData)
    if err != nil {
        fmt.Println(a...: "Error decoding private key:", err)
        return nil
    }
    if len(keyBytes) != ed25519.PrivateKeySize {
        fmt.Println(a...: "Private key has unexpected length:", len(keyBytes))
        return nil
    }
    privKey := ed25519.PrivateKey(keyBytes)
    return &privKey
}
```

Рисунок 42 – Функция преобразования строки в приватный ключ

Функция ParsePrivate преобразует строку в приватный ключ в случае ошибки работы base64.StdEncoding.DecodeString или неправильного размера ключа функция возвращает nil.

```
func NewUser() *User { 1 usage
    pub, priv := GeneratePrivate()
    return &User{
        PublicKey: pub,
        PrivateKey: priv,
    }
}
```

Рисунок 43 – Функция создания нового пользователя

Функция `NewUser` используется для создания нового пользователя. Она вызывает функцию `GeneratePrivate()` для создания публичного и приватного ключа пользователя затем ключи записывают в структуру `User` и возвращается ее экземпляр.

```
func LoadUser(purse, pubKey string) *User { 1 usage

    purse = strings.ReplaceAll(purse, old: "\r", new: "")
    purse = strings.ReplaceAll(purse, old: " ", new: "")
    purse = strings.ReplaceAll(purse, old: "\n", new: "")

    pubKey = strings.ReplaceAll(pubKey, old: "\r", new: "")
    pubKey = strings.ReplaceAll(pubKey, old: " ", new: "")
    pubKey = strings.ReplaceAll(pubKey, old: "\n", new: "")

    priv := ParsePrivate(purse)
    pub := ParsePublic(pubKey)
    if priv == nil : nil
    return &User{
        PublicKey: pub,
        PrivateKey: priv,
    }
}
```

Рисунок 44 – Функция загрузки пользователя

Функция `LoadUser` получает на вход публичный и приватный ключ пользователя в строковом представлении, затем осуществляет проверку ключей на предмет эскейп последовательностей. Если таковые присутствуют они удаляются и ключи переводятся из строкового представления в формат `*ed25519`

В завершении создается экземпляр структуры `User` на основе ключей.

```

func (block *Block) IsValid(chain *BlockChain, size uint64) bool {
    switch {
    case block == nil:
        return false
    case block.Difficulty != DIFFICULTY:
        return false
    case !block.hashIsValid(chain, size):
        return false
    case !block.signIsValid():
        return false
    case !block.proofIsValid():
        return false
    case !block.mappingIsValid():
        return false
    case !block.timeIsValid(chain):
        return false
    case !block.transactionsIsValid(chain, size):
        return false
    }
    return true
}

```

Рисунок 45 – Функция проверки валидности блока

Функция IsValid проверяет валидность блока в блокчейне.

Она принимает два аргумента: указатель на блок и указатель на блокчейн. Первым делом функция проверяет, что переданный блок не является nil. Если это так, функция возвращает false и заканчивает свою работу. Далее, функция проверяет, что сложность блока соответствует установленной константе DIFFICULTY. Если нет, функция также возвращает false и заканчивает свою работу.

Затем функция проверяет, что хеш блока является действительным, используя метод hashIsValid, который принимает два аргумента: указатель на блокчейн и размер текущей цепочки. Если хеш не является действительным, функция возвращает false и заканчивает свою работу.

Далее, функция проверяет, что подпись блока является действительной, используя метод signIsValid. Если подпись недействительна, функция возвращает false и заканчивает свою работу.

Затем функция проверяет, что доказательство выполнено правильно, используя метод `proofIsValid`. Если доказательство недействительно, функция возвращает `false` и заканчивает свою работу.

Далее, функция проверяет, что маппинг блока является действительным, используя метод `mappingIsValid`. Если маппинг недействителен, функция возвращает `false` и заканчивает свою работу.

Затем функция проверяет, что временная метка блока является действительной, используя метод `timeIsValid`, который принимает указатель на блокчейн. Если временная метка недействительна, функция возвращает `false` и заканчивает свою работу.

Наконец, функция проверяет, что все транзакции в блоке являются действительными, используя метод `transactionsIsValid`, который принимает указатель на блокчейн и размер текущей цепочки. Если какая-либо транзакция недействительна, функция возвращает `false` и заканчивает свою работу.

Если все проверки пройдены успешно, функция возвращает `true`.

```
func SerializeTX(tx *Transaction) string { 1 usage
    jsonData, err := json.MarshalIndent(tx, prefix: "", indent: "\t")
    if err != nil {
        return string(jsonData)
    }
}
```

Рисунок 46 – Функция сериализации транзакции

Функция `SerializeTX` принимает указатель на структуру `Transaction` и возвращает строку, которая является сериализованным JSON-объектом этой структуры.

```
func DeserializeTX(data string) *Transaction {
    var tx Transaction
    err := json.Unmarshal([]byte(data), &tx)
    if err != nil {
        return &tx
    }
}
```

Рисунок 47 – Функция десериализации транзакции

Функция `DeserializeTX` принимает строку, которая является JSON–объектом структуры `Transaction` и десериализует объект в структуру.

```
func (block *Block) hashIsValid(chain *BlockChain, size uint64) bool { 1 usage
    if !bytes.Equal(block.hash(), block.CurrHash) : false
    var id uint64
    row := chain.DB.QueryRow( query: "SELECT Id FROM BlockChain WHERE Hash=$1",
        Base64Encode(block.PrevHash))
    row.Scan(&id)
    return id == size
}
```

Рисунок 48 – Функция проверки действительности хэша

Функция `hashIsValid` проверяет, является ли хэш текущего блока (`block`) действительным. Для этого она сравнивает хэш текущего блока (`block.hash()`) с хэшем (`block.CurrHash`), который был вычислен и сохранен на момент создания блока. Если хэши не совпадают, функция возвращает `false`, что означает, что блок недействительный.

Если хэши совпадают, функция ищет в базе данных запись с `id` блока, который предшествует текущему блоку (`block.PrevHash`). Для этого она выполняет SQL–запрос к таблице `BlockChain`, в которой хранятся все блоки, и ищет запись с хэшем предыдущего блока (`block.PrevHash`). Если такая запись найдена, функция извлекает значение `id` из этой записи и сохраняет его в переменную `id`.

Затем функция сравнивает значение `id` с размером цепочки блоков (`size`), переданным в качестве аргумента функции. Если `id` равно `size`, то это означает, что блок находится в конце цепочки и является действительным. В этом случае функция возвращает `true`. Если `id` не равно `size`, то это означает, что блок находится в середине цепочки и является недействительным. В этом случае функция возвращает `false`.



```

func (block *Block) proofIsValid() bool { 1 usage
    intHash := big.NewInt( x: 1)
    Target := big.NewInt( x: 1)
    hash := CnFastHash(bytes.Join(
        [][]byte{
            block.CurrHash,
            ToBytes(block.Nonce),
        },
        []byte{},
    ))
    intHash.SetBytes(hash)
    Target.Lsh(Target, 256-uint(block.Difficulty))
    if intHash.Cmp(Target) == -1 : true
    return false
}

```

Рисунок 49 – Функция проверки алгоритма консенсуса

Функция проверяет proofIsValid, является ли текущий блок валидным с точки зрения его доказательства работы (proof of work). Для этого она выполняет следующие действия:

1. Создает два объекта типа big.Int: intHash и Target.
2. Вычисляет хеш текущего блока с помощью функции CnFastHash, передавая в нее массив байт, состоящий из хеша предыдущего блока (CurrHash) и значения Nonce текущего блока, преобразованного в массив байт (ToBytes).
3. Преобразует полученный хеш в объект big.Int и сохраняет его в переменную intHash.
4. Вычисляет значение Target, сдвигая число 1 на (256 – сложность текущего блока).
5. Сравнивает значения intHash и Target. Если intHash меньше Target, то функция возвращает true, иначе – false.

Таким образом, функция проверяет, выполнено ли условие доказательства работы для текущего блока, которое состоит в том, чтобы найти хеш блока, который начинается с определенного количества нулей в

зависимости от установленной сложности. Если хеш меньше цели, то доказательство работы считается выполненным успешно, и блок считается валидным.

```
func (block *Block) mappingIsValid() bool { 1 usage
    for hash := range block.Mapping {
        flag := false
        for _, tx := range block.Transactions {
            if tx.Sender == hash || tx.Receiver == hash {
                flag = true
                break
            }
        }
        if !flag : false
    }
    return true
}
```

Рисунок 50 – Функция проверки адресов транзакции

Функция `mappingIsValid` проверяет, является ли отображение адресов транзакций в блоке корректным.

Сначала функция проходится по всем адресам в маппинге блока с помощью цикла. Для каждого адреса проверяется, есть ли хотя бы одна транзакция в блоке, в которой этот адрес является отправителем или получателем.

Если такой транзакции нет, то для данного адреса маппинг считается невалидным и функция возвращает `false`.

Если для всех адресов в маппинге найдена хотя бы одна транзакция, то функция возвращает `true`.

```

func (block *Block) timeIsValid(chain *BlockChain) bool { 1 usage
    btime, err := time.Parse(time.RFC3339, block.TimeStamp)
    if err != nil : false
    diff := time.Now().Sub(btime)
    if diff < 0 : false
    var sblock string
    row := chain.DB.QueryRow( query: "SELECT Block FROM BlockChain WHERE Hash=$1",
        Base64Encode(block.PrevHash))
    row.Scan(&sblock)
    lblock := DeserializeBlock(sblock)
    if lblock == nil : false
    ltime, err := time.Parse(time.RFC3339, lblock.TimeStamp)
    if err != nil : false
    result := btime.Sub(ltime)
    return result > 0
}

```

Рисунок 51 – Функция проверки времени создания блока

Функция `timeIsValid` проверяет, является ли время создания блока корректным.

Сначала функция парсит время создания блока в формате RFC3339 и возвращает `false`, если происходит ошибка при парсинге. Затем функция сравнивает разницу текущего времени и времени создания блока. Если разница меньше нуля, то это означает, что время создания блока в будущем, что является ошибкой.

Затем функция проверяет предыдущий блок, используя базу данных из переданного блокчейна. Если предыдущий блок не найден, функция возвращает `false`.

Затем функция парсит время создания предыдущего блока и сравнивает его с временем создания текущего блока. Если разница между временем создания предыдущего блока и текущего блока меньше нуля, то это означает, что текущий блок был создан раньше, чем предыдущий блок, что является ошибкой.

Если все проверки прошли успешно, то функция возвращает `true`, что означает, что время создания блока является корректным.

### 3.2 Реализация приложения пользователя криптовалюты

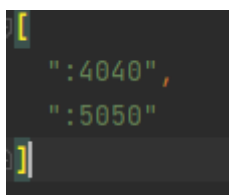
Клиентское приложение состоит из двух частей:

1. Инициализация, при которой необходимо указать файл, в котором находится приватный и публичный ключ пользователя (либо создать такие ключи), а также на адрес узлов.
2. Консольный интерфейс, который будет реализован через стандартный поток ввода. В нём можно будет прописывать команды проверки баланса, отправления транзакций и вывода цепочки блоков.

Примером входа в программу должна служить следующая строка:

```
/user -loaduser:private.key -loadaddr:addrlist.json
```

Указывается подгружаемые приватный и публичный ключи, который создадут объект структуры User, а также подгружается список адресов. Стоит заметить, что список адресов должен иметь формат json. Помимо параметра подгрузки пользователя (-loaduser:) можно использовать параметр создания нового пользователя (-newuser:). Если приватный и публичный ключи уже создан и хранятся в файле, а в параметре newuser указан этот же файл, тогда произойдёт перезапись существующего файла, создав тем самым новые ключи



```
[  
  "4040",  
  "5050"  
]
```

Рисунок 52 – Пример файла addrlist.json

Клиентское приложение обладает следующими возможностями:

- Просмотр публичного ключа пользователя
- Просмотр приватного ключа пользователя
- Просмотр баланса пользователя
- Печать блокчейна

- Печать размера блокчейна
- Совершение транзакции

Ниже будут приведены демонстрация работы функция пользовательского приложения и реализация методов функция программы

```
> /user address
Address: Pql0ju/P0anjFYmL2WSgIivWkiy1b+XxQ0LxmcZ5RawAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA==
```

Рисунок 53 – Печать публичного ключа пользователя

Из-за особенностей работы библиотеки для работы с эллиптическим кривыми публичный ключ создается длиной 64 байта, хотя в действительности самым ключем является только первые 32 байта.

```
func userAddress() { 1 usage
    fmt.Println(a...: "Address:", User.Pub(), "\n")
}
```

Рисунок 54 – Функция печати публичного ключа пользователя

Функция userAddress обращается к структуре User и возвращает публичный ключ пользователя, затем печатает его в консоль.

```
> /user purse
Purse: yMfcIQ5frRBu9hePtWR3LsUogUj8F12L2W0A5GZZYdA+qXS078/RqeMVgyXZZKCWK9aSLLVv7FerQv6Zxn1FrA==
```

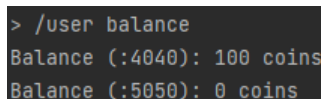
Рисунок 55 – Печать приватного ключа пользователя

В данном примере мы можем наблюдать печать приватного ключа пользователя, который равен 64 байтом.

```
func userPurse() { 1 usage
    fmt.Println(a...: "Purse:", User.Purse(), "\n")
}
```

Рисунок 56 – Функция печати приватного ключа пользователя

Функция `userPurse` обращается к структуре `User` и возвращает приватный ключ пользователя, затем печатает его в консоль.



```
> /user balance  
Balance (:4040): 100 coins  
Balance (:5050): 0 coins
```

Рисунок 57 – Печать баланса пользователя

Здесь мы можем наблюдать разное значение баланса на узлах криптовалюты это произошло, потому что блоки были созданы в разное время.

Чтобы выравнять балансы на разных узлах, существует процесс синхронизации блокчейна. Этот процесс подразумевает обновление информации на всех узлах сети, чтобы они имели одинаковую информацию о транзакциях и балансах. Это происходит благодаря работе майнеров, которые добавляют новые блоки в блокчейн. Каждый блок содержит информацию о транзакциях, которые произошли на момент создания блока, и майнеры получают вознаграждение за свою работу. Когда новый блок добавляется в блокчейн, информация о транзакциях и балансах обновляется на всех узлах сети, и балансы на разных узлах становятся одинаковыми.

В реализации данной криптовалюты майнинг произойдет при добавление 3 транзакций и баланс на всех узлах станет одинаковым

```

func printBalance(useraddr string) { 1 usage
    for _, addr := range Addresses {
        res := nt.Send(addr, &nt.Package{
            Option: GET_BLNCE,
            Data:   useraddr,
        })
        if res == nil {
            continue
        }
        fmt.Printf("Balance ({addr}): #{res.Data} coins\n")
    }
    fmt.Println()
}

```

Рисунок 58 – Функция печати баланса пользователя

Функция `printBalance` печатает баланс пользователя с заданным адресом в блокчейне. Функция использует для этого цикл, который проходит по каждому адресу в слайсе `Addresses`. Для каждого адреса происходит отправка запроса на получение баланса пользователя с заданным адресом.

Запрос создается с помощью метода `Send`, который принимает адрес и объект типа `Package`, содержащий опцию `GET_BLNCE` (запрос баланса) и данные – адрес пользователя, баланс которого нужно получить.

Если ответ на запрос не получен, цикл переходит к следующему адресу. Если ответ получен, то функция печатает строку с балансом пользователя и адресом, с которого был получен ответ.

```

> /chain print
[1] => {
    "Nonce": 0,
    "Difficulty": 0,
    "CurrHash": "yt2Esw==",
    "PrevHash": "R0V0RVNJUY1CTE9DSw==",
    "Transactions": null,
    "Mapping": {
        "PqL0ju/P0anjFYML2WSglivWkiy1b+xXq0LxmcZ5RawAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA==": 100
    },
    "Miner": "PqL0ju/P0anjFYML2WSglivWkiy1b+xXq0LxmcZ5RawAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA==",
    "Signature": null,
    "TimeStamp": "2023-06-01T22:59:03+03:00"
}

```

Рисунок 59 – Печать блокчена

Здесь мы можем наблюдать блокчейн не имеющий транзакция и состоящего из одного блока. Этот блок называется “генезис блок” он является первым блок блокчейна и нужен для построения последующих.

```
func chainPrint() { 1 usage
    for i := 0; ; i++ {
        res := nt.Send(Addresses[0], &nt.Package{
            Option: GET_BLOCK,
            Data:    fmt.Sprintf(format: "%d", i),
        })
        if res == nil || res.Data == "" {
            break
        }
        fmt.Printf("#[i+1] => #{res.Data}\n")
    }
    fmt.Println()
}
```

Рисунок 60 – Функция печати блокчена

Функция chainPrint осуществляет циклический запрос информации о блоках в блокчейне с помощью сетевого пакета nt.Package и отправки его на первый адрес (Addresses[0]).

В данной функции используется бесконечный цикл for, в котором переменная i увеличивается на 1 каждую итерацию. Внутри цикла формируется пакет nt.Package с опцией GET\_BLOCK и данными в виде строки, содержащей номер блока. После этого пакет отправляется на первый адрес.

Затем проверяется ответ, полученный от первого адреса. Если ответ не был получен (res == nil) или данные в ответе пустые (res.Data == ""), то цикл прерывается с помощью break.

Если же ответ был получен, то в консоль выводится информация о блоке.

```
> /chain size
Size: 1 blocks
```

Рисунок 61 – Печать размера блокчена



На данном примере мы видим, что размер текущего блокчейна составляет 1 блок, а значит в цепи существует только генезис блок.

```
func chainSize() { 1 usage
    res := nt.Send(Addresses[0], &nt.Package{
        Option: GET_CSIZE,
    })
    if res == nil || res.Data == "" {
        fmt.Println(a...: "failed: getSize\n")
        return
    }
    fmt.Printf("Size: #{res.Data} blocks\n\n")
}
```

Рисунок 62 – Функция печати размера блокчейна

Данная функция выполняет запрос к сети, используя объект и отправляет пакет с опцией GET\_CSIZE (получить размер цепочки блоков). Результат запроса сохраняется в переменной res. Если результат запроса равен nil или строка данных пуста, то выводится сообщение об ошибке и функция завершается. В противном случае, функция выводит на экран размер цепочки блоков.

```
> /chain tx aaa 3
ок: (:4040)
ок: (:5050)
```

Рисунок 63 – совершение транзакции

Здесь осуществляется транзакция в размере 3 монет на адрес “aaa”. Два узла приняли ее.

```

func chainTX(splited []string) { 1 usage
    if len(splited) != 3 {
        fmt.Println("failed: len(splited) != 3\n")
        return
    }
    splited[2] = strings.TrimRight(splited[2], "\r")
    num, err := strconv.Atoi(splited[2])
    if err != nil {
        fmt.Println("failed: strconv.Atoi(num)\n")
        return
    }
    for _, addr := range Addresses {
        res := nt.Send(addr, &nt.Package{
            Option: GET_LHASH,
        })
        if res == nil {
            continue
        }
        tx := bc.NewTransaction(User, bc.Base64Decode(res.Data), splited[1], uint64(num), chainSizeTransact())
        res = nt.Send(addr, &nt.Package{
            Option: ADD_TRNSX,
            Data:   bc.SerializeTX(tx),
        })
        if res == nil {
            continue
        }
        if res.Data == "ok" {
            fmt.Printf("ok: (%s)\n", addr)
        } else {
            fmt.Printf("fail: (%s)\n", addr)
        }
    }
    fmt.Println()
}

```

Рисунок 64 – Функция для совершения транзакций

Функция chainTX принимает на вход массив строк splited и производит следующие операции:

Проверяет, что длина массива равна 3. Если это не так, выводит сообщение об ошибке и завершает функцию.

Обрезает символы переноса строки справа от третьего элемента массива.

Преобразует третий элемент массива в число типа int.

Для каждого элемента массива Addresses выполняет следующие действия:

Отправляет запрос на получение последнего блока цепочки на адрес addr с помощью функции nt.Send().

Если ответ не получен, переходит к следующему адресу.

Создает новую транзакцию с помощью функции `bc.NewTransaction()`. В качестве отправителя указывается `User`, а получателем – второй элемент массива `splited`.

Отправляет транзакцию на адрес `addr` с помощью функции `nt.Send()` и опцией `ADD_TRNSX`.

Если ответ не получен, переходит к следующему адресу.

Если ответ содержит строку "ok", выводит сообщение об успешном выполнении операции на адресе `addr`. В противном случае выводит сообщение об ошибке.

### **3.3 Реализация приложения узла криптовалюты**

Криптовалютный узел – это программа, которая служит для майнинга, синхронизации, подтверждения транзакций, проверки блоков и записи их в блокчейн.

В отличие от пользовательского приложения не имеет графического интерфейса, она служит для автоматического контроля и управления блокчейном.

```

func addBlock(pack *nt.Package) string { 1 usage
    splited := strings.Split(pack.Data, SEPARATOR)
    if len(splited) != 3 : "fail"
    block := bc.DeserializeBlock(splited[2])
    if !block.IsValid(chain, chain.Size()) {
        currSize := chain.Size()
        num, err := strconv.Atoi(splited[1])
        if err != nil : "fail"
        if currSize < uint64(num) {
            go compareChains(splited[0], uint64(num))
            return "ok "
        }
        return "fail"
    }
    Mutex.Lock()
    chain.AddBlock(block)
    Block = bc.NewBlock(User.Address(), chain.LastHash())
    Mutex.Unlock()
    if IsMining {
        BreakMining <- true
        IsMining = false
    }
    return "ok"
}

```

Рисунок 65 – Функция добавления блока в блокчейн

Функция `addBlock` добавляет новый блок в блокчейн. Сначала она разбивает данные пакета на три части с помощью разделителя. Затем она проверяет, что количество частей равно трем, иначе она возвращает "fail". Далее она десериализует блок из третьей части и проверяет его на валидность. Если блок не является валидным, то она сравнивает размер текущего блокчейна с номером блока, который пришел с пакетом. Если текущий размер меньше, чем номер блока, то она запускает функцию сравнения двух блокчейнов и возвращает "ok". Если же текущий размер больше или равен номеру блока, то она возвращает "fail". Если блок валидный, то она добавляет его в блокчейн, создает новый блок и разблокирует мьютекс. Если майнинг запущен, то она отправляет сигнал остановки майнинга и устанавливает флаг `IsMining` в `false`, а затем возвращает "ok".

```

func addTransaction(pack *nt.Package) string { 1 usage
    var tx = bc.DeserializeTX(pack.Data)
    if tx == nil || len(Block.Transactions) == bc.TXS_LIMIT {
        return "fail"
    }
    Mutex.Lock()
    err := Block.AddTransaction(chain, tx)
    Mutex.Unlock()
    if err != nil {
        return "fail"
    }
    if len(Block.Transactions) == bc.TXS_LIMIT {
        go func() {
            Mutex.Lock()
            block := *Block
            IsMining = true
            Mutex.Unlock()
            res := (&block).Accept(chain, User, BreakMining)
            Mutex.Lock()
            IsMining = false
            if res == nil && bytes.Equal(block.PrevHash, Block.PrevHash) {
                chain.AddBlock(&block)
                pushBlockToNet(&block)
            }
            Block = bc.NewBlock(User.Address(), chain.LastHash())
            Mutex.Unlock()
        }()
    }
    return "ok"
}

```

Рисунок 66 – Функция добавления транзакции в блокчейн

Функция `addTransaction` добавляет транзакцию в блокчейн. Она принимает пакет данных, десериализует транзакцию из пакета и добавляет ее в блок, если блок еще не заполнен. Если блок заполнен, функция запускает горутину, которая начинает добычу нового блока. Если добыча блока завершается успешно, новый блок добавляется в цепочку блоков и передается в сеть. В конце функция возвращает строку "ok", если транзакция была успешно добавлена в блок, или "fail", если произошла ошибка при добавлении.

```
func getBlock(pack *nt.Package) string {
    num, err := strconv.Atoi(pack.Data)
    if err != nil {
        return ""
    }
    size := Chain.Size()
    if uint64(num) < size {
        return selectBlock(Chain, num)
    }
    return ""
}
```

Рисунок 67 – Функция получения блока по его номеру

Функция `getBlock` получает пакет данных (`nt.Package`) и извлекает из него число (`num`) с помощью функции `Atoi` из `strconv`. Затем она проверяет, есть ли ошибка (`err`) при конвертации из строки в число. Если ошибка есть, то функция возвращает пустую строку. Если ошибки нет, то функция определяет размер цепочки блоков (`Chain`) и сравнивает число (`num`) с размером цепочки. Если число меньше размера цепочки, то функция вызывает функцию `selectBlock` с аргументами `Chain` и `num` и возвращает результат. Если число больше или равно размеру цепочки, то функция возвращает пустую строку. Функция `selectBlock` используется для выбора блока из цепочки блоков по его номеру.

```
func getLastHash(pack *nt.Package) string {
    return bc.Base64Encode(Chain.LastHash())
}
```

Рисунок 68 – Функция получения хеша

Функция `getLastHash` получает последний хеш блока в блокчейне.

```
func getBalance(pack *nt.Package) string { 1 usage
    return fmt.Sprintf(format: "%d", Chain.Balance(pack.Data, Chain.Size()))
}
```

Рисунок 69 – Функция получения хеша

Функция `getBalance` возвращает строковое представление баланса (количество монет) на заданном пакете (`pack`)

```
func getChainSize(pack *nt.Package) string { 1 usage
    return fmt.Sprintf(format: "%d", Chain.Size())
}
```

Рисунок 70 – Функция получения размера блокчейна

Функция `getChainSize` возвращает размер блокчейна.

### 3.4 Реализация сетевой библиотеки для работы с блокчейном

```
func Send(address string, pack *Package) *Package { 6 usages
    conn, err := net.Dial(network: "tcp", address)
    if err != nil {
        return nil
    }
    conn.Write([]byte(serializePackage(pack) + ENDBYTES))
    var res = new(Package)
    ch := make(chan bool)
    go func() {
        res = readPackage(conn)
        ch <- true
    }()
    select {
    case <-ch:
    case <-time.After(WAITTIME * time.Second):
    }
    return res
}
```

Рисунок 71 – Функция отправки пакета

Функция Send отправляет сетевой пакет по указанному адресу через TCP-соединение. Если соединение не установлено, возвращает nil. В остальных случаях отправляет пакет с помощью метода и ожидает ответное сообщение, которое считывается с помощью функции readPackage() и сохраняется в переменной res. Если ответ не приходит в течение заданного времени (WAITTIME), функция возвращает nil. В противном случае, возвращает полученный ответный пакет.

```
func readPackage(conn net.Conn) *Package { 2 usages
    var (
        data    string
        size    = uint64(0)
        buffer = make([]byte, BUFFSIZE)
    )
    for {
        length, err := conn.Read(buffer)
        if err != nil {
            return nil
        }
        size += uint64(length)
        if size > DMAXSIZE {
            return nil
        }
        data += string(buffer[:length])
        if strings.Contains(data, ENDBYTES) {
            data = strings.Split(data, ENDBYTES)[0]
            break
        }
    }
    return DeserializePackage(data)
}
```

Рисунок 72 – Функция чтения пакета

Функция readPackage считывает пакет данных из сетевого соединения, пока не встретится символьная последовательность ENDBYTES. После этого извлекает данные до этой последовательности, используя функцию strings.Split(). Затем эти данные десериализуются с помощью функции DeserializePackage() и возвращаются в качестве результата. Если происходит



ошибка чтения или размер данных превышает предельный размер DMAXSIZE, функция возвращает nil.

```
func Listen(address string, handle func(Conn, *Package)) Listener {  
    splitted := strings.Split(address, ":")  
    if len(splitted) != 2 {  
        return nil  
    }  
    listener, err := net.Listen("tcp", "0.0.0.0:"+splitted[1])  
    if err != nil {  
        return nil  
    }  
    go serve(listener, handle)  
    return Listener{listener}  
}
```

Рисунок 73 – Функция прослушивания сокета

Данная функция создает TCP-сервер, прослушивающий указанный адрес и порт. Когда на сервер поступает новое соединение, функция вызывает заданную обработчиком функцию handle, передавая ей объект типа Conn и указатель на пакет (\*Package). Затем функция запускает горутину serve, которая будет обслуживать соединение. Функция возвращает объект типа Listener, который можно использовать для остановки сервера. Если происходит ошибка при создании сервера, функция возвращает nil.

```
func serve(listener net.Listener, handle func(Conn, *Package)) {  
    defer listener.Close()  
    for {  
        conn, err := listener.Accept()  
        if err != nil {  
            break  
        }  
        go handleConn(conn, handle)  
    }  
}
```

Рисунок 74 – Функция обработки соединений

Функция `serve` служит для обработки входящих соединений на заданном сетевом порту. Она ожидает подключения клиентов к заданному `listener`'у и после подключения передает управление в функцию `handle`, которая обрабатывает соединение. Функция `serve` работает в бесконечном цикле, пока не произойдет ошибка при приеме соединения. После каждого обработанного соединения функция `serve` продолжает ожидать новых соединений. После завершения работы функция закрывает `listener`.

```
func handleConn(conn net.Conn, handle func(Conn, *Package)) {
    defer conn.Close()
    pack := readPackage(conn)
    if pack == nil {
        return
    }
    handle(Conn(conn), pack)
}
```

Рисунок 75 – Функция обработки соединений

Функция `handleConn` обрабатывает соединение `'net.Conn'`, считывает пакет данных с помощью функции `'readPackage'`, переданной в качестве аргумента, и вызывает функцию `'handle'`

```
func Handle(option int, conn Conn, pack *Package, handle func(*Package) string) bool {
    if pack.Option != option {
        return false
    }
    conn.Write([]byte(serializePackage(&Package{
        Option: option,
        Data:    handle(pack),
    }) + ENDBYTES))
    return true
}
```

Рисунок 76 – Функция обработки пакетов

Функция `Handle` обрабатывает определенный пакет данных, указанный в параметре `"option"`, который приходит по соединению `"conn"`. Если опция

пакета совпадает с указанной опцией в параметре "option", то функция вызывает функцию "handle" и передает ей данный пакет, чтобы получить строковый результат. Затем функция использует функцию "SerializePackage" для сериализации результата в байты и записывает их в соединение "conn". Функция возвращает true, если пакет был обработан успешно и false в противном случае.

## ЗАКЛЮЧЕНИЕ

В данной работе была реализован криптовалютный блокчейн с применением технологии CryptoNote и других таких как black2b, эллиптические кривые, CryptoNigtht.

Для взаимодействия пользователь в сети и работы CryptoNote был реализован алгоритм консенсуса Proof of Work.

Было реализовано приложения криптовалютного узла для автоматической обработки и контроля операций внутри блокчейна.

С целью контроля масштабируемости блокчен был реализован алгоритм криптовалютой эмиссии для контроля масштабируемости сети. Чтобы пользователи могли совершать транзакции было реализовано пользовательское приложение.

Тесты на криптовалютных узлах и пользовательском приложении показали, что алгоритмы работают корректно и блокчейн функционирует без ошибок.

Все цели работы были достигнуты. В дальнейшем можно модифицировать криптовалюту добавлением древа Меркала для хранения и быстрого доступа к блокам транзакций с целью проверки целостности. Так же можно реализовать графический интерфейс клиентского приложения и добавить технологию i2p для большой анонимности пользователей.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Шнайер Б. Прикладная криптография. Протоколы, алгоритмы и исходный код на С.
2. Кеннеди У., Кетелсен Б., Сент–Мартин Э. Go in Action. 2015.
3. Прасти Н. Блокчейн. Разработка приложений.
4. Керниган Б.У., Донован А.А. Язык программирования Go.
5. Шнайер Б., Фергюсон Н.Т. Практическая криптография.

## ПРИЛОЖЕНИЕ А

Сетевая библиотека, реализующая логику для работы с сетью.

```
package network

import (
    "encoding/json"
    "net"
    "strings"
    "time"
)

type Package struct {
    Option int
    Data   string
}

func Send(address string, pack *Package) *Package {
    conn, err := net.Dial("tcp", address)
    if err != nil {
        return nil
    }
    conn.Write([]byte(SerializePackage(pack) + ENDBYTES))
    var res = new(Package)
    ch := make(chan bool)
    go func() {
        res = readPackage(conn)
        ch <- true
    }()
    select {
    case <-ch:
    case <-time.After(WAITTIME * time.Second):
    }
    return res
}

func SerializePackage(pack *Package) string {
    jsonData, err := json.MarshalIndent(*pack, "", "\t")
    if err != nil {
        return ""
    }
    return string(jsonData)
}
```

```
const (
    ENDBYTES = "\000\005\007\001\001\007\005\000"
)
```

```
func readPackage(conn net.Conn) *Package {
    var (
        data string
        size  = uint64(0)
        buffer = make([]byte, BUFSIZE)
    )
    for {
        length, err := conn.Read(buffer)
        if err != nil {
            return nil
        }
        size += uint64(length)
        if size > DMAXSIZE {
            return nil
        }
        data += string(buffer[:length])
        if strings.Contains(data, ENDBYTES) {
            data = strings.Split(data, ENDBYTES)[0]
            break
        }
    }
    return DeserializePackage(data)
}
```

```
const (
    WAITTIME = 5 // seconds
    DMAXSIZE = (2 << 20) // (2^20)*2 = 2MiB
    BUFSIZE = (4 << 10) // (2^10)*4 = 4KiB
)
```

```
func DeserializePackage(data string) *Package {
    var pack Package
    err := json.Unmarshal([]byte(data), &pack)
    if err != nil {
        return nil
    }
    return &pack
}
```

```

func Listen(address string, handle func(Conn, *Package)) Listener {
    splited := strings.Split(address, ":")
    if len(splited) != 2 {
        return nil
    }
    listener, err := net.Listen("tcp", "0.0.0.0:"+splited[1])
    if err != nil {
        return nil
    }
    go serve(listener, handle)
    return Listener(listener)
}

func serve(listener net.Listener, handle func(Conn, *Package)) {
    defer listener.Close()
    for {
        conn, err := listener.Accept()
        if err != nil {
            break
        }
        go handleConn(conn, handle)
    }
}

func handleConn(conn net.Conn, handle func(Conn, *Package)) {
    defer conn.Close()
    pack := readPackage(conn)
    if pack == nil {
        return
    }
    handle(Conn(conn), pack)
}

type Listener net.Listener
type Conn net.Conn

func Handle(option int, conn Conn, pack *Package, handle func(*Package) string)
bool {
    if pack.Option != option {
        return false
    }
    conn.Write([]byte(SerializePackage(&Package{
        Option: option,
        Data: handle(pack),
    }) + ENDBYTES))
}

```



```
    return true  
}
```

## ПРИЛОЖЕНИЕ Б

Библиотека, реализующая логику для работы с блокчейном.

```
package blockchain
```

```
import (  
    "bytes"  
    "crypto/ed25519"  
    "crypto/rand"  
    "database/sql"  
    "encoding/base64"  
    "encoding/binary"  
    "encoding/json"  
    "errors"  
    "fmt"  
    _ "github.com/mattn/go-sqlite3"  
    "golang.org/x/crypto/blake2b"  
    "log"  
    "math"  
    "math/big"  
    mrand "math/rand"  
    "os"  
    "sort"  
    "strconv"  
    "time"  
)
```

```
type Blockchain struct {  
    DB *sql.DB  
}
```

```
type Block struct {  
    Nonce      uint64      // Результат подтверждения работы  
    Difficulty uint8       // Сложность блока  
    CurrHash   []byte     // Хеш текущего блока  
    PrevHash   []byte     // Хеш предыдущего блока  
    Transactions []Transaction // Транзакции пользователей  
    Mapping    map[string]uint64 //Состояние баланса пользователей  
    Miner      string     //Пользователь замайнивший блок  
    Signature   []byte     // Подпись майнера указывающая на CurrHash  
    TimeStamp   string     // Метка времени создания блока
```

```

}

type Transaction struct {
    RandBytes []byte //Случайные байты
    PrevBlock []byte // Хеш последнего блока в блокчейне
    Sender string //Имя отправителя
    Receiver string //Имя получателя
    Value uint64 //Количество переводимых денег получателю
    CurrHash []byte //хеш текущей транзакции
    Signature []byte //подпись отправителя
    Emission uint64 // показатель эмиссии
}

// Блок констант где содержаться SQL скрипты
const (
    CREATE_TABLES = `
    CREATE TABLE BlockChain (
        Id INTEGER PRIMARY KEY AUTOINCREMENT,
        Hash VARCHAR(44) UNIQUE,
        Block TEXT
    );
    `
)

// Блок где содержаться константы необходимы для работы блокчейна
const (
    GENESIS_BLOCK = "GENESIS-BLOCK" // Константа для создания хеша
    genesis_block
    DIFFICULTY = 20 //Сложность блока
    GENESIS_REWARD = 100 // Награда за создание генезис блока
    DEBUG = true
    RAND_BYTES = 32 // Констанция для генерации слачаных байтов
    TXS_LIMIT = 2 // Константа задающая лимиты транзакций
)

func NewChain(filename, receiver string) error {
    file, err := os.Create(filename)
    if err != nil {
        return err
    }
    file.Close()
    db, err := sql.Open("sqlite3", filename)
    if err != nil {
        return err
    }
}

```

```

    }
    defer db.Close()
    _, err = db.Exec(CREATE_TABLES)

    chain := &BlockChain{
        DB: db,
    }
    genesis := &Block{
        PrevHash: []byte(GENESIS_BLOCK),
        Mapping:   make(map[string]uint64),
        Miner:     receiver,
        TimeStamp: time.Now().Format(time.RFC3339),
    }

    genesis.Mapping[receiver] = GENESIS_REWARD
    genesis.CurrHash = genesis.hash()
    chain.AddBlock(genesis)
    return nil
}

func (chain *BlockChain) AddBlock(block *Block) {
    chain.DB.Exec("INSERT INTO BlockChain (Hash, Block) VALUES ($1, $2)",
        Base64Encode(block.CurrHash),
        SerializeBlock(block),
    )
}

func Base64Encode(data []byte) string {
    return base64.StdEncoding.EncodeToString(data)
}

func SerializeBlock(block *Block) string {
    jsonData, err := json.MarshalIndent(*block, "", "\t")
    if err != nil {
        return ""
    }
    return string(jsonData)
}

func LoadChain(filename string) *BlockChain {
    db, err := sql.Open("sqlite3", filename)
    if err != nil {
        return nil
    }

```

```

chain := &BlockChain{
    DB: db,
}
return chain
}

func NewBlock(miner string, prevHash []byte) *Block {
    return &Block{
        Difficulty: DIFFICULTY,
        PrevHash:   prevHash,
        Miner:     miner,
        Mapping:    make(map[string]uint64),
    }
}

func NewTransaction(user *User, lasthash []byte, to string, value uint64)
*Transaction {
    tx := &Transaction{
        RandBytes: GenerateRandomBytes(RAND_BYTES),
        PrevBlock: lasthash,
        Sender:    user.Address(),
        Receiver:  to,
        Value:     value,
    }
    tx.CurrHash = tx.hash()
    tx.Signature = tx.sign(user.Private())
    return tx
}

```

/\*

Данная функция реализует расчет эмиссии монет(стоймость за создание блока) первые 1024 блока награда 1000, затем каждые 1024 блока она будкт снижаться на 100. Когда в блокчейне станет 9216 блоков, награда станент постоянной и будет равна 100 монетам

\*/

```

func emission(BlockChainSize uint64) uint64 {

    if BlockChainSize <= 1024 {
        return 1000
    } else if BlockChainSize >= 9216 {
        return 100
    } else {
        return 1000 - (BlockChainSize/1024)*100
    }
}

```

```

    }
}

// Структура которая содержит приватный и публичный ключ пользователя.
type User struct {
    PublicKey *ed25519.PublicKey
    PrivateKey *ed25519.PrivateKey
}

// Данная функция создает массив случайных байтов
func GenerateRandomBytes(max uint) []byte {
    var slice []byte = make([]byte, max)
    _, err := rand.Read(slice)
    if err != nil {
        return nil
    }
    return slice
}

// Данная функция возвращает адрес пользователя(публичный ключ).
func (user *User) Address() string {
    return StringPublic(user.Public())
}

// Данная функция возвращает приватный ключ пользователя.
func (user *User) Private() *ed25519.PrivateKey {
    return user.PrivateKey
}

// Данная функция возвращает публичный ключ пользователя.
func (user *User) Public() *ed25519.PublicKey {
    return user.PublicKey
}

// /Данная функция используется для вычисления хеша транзакции.
func (tx *Transaction) hash() []byte {
    return CnFastHash(bytes.Join(
        [][]byte{
            tx.RandBytes,
            tx.PrevBlock,
            []byte(tx.Sender),
            []byte(tx.Receiver),
            ToBytes(tx.Value),
        },

```

```

    []byte{ },
    ))
}

func (tx *Transaction) sign(priv *ed25519.PrivateKey) []byte {
    return Sign(priv, tx.CurrHash)
}

// Данная функция предназначена для преобразования публичного ключа типа
// *ed25519.PublicKey` в строковое представление.
func StringPublic(pub *ed25519.PublicKey) string {
    keyBytes := make([]byte, ed25519.PrivateKeySize)
    copy(keyBytes[:], *pub)
    return base64.StdEncoding.EncodeToString(keyBytes)
}

// Данная функция вычисляет хеш с помощью алгоритма CryptoNight.
func CnFastHash(data []byte) []byte {
    out := blake2b.Sum256(data)
    result := make([]byte, 4)
    binary.LittleEndian.PutUint32(result, binary.LittleEndian.Uint32(out[0:4]))
    return result
}

func ToBytes(num uint64) []byte {
    var data = new(bytes.Buffer)
    err := binary.Write(data, binary.BigEndian, num)
    if err != nil {
        return nil
    }
    return data.Bytes()
}

func Sign(privKey *ed25519.PrivateKey, data []byte) []byte {
    ringKeys := make([]ed25519.PrivateKey, 5)
    for i := 0; i < 5; i++ {
        randKey := make([]byte, 32)
        rand.Read(randKey)
        ringKeys[i] = ed25519.NewKeyFromSeed(randKey)
    }
    var pubKeys []ed25519.PublicKey
    for i := 0; i < 5; i++ {
        pubKey :=
ed25519.PrivateKey(ringKeys[i].Seed()).Public().(ed25519.PublicKey)

```

```

    pubKeys = append(pubKeys, pubKey)
}
var signature []byte
for i := range pubKeys {
    tmpKeys := make([]ed25519.PrivateKey, 5)
    copy(tmpKeys, ringKeys)
    tmpKeys[i] = *privKey
    if len(tmpKeys[i]) != ed25519.PrivateKeySize {
        continue
    }
    tmpSignature := ed25519.Sign(tmpKeys[i], data)
    signature = append(signature, tmpSignature[:])
}
return signature
}

// Данная функция добавляет транзакцию с учетом криптовалютной эмиссии
func (block *Block) AddTransaction(chain *BlockChain, tx *Transaction) error {
    if tx == nil {
        return errors.New("tx is null")
    }
    if tx.Value == 0 {
        return errors.New("tx value = 0")
    }
    if tx.Emission != emission(chain.Size()) {
        return errors.New("emission value is invalid")
    }
    if !bytes.Equal(tx.PrevBlock, chain.LastBlockHash()) {
        return errors.New("prev block in tx != last hash in chain")
    }
    balanceInChain := chain.Balance(tx.Sender, chain.Size())
    balanceInTX := tx.Value + tx.Emission
    if balanceInTX > balanceInChain {
        return errors.New("insufficient funds")
    }
    block.Mapping[tx.Sender] = balanceInChain - balanceInTX
    block.addBalance(chain, tx.Receiver, tx.Value)
    block.addBalance(chain, tx.Receiver, tx.Emission)
    block.Transactions = append(block.Transactions, *tx)
    return nil
}

func (chain *BlockChain) LastBlockHash() []byte {
    var lastBlockHash []byte

```



```

    err := chain.DB.QueryRow("SELECT Hash FROM BlockChain ORDER BY Id
DESC LIMIT 1").Scan(&lastBlockHash)
    if err != nil {
        log.Panic(err)
    }
    return lastBlockHash
}

```

```

func (chain *BlockChain) Balance(address string, size uint64) uint64 {
    var (
        sblock string
        block  *Block
        balance uint64
    )
    rows, err := chain.DB.Query("SELECT Block FROM BlockChain WHERE Id
<= $1 ORDER BY Id DESC", size)
    if err != nil {
        return balance
    }
    defer rows.Close()
    for rows.Next() {
        rows.Scan(&sblock)
        block = DeserializeBlock(sblock)
        if value, ok := block.Mapping[address]; ok {
            balance = value
            break
        }
    }
    return balance
}

```

```

func (block *Block) addBalance(chain *BlockChain, receiver string, value uint64)
{
    var balanceInChain uint64
    if v, ok := block.Mapping[receiver]; ok {
        balanceInChain = v
    } else {
        balanceInChain = chain.Balance(receiver, chain.Size())
    }
    block.Mapping[receiver] = balanceInChain + value
}

```

```

func (chain *BlockChain) Size() uint64 {
    var size uint64

```

```

    row := chain.DB.QueryRow("SELECT Id FROM BlockChain ORDER BY Id
DESC")
    row.Scan(&size)
    return size
}

func DeserializeBlock(data string) *Block {
    var block Block
    err := json.Unmarshal([]byte(data), &block)
    if err != nil {
        return nil
    }
    return &block
}

func (block *Block) Accept(chain *BlockChain, user *User, ch chan bool) error {
    if !block.transactionsIsValid(chain, chain.Size()) {
        return errors.New("transactions is not valid")
    }
    block.AddTransaction(chain, &Transaction{
        RandBytes: GenerateRandomBytes(RAND_BYTES),
        PrevBlock: chain.LastHash(),
        Sender:    "",
        Receiver:  user.Address(),
        Value:     chain.Size(),
    })
    block.TimeStamp = time.Now().Format(time.RFC3339)
    block.CurrHash = block.hash()
    block.Signature = block.sign(user.Private())
    block.Nonce = block.proof(ch)
    return nil
}

func (block *Block) transactionsIsValid(chain *BlockChain, size uint64) bool {
    lentxs := len(block.Transactions)
    chainSize := chain.Size()
    em := emission(chainSize)
    str := strconv.FormatUint(em, 10)
    plusStorage := 0
    for i := 0; i < lentxs; i++ {
        if block.Transactions[i].Sender == str {
            plusStorage = 1
            break
        }
    }
}

```

```

    }
    if lentxs == 0 || lentxs > TXS_LIMIT+plusStorage {
        return false
    }
    for i := 0; i < lentxs-1; i++ {
        for j := i + 1; j < lentxs; j++ {
            if bytes.Equal(block.Transactions[i].RandBytes,
block.Transactions[j].RandBytes) {
                return false
            }
            if block.Transactions[i].Sender == str &&
                block.Transactions[j].Sender == str {
                return false
            }
        }
    }
    for i := 0; i < lentxs; i++ {
        tx := block.Transactions[i]
        if tx.Sender == str {
            if tx.Receiver != block.Miner || tx.Value != emission(chainSize) {
                return false
            }
        } else {
            if !tx.hashIsValid() {
                return false
            }
            if !tx.signIsValid() {
                return false
            }
        }
        if !block.balanceIsValid(chain, tx.Sender, size) {
            return false
        }
        if !block.balanceIsValid(chain, tx.Receiver, size) {
            return false
        }
    }
    return true
}

```

// Данная функция вычисляет хеш блока (структуры данных), содержащей информацию о транзакциях.

```

func (block *Block) hash() []byte {
    var tempHash []byte

```

```

for _, tx := range block.Transactions {
    tempHash = CnFastHash(bytes.Join(
        [][]byte{
            tempHash,
            tx.CurrHash,
        },
        []byte{ },
    ))
}
var list []string
for hash := range block.Mapping {
    list = append(list, hash)
}
sort.Strings(list)
for _, hash := range list {
    tempHash = CnFastHash(bytes.Join(
        [][]byte{
            tempHash,
            []byte(hash),
            ToBytes(block.Mapping[hash]),
        },
        []byte{ },
    ))
}
return CnFastHash(bytes.Join(
    [][]byte{
        tempHash,
        ToBytes(uint64(block.Difficulty)),
        block.PrevHash,
        []byte(block.Miner),
        []byte(block.TimeStamp),
    },
    []byte{ },
))
}

func (block *Block) sign(priv *ed25519.PrivateKey) []byte {
    return Sign(priv, block.CurrHash)
}

func (block *Block) proof(ch chan bool) uint64 {
    return ProofOfWork(block.CurrHash, block.Difficulty, ch)
}

```

```

func (tx *Transaction) hashIsValid() bool {
    return bytes.Equal(tx.hash(), tx.CurrHash)
}

func (tx *Transaction) signIsValid() bool {
    return Verify(ParsePublic(tx.Sender), tx.CurrHash, tx.Signature) == nil
}

func (block *Block) balanceIsValid(chain *BlockChain, address string, size
uint64) bool {
    if _, ok := block.Mapping[address]; !ok {
        return false
    }
    lentxs := len(block.Transactions)
    balanceInChain := chain.Balance(address, size)
    balanceSubBlock := uint64(0)
    balanceAddBlock := uint64(0)
    for j := 0; j < lentxs; j++ {
        tx := block.Transactions[j]
        if tx.Sender == address {
            balanceSubBlock += tx.Value + tx.Emission
        }
        if tx.Receiver == address {
            balanceAddBlock += tx.Value
        }
    }
    if (balanceInChain + balanceAddBlock - balanceSubBlock) !=
block.Mapping[address] {
        return false
    }
    return true
}

func ProofOfWork(blockHash []byte, difficulty uint8, ch chan bool) uint64 {
    var (
        Target = big.NewInt(1)
        intHash = big.NewInt(1)
        nonce = uint64(mrand.Intn(math.MaxUint32))
        hash []byte
    )
    Target.Lsh(Target, 256-uint(difficulty))
    for nonce < math.MaxUint64 {
        select {

```

```

case <-ch:
    if DEBUG {
        fmt.Println()
    }
    return nonce
default:
    hash = CnFastHash(bytes.Join(
        [][]byte{
            blockHash,
            ToBytes(nonce),
        },
        []byte{ },
    ))
    if DEBUG {
        fmt.Printf("\rMining: %s", Base64Encode(hash))
    }
    intHash.SetBytes(hash)
    if intHash.Cmp(Target) == -1 {
        if DEBUG {
            fmt.Println()
        }
        return nonce
    }
    nonce++
}
}
return nonce
}

// Данная функция проверяет подпись для данных с использованием
открытого ключа.
func Verify(pub *ed25519.PublicKey, data, sign []byte) error {
    if len(*pub) != ed25519.PublicKeySize {
        return errors.New("invalid public key size")
    }
    if !ed25519.Verify(*pub, data, sign) {
        return errors.New("signature verification failed")
    }
    return nil
}

// Данная функция перевод строку в публичный ключ.
func ParsePublic(pubData string) *ed25519.PublicKey {
    keyBytes, err := base64.StdEncoding.DecodeString(pubData)

```

```

    if err != nil {
        fmt.Println("Error decoding public key:", err)
        return nil
    }
    if len(keyBytes) != ed25519.PublicKeySize*2 {
        fmt.Println("Public key has unexpected length:", len(keyBytes))
        return nil
    }
    pubKey := ed25519.PublicKey(keyBytes[:ed25519.PublicKeySize])
    return &pubKey
}

func Base64Decode(data string) []byte {
    result, err := base64.StdEncoding.DecodeString(data)
    if err != nil {
        return nil
    }
    return result
}

// Данная функция создает публичный и приватный ключ на основе
// эллиптической кривой (Curve25519).
func GeneratePrivate() (*ed25519.PublicKey, *ed25519.PrivateKey) {
    pubKey, privKey, err := ed25519.GenerateKey(rand.Reader)
    if err != nil {
        return nil, nil
    }
    return &pubKey, &privKey
}

// Данная функция предназначена для преобразования приватного ключа типа
// `*ed25519.PrivateKey` в строковое представление.
func StringPrivate(priv *ed25519.PrivateKey) string {
    keyBytes := make([]byte, ed25519.PrivateKeySize)
    copy(keyBytes[:], *priv)
    return base64.StdEncoding.EncodeToString(keyBytes)
}

// Данная функция перевод строку в приватный ключ.
func ParsePrivate(privData string) *ed25519.PrivateKey {
    keyBytes, err := base64.StdEncoding.DecodeString(privData)
    if err != nil {
        return nil
    }
}

```

```

    if len(keyBytes) != ed25519.PrivateKeySize {
        return nil
    }
    privKey := new(ed25519.PrivateKey)
    copy((*privKey)[:], keyBytes)
    return privKey
}

// Данная функция создает нового пользователя
func NewUser() *User {
    pub, priv := GeneratePrivate()
    return &User{
        PublicKey: pub,
        PrivateKey: priv,
    }
}

func LoadUser(purse, pubKey string) *User {
    priv := ParsePrivate(purse)
    pub := ParsePublic(pubKey)
    if priv == nil {
        return nil
    }
    return &User{
        PublicKey: pub,
        PrivateKey: priv,
    }
}

func (user *User) Purse() string {
    return StringPrivate(user.Private())
}

func (user *User) Pub() string {
    return StringPublic(user.Public())
}

func (chain *BlockChain) LastHash() []byte {
    var hash string
    row := chain.DB.QueryRow("SELECT Hash FROM BlockChain ORDER BY Id DESC")
    row.Scan(&hash)
    return Base64Decode(hash)
}

```



```

func (block *Block) IsValid(chain *BlockChain, size uint64) bool {
    switch {
    case block == nil:
        return false
    case block.Difficulty != DIFFICULTY:
        return false
    case !block.hashIsValid(chain, size):
        return false
    case !block.signIsValid():
        return false
    case !block.proofIsValid():
        return false
    case !block.mappingIsValid():
        return false
    case !block.timeIsValid(chain):
        return false
    case !block.transactionsIsValid(chain, size):
        return false
    }
    return true
}

func SerializeTX(tx *Transaction) string {
    jsonData, err := json.MarshalIndent(tx, "", "\t")
    if err != nil {
        return ""
    }
    return string(jsonData)
}

func DeserializeTX(data string) *Transaction {
    var tx Transaction
    err := json.Unmarshal([]byte(data), &tx)
    if err != nil {
        return nil
    }
    return &tx
}

func (block *Block) hashIsValid(chain *BlockChain, size uint64) bool {
    if !bytes.Equal(block.hash(), block.CurrHash) {
        return false
    }
}

```

```

    var id uint64
    row := chain.DB.QueryRow("SELECT Id FROM BlockChain WHERE
Hash=$1",
    Base64Encode(block.PrevHash))
    row.Scan(&id)
    return id == size
}

func (block *Block) signIsValid() bool {
    return Verify(ParsePublic(block.Miner), block.CurrHash, block.Signature) == nil
}

func (block *Block) proofIsValid() bool {
    intHash := big.NewInt(1)
    Target := big.NewInt(1)
    hash := CnFastHash(bytes.Join(
    [][]byte{
        block.CurrHash,
        ToBytes(block.Nonce),
    },
    []byte{ },
    ))
    intHash.SetBytes(hash)
    Target.Lsh(Target, 256-uint(block.Difficulty))
    if intHash.Cmp(Target) == -1 {
        return true
    }
    return false
}

func (block *Block) mappingIsValid() bool {
    for hash := range block.Mapping {
        flag := false
        for _, tx := range block.Transactions {
            if tx.Sender == hash || tx.Receiver == hash {
                flag = true
                break
            }
        }
        if !flag {
            return false
        }
    }
    return true
}

```

```

}

func (block *Block) timeIsValid(chain *BlockChain) bool {
    btime, err := time.Parse(time.RFC3339, block.TimeStamp)
    if err != nil {
        return false
    }
    diff := time.Now().Sub(btime)
    if diff < 0 {
        return false
    }
    var sblock string
    row := chain.DB.QueryRow("SELECT Block FROM BlockChain WHERE
Hash=$1",
        Base64Encode(block.PrevHash))
    row.Scan(&sblock)
    lblock := DeserializeBlock(sblock)
    if lblock == nil {
        return false
    }
    ltime, err := time.Parse(time.RFC3339, lblock.TimeStamp)
    if err != nil {
        return false
    }
    result := btime.Sub(ltime)
    return result > 0
}
.

```

## ПРИЛОЖЕНИЕ В

Клиентская программа для взаимодействия с блокчейном

```
package main

import (
    bc "CrystalCoin/blockchain"
    nt "CrystalCoin/network"
    "bufio"
    "encoding/json"
    "fmt"
    "io/ioutil"
    "os"
    "strconv"
    "strings"
)

func init() {
    if len(os.Args) < 2 {
        panic("failed: len(os.Args) < 2")
    }
    var (
        addrStr    = ""
        userNewStr = ""
        userLoadStr = ""
    )
    var (
        addrExist    = false
        userNewExist = false
        userLoadExist = false
    )
    for i := 1; i < len(os.Args); i++ {
        arg := os.Args[i]
        switch {
        case strings.HasPrefix(arg, "-loadaddr:"):
            addrStr = strings.Replace(arg, "-loadaddr:", "", 1)
            addrExist = true
        case strings.HasPrefix(arg, "-newuser:"):
            userNewStr = strings.Replace(arg, "-newuser:", "", 1)
            userNewExist = true
        case strings.HasPrefix(arg, "-loaduser:"):
            userLoadStr = strings.Replace(arg, "-loaduser:", "", 1)
        }
```

```

        userLoadExist = true
    }
}
if !(userNewExist || userLoadExist) || !addrExist {
    panic("failed: !(userNewExist || userLoadExist) || !addrExist")
}
err := json.Unmarshal([]byte(readFile(addrStr)), &Addresses)
if err != nil {
    panic("failed: load addresses")
}
if len(Addresses) == 0 {
    panic("failed: len(Addresses) == 0")
}
if userNewExist {
    User = userNew(userNewStr)
}
if userLoadExist {
    User = userLoad(userLoadStr)
}
if User == nil {
    panic("failed: load user")
}
}

func readFile(filename string) string {
    data, err := ioutil.ReadFile(filename)
    if err != nil {
        return ""
    }
    return string(data)
}

func userNew(filename string) *bc.User {
    user := bc.NewUser()
    if user == nil {
        return nil
    }
    err := writeFile(filename, user.Purse())
    if err != nil {
        return nil
    }
    return user
}

```

```

func userLoad(filename string) *bc.User {
    keyBytes, err := ioutil.ReadFile(filename)
    if err != nil {
        return nil
    }
    keys := strings.Split(string(keyBytes), "\n")
    privKey := keys[0]
    pubKey := keys[1]
    user := bc.LoadUser(privKey, pubKey)
    if err != nil {
        return nil
    }
    return user
}

var (
    Addresses []string
    User      *bc.User
)

func writeFile(filename string, data string) error {
    return ioutil.WriteFile(filename, []byte(data), 0644)
}

func handleClient() {
    var (
        message string
        splited []string
    )
    for {
        message = inputString("> ")
        splited = strings.Split(message, " ")
        switch splited[0] {
            case "/exit":
                os.Exit(0)
            case "/user":
                if len(splited) < 2 {
                    fmt.Println("failed: len(user) < 2\n")
                    continue
                }
                switch splited[1] {
                    case "address":
                        userAddress()
                    case "purse":

```

```

        userPurse()
    case "balance":
        userBalance()
    default:
        fmt.Println("command undefined\n")
    }
    case "/chain":
        if len(splited) < 2 {
            fmt.Println("failed: len(chain) < 2\n")
            continue
        }
        switch splited[1] {
        case "print":
            chainPrint()
        case "tx":
            chainTX(splited[1:])
        case "size":
            chainsize()
        default:
            fmt.Println("command undefined\n")
        }
    default:
        fmt.Println("command undefined\n")
    }
}

func inputString(begin string) string {
    fmt.Print(begin)
    msg, _ := bufio.NewReader(os.Stdin).ReadString('\n')
    return strings.Replace(msg, "\n", "", 1)
}

func userAddress() {
    fmt.Println("Address:", User.Pub(), "\n")
}

func userPurse() {
    fmt.Println("Purse:", User.Purse(), "\n")
}

func userBalance() {
    printBalance(User.Address())
}

```

```

func chainPrint() {
    for i := 0; ; i++ {
        res := nt.Send(Addresses[0], &nt.Package{
            Option: GET_BLOCK,
            Data:   fmt.Sprintf("%d", i),
        })
        if res == nil || res.Data == "" {
            break
        }
        fmt.Printf("[%d] => %s\n", i+1, res.Data)
    }
    fmt.Println()
}

func chainTX(splited []string) {
    if len(splited) != 3 {
        fmt.Println("failed: len(splited) != 3\n")
        return
    }
    splited[2] = strings.TrimRight(splited[2], "\r")
    num, err := strconv.Atoi(splited[2])
    if err != nil {
        fmt.Println("failed: strconv.Atoi(num)\n")
        return
    }
    for _, addr := range Addresses {
        res := nt.Send(addr, &nt.Package{
            Option: GET_LHASH,
        })
        if res == nil {
            continue
        }
        tx := bc.NewTransaction(User, bc.Base64Decode(res.Data), splited[1],
            uint64(num), chainsizeTrunsuct())
        res = nt.Send(addr, &nt.Package{
            Option: ADD_TRNSX,
            Data:   bc.SerializeTX(tx),
        })
        if res == nil {
            continue
        }
        if res.Data == "ok" {
            fmt.Printf("ok: (%s)\n", addr)
        }
    }
}

```



```

    } else {
        fmt.Printf("fail: (%s)\n", addr)
    }
}
fmt.Println()
}

func chainSize() {
    res := nt.Send(Addresses[0], &nt.Package{
        Option: GET_CSIZE,
    })
    if res == nil || res.Data == "" {
        fmt.Println("failed: getSize\n")
        return
    }
    fmt.Printf("Size: %s blocks\n\n", res.Data)
}

func chainSizeTransact() string {
    res := nt.Send(Addresses[0], &nt.Package{
        Option: GET_CSIZE,
    })
    if res == nil || res.Data == "" {
        fmt.Println("failed: getSize\n")
        return "failed: getSize"
    }
    return res.Data
}

func printBalance(userAddr string) {
    for _, addr := range Addresses {
        res := nt.Send(addr, &nt.Package{
            Option: GET_BLNCE,
            Data: userAddr,
        })
        if res == nil {
            continue
        }
        fmt.Printf("Balance (%s): %s coins\n", addr, res.Data)
    }
    fmt.Println()
}

const (

```

```
    ADD_BLOCK = iota + 1
    ADD_TRNSX
    GET_BLOCK
    GET_LHASH
    GET_BLNCE
    GET_CSIZE
)

func main() {
    handleClient()
}
```

## ПРИЛОЖЕНИЕ Г

Приложение реализующие логику криптовалютного узла.

```
package main

import (
    bc "Node/blockchain"
    nt "Node/network"
    "bytes"
    "database/sql"
    "encoding/hex"
    "encoding/json"
    "fmt"
    _ "github.com/mattn/go-sqlite3"
    "io"
    "io/ioutil"
    "os"
    "sort"
    "strconv"
    "strings"
    "sync"
)

func readFile(filename string) string {
    data, err := ioutil.ReadFile(filename)
    if err != nil {
        return ""
    }
    return string(data)
}

func userNew(filename string) *bc.User {
    user := bc.NewUser()
    if user == nil {
        return nil
    }
    f, err := os.Create(filename)
    if err != nil {
        return nil
    }
    fmt.Fprintln(f, user.Purse())
    fmt.Fprintln(f, user.Pub())
}
```

```

    return user
}

func userLoad(filename string) *bc.User {
    keyBytes, err := ioutil.ReadFile(filename)
    if err != nil {
        return nil
    }
    keys := strings.Split(string(keyBytes), "\n")
    privKey := keys[0]
    pubKey := keys[1]
    user := bc.LoadUser(privKey, pubKey)
    if err != nil {
        return nil
    }
    return user
}

var (
    Addresses []string
    User      *bc.User
)

func writeFile(filename string, data string) error {
    return ioutil.WriteFile(filename, []byte(data), 0644)
}

const (
    ADD_BLOCK = iota + 1
    ADD_TRNSX
    GET_BLOCK
    GET_LHASH
    GET_BLNCE
    GET_CSIZE
)

func init() {
    if len(os.Args) < 2 {
        panic("failed: len(os.Args) < 2")
    }
    var (
        serveStr    = ""
        addrStr      = ""
        userNewStr   = ""
    )
}

```

```

    userLoadStr = ""
    chainNewStr = ""
    chainLoadStr = ""
)
var (
    serveExist    = false
    addrExist     = false
    userNewExist  = false
    userLoadExist = false
    chainNewExist = false
    chainLoadExist = false
)
for i := 1; i < len(os.Args); i++ {
    arg := os.Args[i]
    switch {
    case strings.HasPrefix(arg, "-serve:"):
        serveStr = strings.Replace(arg, "-serve:", "", 1)
        serveExist = true
    case strings.HasPrefix(arg, "-loadaddr:"):
        addrStr = strings.Replace(arg, "-loadaddr:", "", 1)
        addrExist = true
    case strings.HasPrefix(arg, "-newuser:"):
        userNewStr = strings.Replace(arg, "-newuser:", "", 1)
        userNewExist = true
    case strings.HasPrefix(arg, "-loaduser:"):
        userLoadStr = strings.Replace(arg, "-loaduser:", "", 1)
        userLoadExist = true
    case strings.HasPrefix(arg, "-newchain:"):
        chainNewStr = strings.Replace(arg, "-newchain:", "", 1)
        chainNewExist = true
    case strings.HasPrefix(arg, "-loadchain:"):
        chainLoadStr = strings.Replace(arg, "-loadchain:", "", 1)
        chainLoadExist = true
    }
}
if !(userNewExist || userLoadExist) || !(chainNewExist || chainLoadExist) ||
    !serveExist || !addrExist {
    panic("failed: !(userNewExist || userLoadExist)" +
        "|| !(chainNewExist || chainLoadExist) || !serveExist || !addrExist")
}

Serve = serveStr
var addresses []string
err := json.Unmarshal([]byte(readFile(addrStr)), &addresses)

```

```

if err != nil {
    panic("failed: load addresses")
}
var mapaddr = make(map[string]bool)
for _, addr := range addresses {
    if addr == Serve {
        continue
    }
    if _, ok := mapaddr[addr]; ok {
        continue
    }
    mapaddr[addr] = true
    Addresses = append(Addresses, addr)
}
if userNewExist {
    User = userNew(userNewStr)
}
if userLoadExist {
    User = userLoad(userLoadStr)
}
if User == nil {
    panic("failed: load user")
}
if chainNewExist {
    Filename = chainNewStr
    Chain = chainNew(chainNewStr)
}
if chainLoadExist {
    Filename = chainLoadStr
    Chain = chainLoad(chainLoadStr)
}
if Chain == nil {
    panic("failed: load chain")
}
Block = bc.NewBlock(User.Address(), Chain.LastHash())
}

var (
    Filename string
    Serve    string
    Chain    *bc.BlockChain
    Block    *bc.Block
)

```

```

func chainNew(filename string) *bc.BlockChain {
    err := bc.NewChain(filename, User.Address())
    if err != nil {
        return nil
    }
    return bc.LoadChain(filename)
}

func chainLoad(filename string) *bc.BlockChain {
    chain := bc.LoadChain(filename)
    if chain == nil {
        return nil
    }
    return chain
}

func main() {
    nt.Listen(Serve, handleServer)
    for {
        fmt.Scanln()
    }
}

func handleServer(conn nt.Conn, pack *nt.Package) {
    nt.Handle(ADD_BLOCK, conn, pack, addBlock)
    nt.Handle(ADD_TRNSX, conn, pack, addTransaction)
    nt.Handle(GET_BLOCK, conn, pack, getBlock)
    nt.Handle(GET_LHASH, conn, pack, getLastHash)
    nt.Handle(GET_BLNCE, conn, pack, getBalance)
    nt.Handle(GET_CSIZE, conn, pack, getChainSize)
}

func addBlock(pack *nt.Package) string {
    splited := strings.Split(pack.Data, SEPARATOR)
    if len(splited) != 3 {
        return "fail"
    }
    block := bc.DeserializeBlock(splited[2])
    if !block.IsValid(chain, chain.Size()) {
        currSize := chain.Size()
        num, err := strconv.Atoi(splited[1])
        if err != nil {
            return "fail"
        }
    }
}

```

```

    if currSize < uint64(num) {
        go compareChains(splited[0], uint64(num))
        return "ok "
    }
    return "fail"
}
Mutex.Lock()
Chain.AddBlock(block)
Block = bc.NewBlock(User.Address(), Chain.LastHash())
Mutex.Unlock()
if IsMining {
    BreakMining <- true
    IsMining = false
}
return "ok"
}

func addTransaction(pack *nt.Package) string {
    var tx = bc.DeserializeTX(pack.Data)
    if tx == nil || len(Block.Transactions) == bc.TXS_LIMIT {
        return "fail"
    }
    Mutex.Lock()
    err := Block.AddTransaction(Chain, tx)
    Mutex.Unlock()
    if err != nil {
        return "fail"
    }
    if len(Block.Transactions) == bc.TXS_LIMIT {
        go func() {
            Mutex.Lock()
            block := *Block
            IsMining = true
            Mutex.Unlock()
            res := (&block).Accept(Chain, User, BreakMining)
            Mutex.Lock()
            IsMining = false
            if res == nil && bytes.Equal(block.PrevHash, Block.PrevHash) {
                Chain.AddBlock(&block)
                pushBlockToNet(&block)
            }
            Block = bc.NewBlock(User.Address(), Chain.LastHash())
            Mutex.Unlock()
        }()
    }
}

```



```

    }
    return "ok"
}

func getBlock(pack *nt.Package) string {
    num, err := strconv.Atoi(pack.Data)
    if err != nil {
        return ""
    }
    size := Chain.Size()
    if uint64(num) < size {
        return selectBlock(Chain, num)
    }
    return ""
}

func getLastHash(pack *nt.Package) string {
    return bc.Base64Encode(Chain.LastHash())
}

func getBalance(pack *nt.Package) string {
    return fmt.Sprintf("%d", Chain.Balance(pack.Data, Chain.Size()))
}

func getChainSize(pack *nt.Package) string {
    return fmt.Sprintf("%d", Chain.Size())
}

const (
    SEPARATOR = "_SEPARATOR_"
)

var (
    IsMining    bool
    BreakMining = make(chan bool)
)

func compareChains(address string, num uint64) {
    filename := "temp_" + hex.EncodeToString(bc.GenerateRandomBytes(8))
    file, err := os.Create(filename)
    if err != nil {
        return
    }
    file.Close()
}

```

```

defer func() {
    os.Remove(filename)
}()
res := nt.Send(address, &nt.Package{
    Option: GET_BLOCK,
    Data:   fmt.Sprintf("%d", 0),
})
if res == nil {
    return
}
genesis := bc.DeserializeBlock(res.Data)
if genesis == nil {
    return
}
if !bytes.Equal(genesis.CurrHash, hashBlock(genesis)) {
    return
}
db, err := sql.Open("sqlite3", filename)
if err != nil {
    return
}
defer db.Close()
_, err = db.Exec(bc.CREATE_TABLES)
chain := &bc.BlockChain{
    DB: db,
}
chain.AddBlock(genesis)
for i := uint64(1); i < num; i++ {
    res := nt.Send(address, &nt.Package{
        Option: GET_BLOCK,
        Data:   fmt.Sprintf("%d", i),
    })
    if res == nil {
        return
    }
    block := bc.DeserializeBlock(res.Data)
    if block == nil {
        return
    }
    if !block.IsValid(chain, i) {
        return
    }
    chain.AddBlock(block)
}

```

```

Mutex.Lock()
Chain.DB.Close()
os.Remove(Filename)
copyFile(filename, Filename)
Chain = bc.LoadChain(Filename)
Block = bc.NewBlock(User.Address(), Chain.LastHash())
Mutex.Unlock()
if IsMining {
    BreakMining <- true
    IsMining = false
}
}

var (
    Mutex sync.Mutex
)

func pushBlockToNet(block *bc.Block) {
    var (
        sblock = bc.SerializeBlock(block)
        msg     = Serve + SEPARATOR + fmt.Sprintf("%d", Chain.Size()) +
SEPARATOR + sblock
    )
    for _, addr := range Addresses {
        go nt.Send(addr, &nt.Package{
            Option: ADD_BLOCK,
            Data:   msg,
        })
    }
}

func selectBlock(chain *bc.BlockChain, i int) string {
    var block string
    row := chain.DB.QueryRow("SELECT Block FROM BlockChain WHERE
Id=$1", i+1)
    row.Scan(&block)
    return block
}

func hashBlock(block *bc.Block) []byte {
    var tempHash []byte
    for _, tx := range block.Transactions {
        tempHash = bc.CnFastHash(bytes.Join(
            [][]byte{

```

```

        tempHash,
        tx.CurrHash,
    },
    []byte{ },
))
}
var list []string
for hash := range block.Mapping {
    list = append(list, hash)
}
sort.Strings(list)
for _, hash := range list {
    tempHash = bc.CnFastHash(bytes.Join(
        [][]byte{
            tempHash,
            []byte(hash),
            bc.ToBytes(block.Mapping[hash]),
        },
        []byte{ },
    ))
}
return bc.CnFastHash(bytes.Join(
    [][]byte{
        tempHash,
        bc.ToBytes(uint64(block.Difficulty)),
        block.PrevHash,
        []byte(block.Miner),
        []byte(block.TimeStamp),
    },
    []byte{ },
))
}

func copyFile(src, dst string) error {
    in, err := os.Open(src)
    if err != nil {
        return err
    }
    defer in.Close()
    out, err := os.Create(dst)
    if err != nil {
        return err
    }
    defer out.Close()

```

```
_, err = io.Copy(out, in)
if err != nil {
    return err
}
return out.Close()
}
```

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ  
федеральное государственное автономное образовательное учреждение высшего образования  
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

ОТЗЫВ РУКОВОДИТЕЛЯ  
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

на тему Разработка криптовалюты с применением технологии CryptoNote

выполненную студентом группы № 1941

Клейменовым Никитой Александровичем

фамилия, имя, отчество студента

по направлению подготовки/ 09.03.01  
специальности (код)

Информатика

(наименование направления подготовки/специальности)

и вычислительная техника

(наименование направления подготовки/специальности)

Актуальность темы работы:

Тема данной работы является актуальной поскольку криптовалюты становятся все более популярными и востребованными как средство инвестирования и хранения денежных средств. Технология CryptoNote является одной из самых инновационных и безопасных технологий, которая обеспечивает анонимность и защиту данных пользователей. Разработка криптовалюты с применением технологии CryptoNote может быть полезна как для инвесторов, так и для разработчиков, которые могут использовать эту технологию для создания безопасных и надежных криптовалют. Кроме того, разработка новых криптовалют с использованием технологии CryptoNote может способствовать развитию экономики и бизнеса в целом, поскольку это позволяет создавать новые инструменты для финансовых операций и улучшать качество обслуживания пользователей.

Цель и задачи работы:

Целью бакалаврской работы является реализация криптовалюты с применением технологии CryptoNote. Для достижения этой цели были поставлены и решены следующие задачи:

1. Разработка криптовалютного блокчейна;
2. Применение криптографических методов для повышения безопасности пользователей и транзакций таких как CryptoNote, эллиптические кривые, алгоритм blake2b;
3. Реализация приложения криптографического узла;
4. Реализация клиентского приложения для работы с блокчейном;
5. Реализация сетевой библиотеки для работы с блокчейном

Общая оценка выполнения поставленной перед студентом задачи, основные достоинства и недостатки работы:

В бакалаврской работе раскрыт вопрос построения блокчейна и взаимодействия пользователей и криптографических узлов с ним. Были разобраны и применены криптографические алгоритмы в том числе технология CryptoNote для повышения защищенности пользователей и транзакций внутри блокчейна.

Была разработана программа блокчейна, клиента, криптографического узла и сетевой

библиотеки для работы с блокчейном.

Степень самостоятельности и способности к исследовательской работе студента (умение и навыки поиска, обобщения, анализа материала и формулирования выводов):

Студент Клейменов Н.А. выполнил работу на высоком техническом уровне, показал знания предметной области, продемонстрировал применение системного анализа, представил законченное решение, программные коды которого обладают актуальностью.

Проверка текста выпускной квалификационной работы с использованием системы Антиплагиат ВУЗ, проводившаяся «05» июня 2023 г. показывает оригинальность содержания на уровне 95.97%.

Степень грамотности изложения и оформления материала:

Материал представлен структурировано, задачи обоснованы, представлены практические результаты и фрагменты исходного кода программной реализации криптовалюты, текст оформлен в соответствии с требованиями ВКР ГУАП.

Оценка деятельности студента в период подготовки выпускной квалификационной работы (добросовестность, работоспособность, ответственность, аккуратность и т.п.):

Студент выполнил исследование в рамках выпускной квалификационной бакалаврской работы добросовестно, инициативно, ответственно, аккуратно.

Общий вывод:


Бакалаврская работа выполнена на высоком техническом уровне, а ее автор Клейменов Никита Александрович заслуживает оценки «отлично» и присуждения «Бакалавра» по направлению 09.03.01 «Информатика и вычислительная техника».

*В работе не содержится информация с ограниченным доступом и отсутствуют сведения, представляющие коммерческую ценность.*

Руководитель

доцент, к.т.н.

должность, уч. степень, звание

 08.06.2023

подпись, дата

Курицын К.А.

инициалы, фамилия