Criptografia de dados em um contexto de blockchain

Adson Nogueira Alves[1],

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Campinas – SP – Brazil

á220655@dac.unicamp.br

Abstract. *Blockchain has been among the technologies that most increse academic and professional interest, characteristics such as reliability, security and data tracking have boosted this sector. The objective of the work is to carry out a study and analysis of cryptographic techniques in the context of blockchain. To validate the technique studied, a basic blockchain algorithm was tested, the code was commented and made available in this work. Available on Github.*

Resumo. *Blockchain tem estado entre as tecnologias que mais despertão interesse acadêmico e profissional, características como confiabilidade, sigilo e rastreabilidade dos dados tem impulsionado esse setor. O objetivo do trabalho é realizar um estudo e análise das técnicas criptográficas no contexto de blockchain. Para validar a técnica estudada um algoritmo básico de blockchain foi testado, o código foi comentado e disponibilizado nesta trabalho. Disponível no Github.*

1. Introdução

O sucesso da técnica Blockchain está ligado ao sucesso financeiro da criptomoeda Bitcoin [Ghimire and Selvaraj 2018] que foi desenvolvido há poucos anos e após isso houve uma avalanche de mais de 2.140 outras criptomoedas que juntas construíram um mercado financeiro no valor de cerca de US$ 285 bilhões (em 16 de junho de 2019). O objetivo de criptomoedas como bitcoin é apoiar transações financeiras sem a necessidade de um banco central. A ideia dessa tecnologia é utilizar um livro público de transações que é mantido, verificado e atualizado voluntariamente por milhões de dispositivos que utilizam a estrutura. Este livro público é chamado de Blockchain.

Uma transação, neste contexto, é a transferência de uma quantia fixa de dinheiro digital (podendo ser qualquer criptomoeda) de uma conta para outra. A conta é o endereço da criptomoeda, que é apenas uma sequência aleatória de números; É na verdade um resumo de mensagem obtido por *hash* de uma chave pública (de verificação) para um esquema de assinatura, faleremos sobre a função *hash* adiante. Dessa forma, uma transação pode ser imaginada como uma mensagem M com um formato apresentado em 1:

$$transferirXbitcoinsdoendereçoA1paraoendereçoA2. \tag{1}$$

Assim como definido em [Raikwar et al. 2019], Blockchain é um livro-razão distribuído que mantém uma lista crescente de registros de dados que são confirmados por todos os nós participantes. Os dados são registrados nesse livro público na forma de

blocos de validação de transação, e esse livro público é compartilhado e disponível para todos os nós da estrutura.

Blockchain encontra muitos desafios de que devem ser superados. Os principais seriam melhora de segurança e privacidade, gerenciamento de chaves, escalabilidade, análise de novos ataques, gerenciamento de contratos inteligentes e introdução incremental de novos recursos criptográficos em Blockchains existentes.

Este trabalho foi organizado da seguinte forma: na seção 2 trabalhamos a fundamentação teórica de criptografia e blockchain; seção 2.1 tratamos dos trabalhos relacionados e técnicas que estão sendo utilizadas e por último na seção 4 temos a conclusão do trabalho.

2. Fundamentação teórica

Antes de conversar de conceitos particulares do blockchain acredito que seja importante fundamentar alguns aspectos de criptografia de dados, entre eles a **criptografia de chave** pública. A chave pública poderia ser usada para encriptar a mensagem original e a chave privada possibilita que o texto cifrado seja desencriptado pelo destinatário. Note que a chave pública é conhecida por todos, assim qualquer pessoa poderia encriptar a mensagem que será transmitida, porém apenas o destinatário consegue desencriptar. Um robusto sistema de criptografia de chave publica é o RSA [Mallouli et al. 2019].

Outro aspecto seria a **integridade da mensagem** que inclui códigos de autenticação de mensagem (MAC's), esquema de assinatura e função *hash*. O **MAC** requer que ambos, emissor e destinatário compartilhem uma mesma chave secreta para criar uma TAG que é adicionada a mensagem. Assim quando o destinatário poderá verificar a autenticidade da mensagem. Note que a mensagem pode ou não ser encriptada. O *esquema de assinatura*, na configuração de chave pública fornece garantia similar ao fornecido por um MAC. Neste caso a chave privada especifica um algorítimo de assinatura para a mensagem, e a saída produzida é chamada de assinatura que é anexada a mensagem. Para validar se a mensagem deve ser aceita ou não é utilizado um algoritmo de verificação que recebe como entrada a mensagem e a assinatura e emite *True* ou *False*. A vantagem é que qualquer pessoa pode verificar as assinaturas, em contraste, na configuração MAC, que apenas o destinatário poderia verificar as TAG's criadas.

É importante comentar do **Não repúdio** de mensagem, observe que há uma diferença sutil entre MACs e esquemas de assinatura. No esquema de assinatura, o algoritmo de verificação é público, logo poderá ser validada por todos. Dessa forma quem envia não pode alegar que ela não assinou, para isso usamos o termo **não repúdio**. No caso do MAC, isso não seria possível visto que a chave é secreta, conhecida apenas por quem envia e recebe.

Por último o **Certificado** é uma ferramenta que ajuda a verificar a autenticidade das chaves públicas. Nele contém informações sobre um usuário específico ou, mais comumente, um site, incluindo as chaves públicas do site. Essas chaves públicas serão assinadas por uma autoridade confiável. Presume-se que todos possuam a chave de verificação pública da autoridade confiável, para que qualquer pessoa possa verificar a assinatura da autoridade confiável em um certificado.

Os esquemas de assinatura tendem a ser muito menos eficientes que os MACs.

A maioria dos esquemas de assinatura são projetados para assinar apenas mensagens de comprimento fixo e curto. Antes de serem assinadas uma função de hash criptográfica é usada para compactar uma mensagem de tamanho arbitrário em um resumo de mensagem curto, de aparência aleatória e de tamanho fixo. Assim, a função **hash** é uma função pública que é considerada conhecida por todos, além de não ter chave. Uma função de hash criptográfica é diferente de uma função de hash usada para construir uma tabela de hash, não é o foco deste trabalho aprofundar no tema, mais informações podem ser consultadas em [Raikwar et al. 2019].

2.1. Fundamentação Blockchain

Blockchain é uma maneira de encapsular transações na forma de blocos onde os blocos são vinculados através do hash criptográfico, o que forma uma cadeia de blocos. A figura 1 mostra a estrutura de dados do blockchain do Bitcoin mostrando em detalhes o formato do bloco.

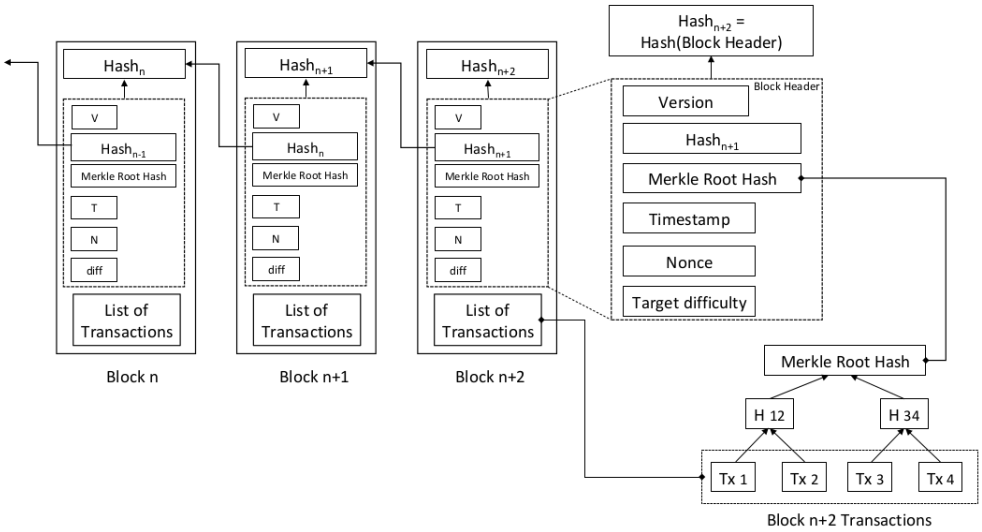

Figure 1. Formato de blocos - Estrutura Blockchain. [Raikwar et al. 2019]

Cada bloco contém um cabeçalho de bloco e uma representação da transação, cada bloco consiste em seu hash, o hash do bloco anterior, um carimbo de data/hora e alguns outros campos de bloco como versão, nonce etc.

O **hash root Merkle** representa o conjunto de transações na árvore Merkle, e essa representação de transações varia de acordo com o design da implementação do blockchain.

Os principais desafios na área são: Melhoria constante de segurança e privacidade, Gerenciamento de chaves, Escalabilidade, Análise de novos ataques, Gerenciamento de contratos inteligentes Introdução incremental de novos recursos criptográficos em Blockchains existentes.

Um fator importante no contexto de blockchain é o mecanismo de consenso. Um mecanismo de consenso refere-se a qualquer número de metodologias usadas para obter

acordo, confiança e segurança em uma rede de computadores descentralizada.O consenso é o componente chave do blockchain para sincronizar ou atualizar o livro-razão, chegando a um acordo entre os participantes. Muitos mecanismos de consenso foram propostos, alguns estão relacionados na Figura 2. No mecanismo de consenso novas transmissões são transmitidas para todos os nós, na sequência um ”lider” é escolhido a partir de um mecanismo de seleção, podendo ser um Puzzle Competion por exemplo; O líder então cria um bloco com todas as transações e transmite para todos; Baseado em múltiplos rounds de votação implícita ou explicita um consenso é alcançado no block; Por último todos os nós adicionam esse bloco na blockchain.

Leader Election Criteria	**Reference Protocols**
PoW Puzzle Competition	Bitcoin-NG [33], Casper [34], Proof of Stake velocity [35]
Verifiable Random Function	Tendermint [36], Algorand [37], Secure Proof of Stake [38]
Trusted Random Function	Proof of luck [39], Proof of elapsed time [40]
Modified Preimage Search	Snow White [41]
Sub-network of Masternodes / Validator nodes	Darkcoin and DASH [42], Libra [43]

Figure 2. Mecanismos de consenso propostos. Fonte: [Raikwar et al. 2019]

Muitas vezes, os nós da rede não podem chegar a um consenso unânime em relação ao estado futuro do blockchain. Este evento leva a bifurcações, significando aquele ponto em que a cadeia ’única’ ideal de blocos é dividida em duas ou mais cadeias válidas, como na Figura 3.

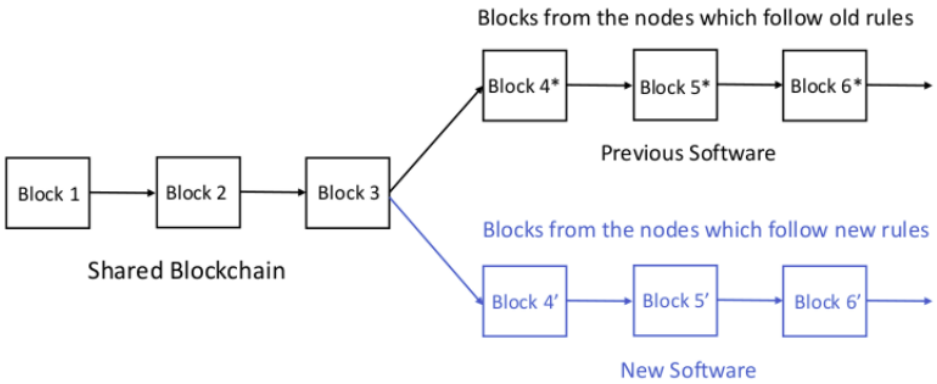

Figure 3. Bifurcação (Fork). Fonte: [Raikwar et al. 2019]

Dando origem a três tipos de bifurcações que podem ocorrer com base na compatibilidade com versões anteriores do protocolo blockchain e no instante de tempo em que um novo bloco é extraído. Esses tipos são os seguintes:

- Soft Fork: quando o protocolo blockchain é alterado de forma compatível com versões anteriores.
- Hard Fork: quando o protocolo blockchain é alterado de forma não compatível com versões anteriores.
- Garfo Temporário: quando dois mineradores mineram um novo bloco ao mesmo tempo.

É importante conversar sobre alguns tipos de arquiteturas que existem no blockchain:

1. Público sem permissão Qualquer pessoa pode entrar ou sair da rede a qualquer momento e participar do consenso também para manter o registro. E neste caso todos têm acesso de leitura e gravação ao blockchain.

2. Público autorizado: Todos podem ler o estado e os dados do blockchain, porém a gravação de dados e participa do consenso é restringida, ou seja, não teriam acesso.

3. Privado sem permissão: Colaborar sem a necessidade de compartilhar informações publicamente; Qualquer pessoa entrar ou sair do blockchain a qualquer momento; É definido quem tem permissão para ler o contrato e os dados relacionados.

4. Privado com permissão: Controle de acesso restrito / autorizado apenas por membros da organização; Acesso de leitura e gravação é fornecido pelo administrador de rede.

2.2. Primitivas criptográficas promissoras

A **Assinatura Agregada** segundo [Raikwar et al. 2019] Uma assinatura agregada permite criar uma única assinatura compacta com assinaturas e assinantes bem determinados. No contexto de blockchain assinaturas agregadas podem ser usadas para redução de armazenamento e computação, apesar de não ser trivial. As técnicas para agregar são bem conhecidas para uma variedade de esquema como DSA, Schnorr, baseado em pareamento e baseado em rede. A técnica ajuda a evitar o adversário de criar uma assinatura agregada válida por conta própria [Zhao 2018].

A Criptografia Baseada em Identidade é uma outra proposta que permite que a parte criptografadora use qualquer identidade conhecida de qualquer parte receptora como sua chave pública. Solicitando a um terceiro confiável "Gerador de Chave Privada (PKG)" para gerar a chave privada correspondente. A descriptografia da mensagem é feita usando a chave privada recebida [Boneh et al. 2005] [Goyal et al. 2006] [Malluhi et al. 2019]. Segundo [Raikwar et al. 2019], o IBE substituiu o papel da Infraestrutura de Chave Pública pelo PKG de terceiros confiáveis. Apesar da presença de um terceiro confiável não é interessante para o blockchain sem permissão, ele ainda pode ser usado no livro-razão distribuído, ou seja, é interessante no blockchain com permissão [Bose et al. 2018].

A Recuperação de Informações Privadas (PIR) é uma primitiva criptográfica na qual um cliente consulta um servidor e recupera a resposta correspondente do servidor

sem expor os termos de consulta e a resposta, seria outra técnica promissora, visto que pode facilitar consultas privadas de blockchain para buscar dados de transações de forma privada do blockchain [Kumar et al. 2016].

No trabalho [Raikwar et al. 2019] o autor cita diversas outra técnicas promissoras no contexto criptográfico e de blockchain.

3. Algoritmo

Para validar a técnica foi desenvolvido e testado um algoritmo básico de Blockchain. O algoritmo foi inspirado na proposta feita em [blo]. Antes é importante reforçar que o desenvolvimento de um projeto de blockchain consiste de três componentes principais, **clientes**,**mineradores** e **blockchain**. Os *Clientes* são os que realizam as transações, dessa forma uma analogia ao mercado de bens e serviços são os que compra/vendem produtos ou serviços. Os **Mineradores** são responsáveis por pegar as transações, de um **pool** de transações, e montar em bloco. O minerador precisa fornecer uma *proof-of-work* válida para obter uma recompensa de mineração. E por ultimo o **Blockchain** que é uma estrutura de dados que organiza todos os blocos minerados em ordem cronológica e imutável. Dessa forma será discutido os 3 componentes em mais detalhes no código.

O cliente realiza transações entre carteiras, dessa forma eles devem manter relacionamento entre eles. Para enviar o cliente cria uma transação com o nome do remetente e o valor a ser pago. Para receber ele fornece sua identidade ao terceiro. Nas primeiras linhas do código é apresentado as bibliotecas padrão utilizadas. Visto que precisamos assinar nossas transações, criar hash dos objetos, etc, bibliotecas especificas de criptografia foram utilizadas (PKI), código 1.

```
import hashlib
import binascii
from typing_extensions import Self
import numpy as np
import pandas as pd
import pylab as pl
import datetime
import collections

# required by PKI
import Crypto
import Crypto.Random
from Crypto.Hash import SHA
from Crypto.PublicKey import RSA
from Crypto.Signature import PKCS1_v1_5
```

Código 1. Bibliotecas utilizadas

No codigo 2 a Classe Client gera as chaves privada e pública usando o algoritmo RSA interno do Python, observe que durante a inicialização do objeto ambas, chaves privadas e públicas são geradas e armazenada seus valores na variável de instância. A chave pública gerada será utilizada como identidade do cliente, assim uma propriedade chamada *identity* que retorna a representação *HEX* da chave pública. O código da Classe Client é mostrado em 2.

```
class Client:

    def __init__(self):

        random = Crypto.Random.new().read
        self._private_key = RSA.generate(1024, random)
        self._public_key = self._private_key.publickey()
        self._signer = PKCS1_v1_5.new(self._private_key)



    @property
    def identity(self):
        return binascii.hexlify(self._public_key.exportKey(format='DER'
    )).decode('ascii')
```

Código 2. Classe Cliente

A classe de **transação** serve para o cliente possa realizar as movimentações de entrada e saída de fundos, podendo assim ser remetente ou destinatário. Para receber, algum outro remetente criará uma transação e especificará o endereço público do destinatário. A inicialização da classe é composta de 3 parâmetros – a chave pública do remetente, a chave pública do destinatário e o valor a ser enviado. No *init*,é criado a variável *self.time* para armazenar a hora da transação.

É adicionado o método *to_dict* que coloca todas as quatro variáveis em um objeto de dicionário. Neste projeto, o primeiro bloco no *blockchain* foi chamado de *Genesis*, responsável por armazenar a primeira transação iniciada pelo criador do blockchain. Assim, é verificado se o remetente é o Genesis, em caso afirmativo, é atribuímos um valor de string à variável identity; caso contrário, é atribuído a identidade do remetente à variável de identidade. Por fim é assinado este objeto de dicionário usando a chave privada do remetente. É utilizado o PKI integrado com algoritmo SHA. O metodo de assinatura é o *sign_transaction*. O codigo 3 pode ser visto abaixo.

```
class Transaction:

    def __init__(self, sender, recipient, value):

        self.sender = sender
        self.recipient = recipient
        self.value = value
        self.time = datetime.datetime.now()

    def to_dict(self):
        if self.sender == "Genesis":
            identity = "Genesis"
        else:
            identity = self.sender.identity

        return collections.OrderedDict({'sender': identity,'recipient':
     self.recipient,'value': self.value,'time' : self.time})

    def sign_transaction(self):
        private_key = self.sender._private_key
        signer = PKCS1_v1_5.new(private_key)
```

```
        h = SHA.new(str(self.to_dict()).encode('utf8'))
        return binascii.hexlify(signer.sign(h)).decode('ascii')
```

Código 3. Classe Transação

A função *display_transaction* serve para exibir as transações, sempre que chamada.O objeto é copiado para uma variável temporária e os valores são impressos no console. O código 4 pode ser visto abaixo.

```
def display_transaction(transaction):
    #transaction:
    dict = transaction.to_dict()
    print ("sender: " + dict['sender'])
    print ('-----')
    print ("recipient: " + dict['recipient'])
    print ('-----')
    print ("value: " + str(dict['value']))
    print ('-----')
    print ("time: " + str(dict['time']))
    print ('-----')
```

Código 4. Exibir transação

Para atender varias transações foi criado uma variável global lista, assim as transações após geradas e assinadas são adicionadas a esta lista. Esse trecho de codigo 5 pode ser visto abaixo.

```

transactions = []

jose = Client()
luiz = Client()
carlos = Client()
ana = Client()

t1 = Transaction(jose,luiz.identity,15.0)
t1.sign_transaction()
transactions.append(t1)

t2 = Transaction(jose,carlos.identity,6.0)
t2.sign_transaction()
transactions.append(t2)

t3 = Transaction(luiz,ana.identity,2.0)
t3.sign_transaction()
transactions.append(t3)

t4 = Transaction(carlos,luiz.identity,4.0)
t4.sign_transaction()
transactions.append(t4)

t5 = Transaction(ana,carlos.identity,7.0)
t5.sign_transaction()
transactions.append(t5)

t6 = Transaction(luiz,carlos.identity,3.0)
t6.sign_transaction()
```

```
transactions.append(t6)

t7 = Transaction(carlos,jose.identity,8.0)
t7.sign_transaction()
transactions.append(t7)

t8 = Transaction(carlos,luiz.identity,1.0)
t8.sign_transaction()
transactions.append(t8)

t9 = Transaction(ana,jose.identity,5.0)
t9.sign_transaction()
transactions.append(t9)

t10 = Transaction(ana,luiz.identity,3.0)
t10.sign_transaction()
transactions.append(t10)
```

Código 5. Lista transações

Lembre que os Mineradores pegam transações e montam blocos. O bloco poder variar em numero de transações, neste projeto o bloco consiste em um número fixo de transações, sendo esse três. Foi declarado a variável *verify_transactions* que significa as transações válidas, ou seja, verificadas e adicionadas ao bloco. O bloco contém também o valor de hash tornando a cadeia de blocos imutável. O hash anterior é armazenado na variável *self.previous_block_hash*. Foi declarado também a variável *Nonce* criado pelo minerador durante o processo de mineração. Po último, como cada bloco precisa do valor do hash do bloco anterior, a variável global chamada *last_block_hash* foi criada. O código 6 é dado abaixo.

```
class Block:
    def __init__(self):
        self.verified_transactions = []
        self.previous_block_hash = ""
        self.Nonce = ""
last_block_hash = ""
```

Código 6. Class Block

Para criar o bloco *Genesis* utilizamos o objeto do tipo cliente, Jose, instanciando anteriormente. Uma transação de 500 QuatiCoins foi enviado para o endereço público de Jose. Na sequência uma instância da classe Block chamada block0 foi gerada. As variáveis *previous_block_hash* e *Nonce* foram iniciadas. Em seguida, a transação t0 é adicionada a lista *verify_transactions*.

O bloco está completamente inicializado e pronto para ser adicionado ao nosso blockchain. Então o hash do bloco (block0) é gerado e o valor armazenado na variável global *last_block_hash*. Este valor será usado pelo próximo minerador em seu bloco. O código 7 pode ser visto abaixo.

```
# Miner 0
# jose = Client()
t0 = Transaction("Genesis",jose.identity,500.0)

block0 = Block()
```

```
block0.previous_block_hash = None
block0.Nonce = None

block0.verified_transactions.append(t0)
digest = hash(block0)

last_block_hash = digest
```

Código 7. Genesis

Finalmente chegamos no parte de criação do blockchain que contém a lista de blocos encadeados. Essa lista de blocos, foi definida como *QuatiCoin*. Um método útil chamado *dump_blockchain* também foi criado para visualização de todo conteúdo do blockchain. Uma variável temporária chamada *block_temp* é utilizado para referenciar/iterar sobre cada bloco da lista. Lembre, em cada bloco temos uma lista de três transações (exceto o bloco genesis) em uma variável chamada *verified_transactions*. A função *display_transaction* é utilizada para para exibir os detalhes da transação. O código 8 pode ser visto abaixo.

```
QuatiCoin = []

def dump_blockchain (self):

    print ("Number of blocks in the chain: " + "4")
    for x in range (len(QuatiCoin)):
        block_temp = QuatiCoin[x]
        if x == len(QuatiCoin)-1:
            print ("block # " + str(x))
    for transaction in block_temp.verified_transactions:
        display_transaction (transaction)
        print('--------------')
        print('====================================')
```

Código 8. Criando o blockchain

Para iniciar a mineração, foi desenvolvido a função de mineração, que gera um resumo em uma determinada string de mensagem e fornecer uma proof-to-work.

O resumo é criado pela função *sha256* codifica-a em ASCII, gera um resumo hexadecimal e retorna o valor.

A função de mineração, chamada *mine* implementa a estratégia de mineração, que neste caso seria gerar um hash na mensagem dada com um determinado número de 1's, relacionado como o nível de dificuldade. Por exemplo, nível de dificuldade 2, o hash gerado em uma determinada mensagem deve começar com dois 1's - como 11xxxxxxxx. Se for 3, deve começar com três 1's - como 111xxxxxxxx, etc. O nível de dificuldade precisa ser maior ou igual a 1, assert é utilizado para garantir isso. Se a condição for satisfeita, o loop termina e o valor do resumo é retornado. Um minerador com mais poder de processamento poderá minerar uma determinada mensagem mais cedo. É assim que os mineradores competem entre si. O código 9 pode ser visto abaixo.

```
def sha256(message):
    return hashlib.sha256(message.encode('ascii')).hexdigest()

def mine(message, difficulty=1):
```

```
    assert difficulty >= 1
    prefix = '1' * difficulty

    for i in range(1000):
        digest = sha256(str(hash(message)) + str(i))
        if digest.startswith(prefix):
            return digest
```

Código 9. Função de mineração

Por fim será adicionado blocos ao blockchain. Para conhecer o número de mensagens já extraídas é criada a variável *last_transaction_index*. Um novo bloco é gerado criando uma instância da classe Block. É coletado 3 transações da fila, e antes de adicionar a transação ao bloco, o minerador deve verificar a validade da transação, dita testando igualdade de hash fornecido pelo remetente em relação ao hash gerado pelo minerador usando a chave pública do remetente. Além de verificará se o remetente possui saldo suficiente para a transação. Depois de validada é então adicionada à lista *verified_transactions*. O índice de transação é incrementado para o próximo minerador. É adicionado o hash do último bloco no bloco atual. Em seguida é minerado o bloco com nível de dificuldade 2. Por último é feito o hash de todo o bloco. Por fim, o bloco é adicionado ao blockchain e a variável global, *last_block_hash* e atualizada para uso no próximo bloco. O código 10 é mostrado abaixo.

```
last_transaction_index = 0
# Miner 1
block = Block()

for i in range(3):
    temp_transaction = transactions[last_transaction_index]
    # validate transaction
    # if valid
    block.verified_transactions.append(temp_transaction)
    last_transaction_index += 1

block.previous_block_hash = last_block_hash

block.Nonce = mine(block, 2)
digest = hash(block)
QuatiCoin.append(block)
last_block_hash = digest
dump_blockchain(QuatiCoin)
```

Código 10. Adicionando blocos ao blockchain

Foi adicionado mais 2 blocos ao blockchain, código 11.

```
# Miner 2
block = Block()

for i in range(3):
    temp_transaction = transactions[last_transaction_index]
    # validate transaction
    # if valid
    block.verified_transactions.append(temp_transaction)
    last_transaction_index += 1

```

```
block.previous_block_hash = last_block_hash

block.Nonce = mine(block, 2)
digest = hash(block)
QuatiCoin.append(block)
last_block_hash = digest

dump_blockchain(QuatiCoin)

# Miner 3
block = Block()

for i in range(3):
    temp_transaction = transactions[last_transaction_index]
    # validate transaction
    # if valid
    block.verified_transactions.append(temp_transaction)
    last_transaction_index += 1

block.previous_block_hash = last_block_hash

block.Nonce = mine(block, 2)
digest = hash(block)
QuatiCoin.append(block)
last_block_hash = digest


dump_blockchain(QuatiCoin)
```

Código 11. Adicionando + 2 blocos ao blockchain

A função *dump_blockchain* é utilizada para verificar o resultado, veja abaixo, figura 4.

A Figura 4, apresenta uma visão parcial dos 4 blocos gerados, para visualizar o resultado completo acesse os arquivos disponibilizados em github ou vá para `https://github.com/AdsonNAlves/blockchain`. O código utilizado neste trabalho é blockchainfull.py o resultado visto no terminal está no arquivo resultadoCMD.pdf.

4. Conclusão

Através deste trabalho foi possível compreender de forma ampla vários aspectos do contexto de blockchain, protocolos de criptografia e arquiteturas de aplicação.

O algoritmo apresentado busca mostra de forma detalhada o passo a passo da estrutura do blockchain. É possível concluir que é uma estrutura imutável e de áreas de aplicação bem amplas o que justifica o alto incentivo e procura de profissional nesta área.

Melhorias no algoritmo como funções para gerenciar a fila de transações e uma interface para o cliente poderiam ser bem vindas.

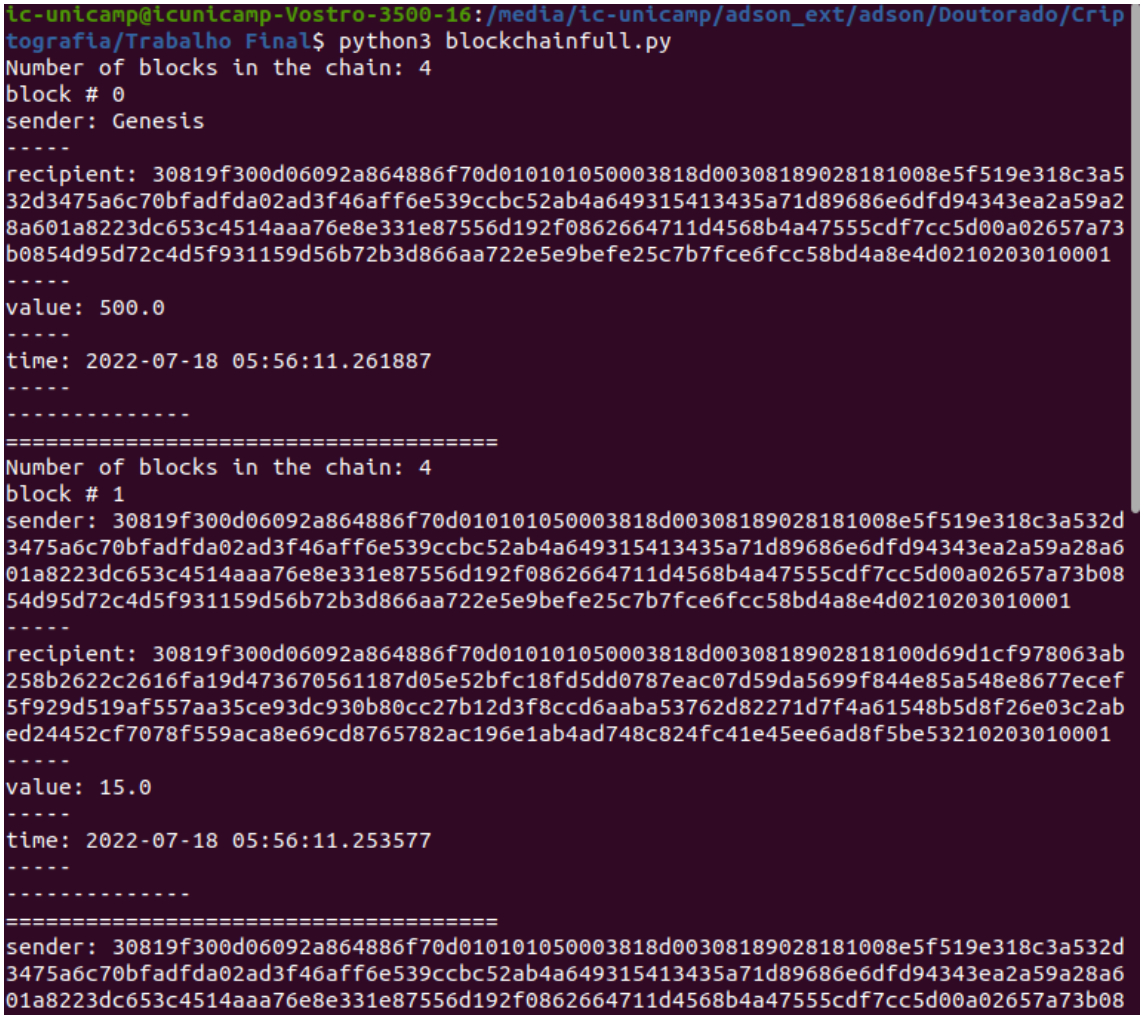

Figure 4. Resultado terminal

References

[blo] Python Blockchain kernel description. `https://www.tutorialspoint.com/python_blockchain/index.htm`. Accessed: 2022-07-18.

[Boneh et al. 2005] Boneh, D., Boyen, X., and Goh, E.-J. (2005). Hierarchical identity based encryption with constant size ciphertext. In Cramer, R., editor, *Advances in Cryptology – EUROCRYPT 2005*, pages 440–456, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Bose et al. 2018] Bose, S., Raikwar, M., Mukhopadhyay, D., Chattopadhyay, A., and Lam, K.-Y. (2018). Blic: A blockchain protocol for manufacturing and supply chain management of ics. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1326–1335.

[Ghimire and Selvaraj 2018] Ghimire, S. and Selvaraj, H. (2018). A survey on bitcoin cryptocurrency and its mining. In *2018 26th International Conference on Systems Engineering (ICSEng)*, pages 1–6.

[Goyal et al. 2006] Goyal, V., Pandey, O., Sahai, A., and Waters, B. (2006). Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, page 89–98, New York, NY, USA. Association for Computing Machinery.

[Kumar et al. 2016] Kumar, S., Rosnes, E., and i Amat, A. G. (2016). Private information retrieval in distributed storage systems using an arbitrary linear code. *CoRR*, abs/1612.07084.

[Mallouli et al. 2019] Mallouli, F., Hellal, A., Sharief Saeed, N., and Abdulraheem Alzahrani, F. (2019). A survey on cryptography: Comparative study between rsa vs ecc algorithms, and rsa vs el-gamal algorithms. In *2019 6th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/ 2019 5th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*, pages 173–176.

[Malluhi et al. 2019] Malluhi, Q., Shikfa, A., Tran, V., and Trinh, V. (2019). Decentralized ciphertext-policy attribute-based encryption schemes for lightweight devices. *Computer Communications*, 145:113–125.

[Raikwar et al. 2019] Raikwar, M., Gligoroski, D., and Kralevska, K. (2019). Sok of used cryptography in blockchain. Cryptology ePrint Archive, Paper 2019/735. `https://eprint.iacr.org/2019/735`.

[Zhao 2018] Zhao, Y. (2018). Aggregation of gamma-signatures and applications to bitcoin. Cryptology ePrint Archive, Paper 2018/414. `https://eprint.iacr.org/2018/414`.