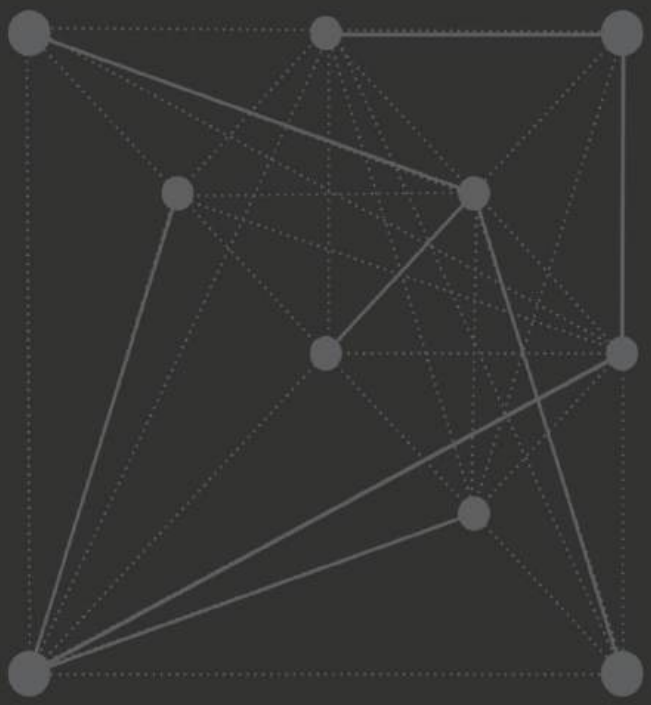
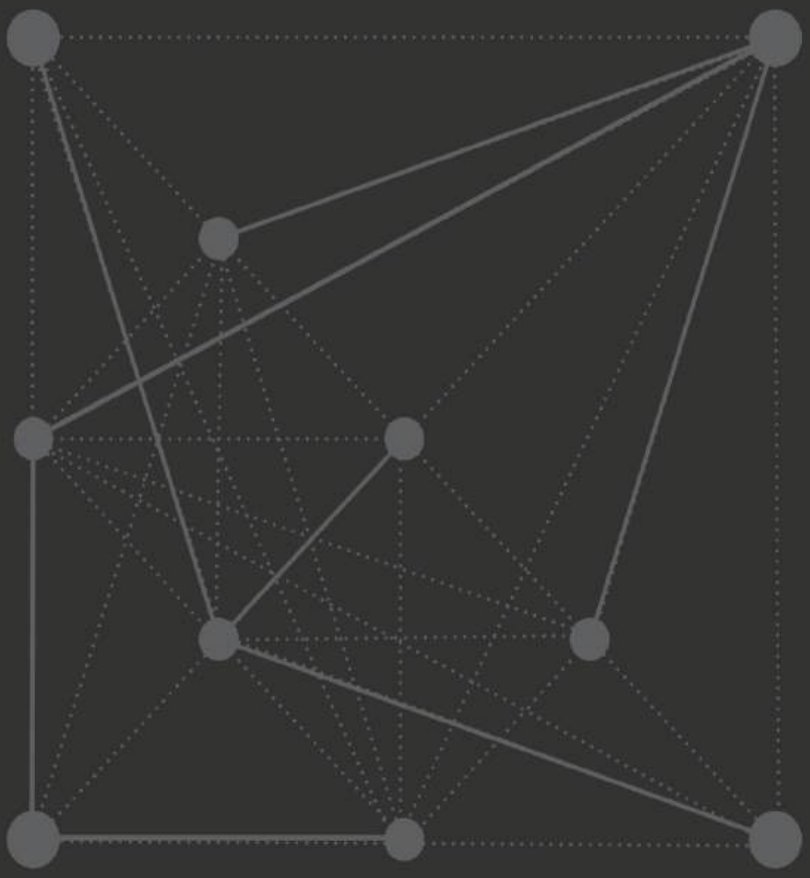
BLOCK
CHAIN
HUB



Bitcoin para Programadores



Marco Agner

ITS
Instituto
de Tecnologia
& Sociedade
do Rio

Tabela de conteúdos

Bitcoin para Programadores

Este livro em formato digital foi escrito por Marco Agner e faz parte de um projeto em conjunto com ITS Rio. O código-fonte deste livro está em: https://github.com/lxparallel/bitcoin-para-programadores.

Sobre o Conteúdo

Tornar-se apto a desenvolver aplicações Bitcoin de forma segura e consciente requer o entendimento de um grupo peculiar de assuntos, incluindo - sem limitar - *Network*, Criptografia, Sistemas Distribuídos, etc. E, tão importante quanto assuntos exatos, estão assuntos humanos para que possamos ter a visão de como sistemas podem criar incentivos para que atores humanos ajam de uma forma ou de outra, ou como uma aplicação pode se tornar mais segura utilizando a linguagem correta.

Este material tem o objetivo de introduzir programadores com interesse nascente nesta tecnologia aos conceitos básicos necessários para o entendimento e desenvolvimento de aplicações Bitcoin. O foco é o mais prático quanto possível sem perder de vista a teoria necessária para uma prática sólida e independente. Logo, por não ser uma especificacão técnica detalhada sobre cada minúncia do protocolo, materiais opcionais para aprofundamento são recomendados ao longo do texto. E, por sua natureza digital, este material permanecerá em evolução em busca de estender e melhorar seu conteúdo.

A linguagem de programação utilizada nos exemplos de implementação nesta primeira versão é Python 3 devido à sua sintaxe de fácil compreensão mesmo para programadores sem experiência na linguagem e a disposição de ótimas bibliotecas para abstração de todo conhecimento que não é especificamente necessário para o conteúdo apresentado, fazendo com que o material atinja o objetivo de capacitar o maior número de pessoas com os mesmos recursos.

Tudo o que você precisa para tirar máximo proveito do material apresentado é ter alguma experiência na arte da programação de computadores, exposição ao paradigma de orientação a objetos e interesse por esta tecnologia revolucionária. No entanto, em razão do caráter introdutório do conteúdo, muitas pessoas sem experiência em programação poderão tirar bastante proveito do material.

Que este possa ser apenas o início de sua jornada numa tecnologia que realmente importa.

Sumário

Introdução

O que é Bitcoin?

De Forma Resumida

Em tradução livre do repositório da principal implementação atualmente (Bitcoin Core):

> *"Bitcoin é uma nova moeda digital experimental que permite pagamento instantâneo para qualquer pessoa, em qualquer lugar do mundo. Bitcoin usa tecnologia peer-to-peer (P2P) para operar sem autoridade central: a gerência de transações e a emissão de dinheiro é executada coletivamente pela rede. Bitcoin Core é o nome do software open source que habilita o uso desta moeda."*

Indo um Pouco Além...

Bitcoin é a união de tecnologias e abstracões que possibilitam que o consenso entre atores não necessariamente conhecidos possa ser alcançado de forma descentralizada sem que a confiaça tenha que ser depositada em um ponto de controle central ou que a segurança rede esteja sujeita à um único ponto de falha. Estas tecnologias em conjunto formam as bases para a existência de uma moeda digital descentralizada e para qualquer outro caso de uso que possamos abstrair para um modelo baseado em consenso - como contratos - de forma independente de autoridades centrais como bancos ou governos.

E, é importante reparar que o mesmo termo "**B**itcoin" com "B" maiúsculo é comumente utilizado para designar a tecnologia como um todo, a rede P2P Bitcoin ou o protocolo Bitcoin enquanto **b**itcoin(s) com "b" minúsculo é utilizado para designar a unidade de conta usada na rede.

A Criptografia no Bitcoin

Bitcoin é uma cripto-moeda; e isto se deve ao fato de a Criptografia ser uma parte essencial em seu funcionamento.

A Criptografia é um ramo da matemática que, em sua definição moderna, acolhe toda a tecnologia criada e utilizada para restringir verdades fundamentais da natureza da informação com o intuito de alcançar objetivos como: esconder mensagens, provar a

existência de um segredo sem a necessidade de revelar o segredo, provar autenticidade e integridade de dados, provar trabalho computacional etc...

A princípio, no Bitcoin, estamos interessados em atingir os seguintes objetivos com uso de algoritmos criptográficos: Garantia de integridade e consistência de dados na rede e prova de trabalho computacional utilizando *hashes* e autenticidade das transações utilizando assinaturas digitais de Criptografia de Chave Pública.

Funções *Hash* Criptográficas

Este tipo de função é usado como bloco fundamental em muitas aplicações criptográficas e tem como comportamento básico receber um conjunto de dados de tamanho arbitrário como *input* e produzir um valor *hash* de um tamanho fixo como *output* que chamamos de *digest* como forma de representação do dado de entrada. Chamamos de funções *hash* **criptográficas**, todas as funções *hash* que atendem aos seguintes requisitos: ela deve resistir a todo tipo de ataque cripto-analítico conhecido, deve ter resistência à colisão - ou seja, a geração de um mesmo *digest* para a *inputs* diferentes deve ser impraticável -, ser computacionalmente eficiente, agir como uma função matemática trap-door - o que significa dizer que deve ser impraticável/improvável reverter fazer o caminho contrário e reverter o *digest* de volta ao *input* original, além de ser impraticável a retirada de qualquer dado útil do *digest* que possa dizer algo sobre o *input* usado na função.

As duas funções *hash* utilizadas no Bitcoin são a SHA-256 (*Secure Hash Algorithm*) que retorna *digests* de 256 *bits* e a RIPEMD-160 (*RACE Integrity Primitives Evaluation Message Digest*) que retorna *digests* de 160 *bits*.

Exemplo de ambas funções sendo usadas em Python com a *string* "bitcoin" como *input* e *print* do *digest* no formato mais comum em hexadecimal:

```
>>> import hashlib

>>> word = "bitcoin"

>>> word_sha256 = hashlib.sha256(word.encode('utf-8'))
>>> print(word_sha256.hexdigest())
6b88c087247aa2f07ee1c5956b8e1a9f4c7f892a70e324f1bb3d161e05ca107b

>>> word_ripemd160 = hashlib.new('ripemd160')
>>> word_ripemd160.update(word.encode('utf-8'))
>>> print(word_ripemd160.hexdigest())
5891bf40b0b0e8e19f524bdc2e842d012264624b

# hashes completamente diferentes são formados com pequenas alterações no input
>>> word2 = "bitcoin2"

>>> word2_sha256 = hashlib.sha256(word2.encode('utf-8'))
>>> print(word2_sha256.hexdigest())
1ed7259a5243a1e9e33e45d8d2510bc0470032df964956e18b9f56fa65c96e89

>>> word2_ripemd160 = hashlib.new('ripemd160')
>>> word2_ripemd160.update(word2.encode('utf-8'))
>>> print(word2_ripemd160.hexdigest())
9d2028ac5216d10b85d1a3ab389ebcc57a3ee6eb
```

Com estas funções em mão conseguimos verificar integridade de informações enviadas à rede, gerar e verificar prova de trabalho computacional ou *proof-of-work* e, com isso, criar a "cola" criptográfica fundamental para a segurança da blockchain (mais detalhes em Mineração).

A compreensão sobre *hashes* neste nível de abstração já é suficiente para o entendimento do valor de suas propriedades no Bitcoin e o uso consciente destas propriedades que você verá adiante em mais exemplos.

Criptografia de Chave Pública

Outra tecnologia fundamental para o funcionamento do Bitcoin é a Criptografia de Chave Pública. Esta tecnologia possibilita que pessoas possam usar ferramentas criptográficas para encriptar e provar/verificar autenticidade de informações trocadas sem a necessidade do compartilhamento de um segredo.

No Bitcoin, estamos mais interessados nas assinaturas digitais produzidas por este tipo de criptografia, na qual eu consigo provar que tenho um segredo (a Chave Privada ou *Private Key*) ao mesmo tempo em que autentico uma transação com este segredo sem a

necessidade de compartilhar este segredo; tendo apenas que compartilhar a minha Chave Pública (*Public Key*). Assim, garantindo que apenas o detentor da Chave Privada correta poderá movimentar fundos na rede.

Eu acho que a melhor explicação simplificada ao estilo *Explain Like I'm 5* que já achei e vai te ajudar a ver o problema com mais simplicidade é um trecho do ótimo artigo sobre Assinaturas Digitais de Curva Elíptica (o método utilizado para as assinaturas no Bitcoin) do falecido (e, por enquanto, "ressuscitado") blog The Royal Fork que, em tradução livre e um pouco alterada para adequação, descreve:

> *"Imagine uma turma de crianças nos primeiros anos de escola que sabem multiplicação, mas ainda não aprenderam divisão. No início do ano, o professor proclama 'Meu número especial é 3'. Numa manhã, a mensagem 'Sempre foi assim e sempre irá ser' - assinado 'Professor - 11' aparece no quadro negro. Como os alunos sabem que esta mensagem veio do professor e não de um fraudador que gosta de recitar frases de filmes? Eles multiplicam o "número especial" do professor - 3 - pelo "número da assinatura" - 11 - e se eles obtiverem o número de caracteres contidos na mensagem (33 caracteres), eles julgam a assinatura válida, e estão confiantes de que a mensagem realmente foi escrita pelo professor. Sem a mágica da divisão, os alunos não conseguem produzir uma assinatura válida para qualquer mensagem arbitrária, e porque a assinatura é baseada no tamanho da mensagem, os estudantes não podem mudar a mensagem sem invalidar a assinatura.*
>
> *Isto é fundamentalmente como o Algoritmo de Assinaturas Digitais de Curva Elíptica funciona; Ao conhecedor da chave privada é concedido o poder da divisão, enquanto os conhecedores da chave pública estão restritos à multiplicação, o que os permite checarem se uma assinatura é válida ou não."*

Há mais de um método matemático para alcançar este tipo de funcionalidade e no Bitcoin é utilizado o Algoritmo de Assinaturas Digitais de Curva Elípica (ou *Elliptic Curve Digital Signature Algorithm*) ao qual vou me referenciar a partir de agora pela sigla em inglês ECDSA. A implementação utilizada no Bitcoin Core - a implementação referência atual - é a libsecp256k1 desenvolvida pelo Bitcoin Core *developer* Pieter Wuille para substituir a implementação anterior que utilizava a biblioteca OpenSSL com o objetivo de remover uma dependência do código do Core, além de ser uma implementação muito mais eficiente do algoritmo com um código melhor testado do que na OpenSSL.

Este tipo de assinatura é o que permite que todos na rede possam comprovar que uma transação foi enviada pelo detentor de uma certa chave privada - que nada mais é que um número gigante de 256 *bits* obtido, se feito corretamente, de forma criptograficamente aleatória - sendo, assim, essencial para o funcionamento correto do Bitcoin.

Por exemplo, digamos que Maria envia 1 bitcoin para o endereço bitcoin de João e João, por sua vez, envia 1 bitcoin para o endereço de Raphael. Todas estas transações são apenas mensagens que dizem "passar *n* bitcoins de *x* para *y*" e todas estas mensagens precisam ser assinadas por quem a rede considera o atual detentor dos bitcoins para que sejam consideradas válidas e incluidas na blockchain por algum minerador.

Em Endereços e Carteiras, veremos como a chave privada pode ser gerada e como fazemos para derivar a chave pública a partir da chave privada para, finalmente, gerar um endereço Bitcoin. Em Transações veremos como as assinaturas são realmente enxergadas na rede e como assinar transações utilizando os comandos RPC do Bitcoin Core.

Próximo capítulo: Bitcoin Core

Bitcoin Core

> "O Bitcoin Core é um projeto *open-source* que mantém e publica o *software* cliente chamado 'Bitcoin Core'.
>
> É descendente direto do *software* cliente original Bitcoin publicado por Satoshi Nakamoto após ele ter publicado o famoso *whitepaper* do Bitcoin.
>
> O Bitcoin Core consiste de um software *'full-node'* para completa validação da blockchain, assim como uma carteira bitcoin. O projeto, atualmente, também mantém *softwares* relacionados como a biblioteca de criptografia libsecp256k1 e outros que podem ser encontrados no GitHub.
>
> Qualquer um pode contribuir para o Bitcoin Core."

... e, por razões óbvias, esta implementação em C++ é a escolhida pelo autor neste material.

Instalação

Você pode instalar o Bitcoin Core utilizando um dos binários disponíveis aqui ou compilar a partir do código-fonte. Recomendo que você compile diretamente do código-fonte para que tenha maior autonomia para escolher opções de instalação personalizadas e possa auditar o código que rodará em sua máquina. Porém, você cosneguirá acompanhar o material normalmente se decidir por apenas instalar um binário para o seu sistema se preferir.

Para baixar o código a ser compilado recomendo que utilize Git para lidar com o repositório no GitHub, mas você também pode baixar a última *release* do código-fonte aqui e seguir os mesmos passos da compilação.

Eu seguirei com a instalação geral para sistemas Unix e alguns passos podem ser diferentes para diferentes sistemas. Logo, o repositório também disponibiliza instruções específicas para OpenBSD, OS X e Windows.

Dependências

Antes de compilar o código do Bitcoin Core em si, você terá que cuidar da instalação de algumas dependências. As 3 primeiras listadas são explicitamente necessárias como listado abaixo e, das opcionais, recomendo que você instale todas as necessárias para utilizar a *interface* gráfica e as funcionalidades de carteira do Bitcoin Core.

Cada uma destas dependências podem ser encontradas no *package manager* (APT, yum, dnf, brew...) que você utiliza e os nomes podem ser um pouco diferentes de entre cada um deles. Já para a instalação que eu recomendo do Berkeley DB, talvez você precise baixar e compilar diretamente http://download.oracle.com/berkeley-db/db-4.8.30.NC.tar.gz.

Para resolver o Berkeley DB basta seguir estes comandos:

```
$ wget http://download.oracle.com/berkeley-db/db-4.8.30.NC.tar.gz
$ sha256sum db-4.8.30.NC.tar.gz
# o último comando deve ter gerado o *hash* 12edc0df75bf9abd7f82f821795bcee50f42cb2e5f
76a6a281b85732798364ef
$ tar -xvf db-4.8.30.NC.tar.gz
$ cd db-4.8.30.NC/build_unix
$ mkdir -p build
$ BDB_PREFIX=$(pwd)/build
$ ../dist/configure –disable-shared –enable-cxx –with-pic –prefix=$BDB_PREFIX
$ make install
```

Com isso, você poderá usar as funcionalidades de *wallet*.

Continue na mesma janela para manter a variável $BDB_PREFIX ou guarde este valor para usar na configuração do Bitcoin Core abaixo.

Aqui estão as dependências **necessárias** para a compilação:

Biblioteca	Propósito	Descrição
libssl	Criptografia	Geração Aleatória de Números, Criptografia de Curva Elíptica
libboost	Utilidade	Biblioteca para *threading*, estruturas de dados, etc
libevent	Rede	Operações de rede assíncronas independente de Sistema Operacional

E estas são as dependências **opcionais** são:

Biblioteca	Propósito	Dscrição
miniupnpc	Suporte UPnP	Suporte para *jumping* de Firewall
libdb4.8	Berkeley DB	Armazenamento de carteira (apenas necessário com *wallet* habilitada)
qt	GUI	Conjunto de ferramentas para *GUI* (apenas necessário com *GUI* habilitada)
protobuf	Pagamento na GUI	Intercâmbio de dados usado para o protocolo de pagamento (apenas necessário com *GUI* habilitada)
libqrencode	Códigos QR na GUI	Opcional para geração de códigos QR (apenas necessário com *GUI* habilitada)
univalue	Utilidade	*Parsing* e codificação de JSON (versão empacotada será usada a não ser que --with-system-univalue seja passada na *configure*)
libzmq3	Notificação ZMQ	Opcional, permite geração de notificações ZMQ (requer servidor ZMQ versão >= 4.x)

Compilando

Agora, com as dependências instaladas, você pode baixar o código-fonte utilizando Git para clonar o repositório ou direto da página de *releases*. Como já dito, recomendo a primeira opção.

Vale lembrar que você, provavelmente, terá um ou outro problema específico ao seu sistema com a instalação das dependências e que você irá descobrir neste próximo passo de preparação e compilação do código. No entanto, os *outputs* com os erros causados por falta de dependências corretas são bastante prestativos e as respostas para estes problemas são facilmente encontradas pela internet.

Seguindo com o Git, clone o repositório:

```
$ git clone https://github.com/bitcoin/bitcoin.git
$ cd bitcoin
```

Liste as *tags* de *releases* disponíveis:

```
$ git tag
#[... outras tags ...]
v0.11.2
v0.11.2rc1
v0.12.0
v0.12.0rc1
v0.12.0rc2
v0.12.0rc3
v0.12.0rc4
v0.12.0rc5
v0.12.1
#[... outras tags ...]
```

E dê um *checkout* para a última versão estável ou, se preferir, a última *release candidate*:

```
$ git checkout v0.12.1
Note: checking out 'v0.12.1'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 9779e1e... Merge #7852: [0.12] Add missing reference to release notes
```

Agora basta fazer o *build* (obs.: $BDB_PREFIX definida acima):

```
$ ./autogen.sh
$ ./configure LDFLAGS="-L${BDB_PREFIX}/lib/" CPPFLAGS="-I${BDB_PREFIX}/include/" --wit
h-gui
#[...]
$ make
#[...]
$ make install  # opcional caso queira os binários no seu $PATH
#[...]
```

Se tudo ocorreu bem, você já pode utilizar `bitcoind` e `bitcoin-cli`:

```
$ bitcoind --version
Bitcoin Core Daemon version v0.12.1
Copyright (C) 2009-2016 The Bitcoin Core Developers

This is experimental software.

Distributed under the MIT software license, see the accompanying file COPYING
or <http://www.opensource.org/licenses/mit-license.php>.

This product includes software developed by the OpenSSL Project for use in the
OpenSSL Toolkit <https://www.openssl.org/> and cryptographic software written
by Eric Young and UPnP software written by Thomas Bernard.
```

```
$ bitcoin-cli --version
Bitcoin Core RPC client version v0.12.1
```

Basicamente, o bitcoind é o cliente Bitcoin que expõe uma API JSON-RPC quando em modo *server* e o bitcoin-cli é a ferramenta que utilizamos para nos comunicarmos com esta API pela linha de comando.

API JSON-RPC

Preparação

Para começar, chame o bitcoind e dê um CTRL+C logo em seguida apenas para que ele forme a estrutura da pasta .bitcoin automaticamente:

```
$ bitcoind
^C
```

Em seguida entre na pasta .bitcoin que, por padrão, fica no seu diretório $HOME e crie ou edite um arquivo chamado bitcoin.conf com um usuário para o servidor RPC e uma senha **forte** diferente da mostrada no exemplo. O arquivo ~/.bitcoin/bitcoin.conf deve ficar como este:

```
rpcuser=bitcoinrpc
rpcpassword=cF58sc+MuY5TM4Dhjs46U2MylXS/krSxB+YW1Ghidzg
```

*Caso não esteja usando Linux, o seu diretório padrão será diferente:

Para Windows - `%APPDATA%\Bitcoin`

Para OSX - `~/Library/Applicatio Support/Bitcoin/`

Com isso feito, você já tem tudo pronto para rodar o *software* cliente como um *full-node* na rede escolhida. O bitcoind, por padrão, conecta-se à *mainnet* que é a rede Bitcoin "de produção" com real valor monetário e oferece as flags `-testnet` e `-regtest` para caso escolha se conectar na testnet3 ou usar regtest. Para a maior parte deste material, recomendo que siga rodando a mainnet ou testnet3; Aqui vão algumas informações sobre cada uma das três opções:

---

mainnet: A rede "de produção" do Bitcoin. Aqui é onde as transações movem valor monetário real e qualquer erro cometido, especialmente comum quando desenvolvendo, pode significar uma grande perda monetária para você. O *market cap* atual da rede é de cerca de 7 bilhões de dólares com um *hash rate* aproximado de 1 bilhão de *GigaHashes por segundo* (sim, isso é MUITO poder de processamento!). No momento, é recomendado que você tenha, pelo menos, 80GB de espaço de armazenamento livre para acomodar a blockchain de quase 70GB e deixar algum espaço para acompanhar o crescimento dela caso não use a opção `prune` (mais sobre esta opção logo abaixo). Eu sincronizo com a mainnet para trabalho utilizando um *drive* USB externo com a opção `prune` em 10GB economizando muito espaço; Caso você queira fazer algo parecido basta iniciar o bitcoind com a opção `-datadir=<diretório que deseja popular> -prune=10000`.

testnet3: A rede de teste do Bitcoin. Esta rede tem uma blockchain separada da *mainnet* e é utilizada para testar novas implementações do protocolo (como, atualmente, *segwit* que foi ativada em dezembro de 2015 na *testnet3*), desenvolver sem arriscar valores monetários significativos ou a possibilidade de causar problemas não intencionais na *mainnet*. Esta é a terceira geração da *testnet*; A *testnet2* foi criada apenas pra reiniciar a rede com um novo bloco *genesis* já que algumas pessoas começaram a atribuir valor financeiro à *tesnet* e negociar seus *bitcoins* por dinheiro - o que não é a intenção da rede - e a *testnet3* além de introduzir um novo bloco *genesis* também trouxe algumas melhorias e facilidades para o desenvolvimento, como o ajuste diferente de dificuldade da rede. Os *bitcoins* da *testnet* sao comumente chamados de *test/tesnet coins* e os endereços nesta rede começam com "m" para *hashes* de chave pública e "2" para *script hashes* (mais detalhes sobre ambos tipos de endereço em enderecos). Algo em torno de 10GB~15GB deve bastar para você fazer download da blockchain e trabalhar sem se preocupar com armazenamento, além de poder usar a opção `prune` com um valor menor para armazenamento. Para isso basta que inicie o bitcoind com a opção `-testnet` e, se quiser, pode utilizar a flag `-datadir` normalmente.

regtest: A rede de "teste de regressão" do Bitcoin. A dificuldade de mineração da *regtest* é praticamente 0 e, normalmente, você a inicia do 0 com nenhum bloco minerado e nenhum *peer* conectado. É como uma realidade paralela do Bitcoin em que você pode utilizar para

fazer testes bastante interessantes e analisar o comportamento da rede em situações adversas completamente controladas por você. Você pode iniciar vários nós diferentes conectados à sua *regtest*, minerar, enviar transações entre os nós e testar praticamente qualquer situação necessária. Esta rede é especialmente útil para testar coisas que você não tem controle nas outras redes devido à participação de outros nós ou para fazer testes que não necessite da imprevisibilidade das redes públicas usando pouco espaço de armazenamento. A opção para iniciar o bitcoind na *regtest* é passada pela flag `-regtest`.

*Nota sobre a opção `prune`: A partir do Bitcoin Core v0.11.0, você pode utilizar a opção `prune` - "poda"/"podar" em português - sem possibilidade de utilizar a funcionalidade de carteira do Core, e a partir do Bitcoin Core v0.12.0 já é possível utilizar a funcionalidade de carteira do *software* junto com a opção `prune`. Esta opção permite que você trabalhe guardando uma porção menor da blockchain, ficando apenas com os últimos N blocos com N sendo o número de MB que você decidir para esta opção. É importante notar que esta opção lida apenas com o tamanho da blockchain (na pasta blocks) e não conta o espaço necessário para o armazenamento dos outros arquivos - o que não é muito. Para usar esta opção para limitar os blocos para algo em torno de 10GB, você pode iniciar o `bitcoind` com a flag `-prune=10000` ou colocar uma linha em seu bitcoin.conf``` assim:

```
rpcuser=bitcoinrpc
rpcpassword=cF58sc+MuY5TM4Dhjs46U2MylXS/krSxB+YW1Ghidzg
prune=10000  # <--- limita o armazenamento de blocos
```

Interagindo com a API JSON-RPC

Tendo compreendido a diferença entre as redes, já podemos inciar o *bitcoind* com as opções escolhidas. Recomendo que dê uma lida nas opções disponíveis com o comando `bitcoind --help`. Para iniciar o *bitcoind* conectado à *mainnet* em modo *daemon* para não segurar o terminal, basta:

```
$ bitcoind -daemon
Bitcoin server starting
```

E pronto... O seu nó começará a sincronizar com a rede e você já poderá se comunicar pela linha de comando via a API JSON-RPC (lembrando que o comando `bitcoin-cli` precisa da flag `-testnet` ou `-regtest` caso tenha iniciado o `bitcoind` com alguma destas opções). É uma ótima ideia ler cada comando disponível para se familiarizar:

```
$ bitcoin-cli help
== Blockchain ==
```

```
getbestblockhash
getblock "hash" ( verbose )
getblockchaininfo
getblockcount
getblockhash index
getblockheader "hash" ( verbose )
getchaintips
getdifficulty
getmempoolinfo
getrawmempool ( verbose )
gettxout "txid" n ( includemempool )
gettxoutproof ["txid",...] ( blockhash )
gettxoutsetinfo
verifychain ( checklevel numblocks )
verifytxoutproof "proof"

== Control ==
getinfo
help ( "command" )
stop

== Generating ==
generate numblocks
getgenerate
setgenerate generate ( genproclimit )

== Mining ==
getblocktemplate ( "jsonrequestobject" )
getmininginfo
getnetworkhashps ( blocks height )
prioritisetransaction <txid> <priority delta> <fee delta>
submitblock "hexdata" ( "jsonparametersobject" )

== Network ==
addnode "node" "add|remove|onetry"
clearbanned
disconnectnode "node"
getaddednodeinfo dns ( "node" )
getconnectioncount
getnettotals
getnetworkinfo
getpeerinfo
listbanned
ping
setban "ip(/netmask)" "add|remove" (bantime) (absolute)

== Rawtransactions ==
createrawtransaction [{"txid":"id","vout":n},...] {"address":amount,"data":"hex",...}
( locktime )
decoderawtransaction "hexstring"
decodescript "hex"
fundrawtransaction "hexstring" includeWatching
getrawtransaction "txid" ( verbose )
```

```
sendrawtransaction "hexstring" ( allowhighfees )
signrawtransaction "hexstring" ( [{"txid":"id","vout":n,"scriptPubKey":"hex","redeemSc
ript":"hex"},...] ["privatekey1",...] sighashtype )

== Util ==
createmultisig nrequired ["key",...]
estimatefee nblocks
estimatepriority nblocks
estimatesmartfee nblocks
estimatesmartpriority nblocks
validateaddress "bitcoinaddress"
verifymessage "bitcoinaddress" "signature" "message"

== Wallet ==
abandontransaction "txid"
addmultisigaddress nrequired ["key",...] ( "account" )
backupwallet "destination"
dumpprivkey "bitcoinaddress"
dumpwallet "filename"
encryptwallet "passphrase"
getaccount "bitcoinaddress"
getaccountaddress "account"
getaddressesbyaccount "account"
getbalance ( "account" minconf includeWatchonly )
getnewaddress ( "account" )
getrawchangeaddress
getreceivedbyaccount "account" ( minconf )
getreceivedbyaddress "bitcoinaddress" ( minconf )
gettransaction "txid" ( includeWatchonly )
getunconfirmedbalance
getwalletinfo
importaddress "address" ( "label" rescan p2sh )
importprivkey "bitcoinprivkey" ( "label" rescan )
importpubkey "pubkey" ( "label" rescan )
importwallet "filename"
keypoolrefill ( newsize )
listaccounts ( minconf includeWatchonly)
listaddressgroupings
listlockunspent
listreceivedbyaccount ( minconf includeempty includeWatchonly)
listreceivedbyaddress ( minconf includeempty includeWatchonly)
listsinceblock ( "blockhash" target-confirmations includeWatchonly)
listtransactions ( "account" count from includeWatchonly)
listunspent ( minconf maxconf  ["address",...] )
lockunspent unlock [{"txid":"txid","vout":n},...]
move "fromaccount" "toaccount" amount ( minconf "comment" )
sendfrom "fromaccount" "tobitcoinaddress" amount ( minconf "comment" "comment-to" )
sendmany "fromaccount" {"address":amount,...} ( minconf "comment" ["address",...] )
sendtoaddress "bitcoinaddress" amount ( "comment" "comment-to" subtractfeefromamount )
setaccount "bitcoinaddress" "account"
settxfee amount
signmessage "bitcoinaddress" "message"
```

Informações de Status

O comando RPC `getinfo` retorna informações básicas sobre o seu nó na rede, sua carteira e a blockchain:

```
$ bitcoin-cli getinfo
{
  "version": 120100,
  "protocolversion": 70012,
  "walletversion": 60000,
  "balance": 0.00000000,
  "blocks": 259896,
  "timeoffset": 1,
  "connections": 8,
  "proxy": "",
  "difficulty": 112628548.6663471,
  "testnet": false,
  "keypoololdest": 1461593328,
  "keypoolsize": 101,
  "paytxfee": 0.00000000,
  "relayfee": 0.00001000,
  "errors": ""
}
```

E o `getpeerinfo` retorna informações sobre os nós conectados a você na rede:

```
$ bitcoin-cli getpeerinfo
[
  {
    "id": 13,
    "addr": "91.159.81.78:8333",
    "addrlocal": "209.222.18.59:44718",
    "services": "0000000000000005",
    "relaytxes": true,
    "lastsend": 1461603299,
    "lastrecv": 1461603300,
    "bytessent": 1688470,
    "bytesrecv": 1317410461,
    "conntime": 1461596517,
    "timeoffset": -4,
    "pingtime": 5.683412,
    "minping": 0.256795,
    "version": 70012,
    "subver": "/Satoshi:0.12.0/",
    "inbound": false,
    "startingheight": 408868,
    "banscore": 0,
    "synced_headers": 408876,
    "synced_blocks": 260036,
    "inflight": [
      260929,
      260937,
      260940,
      260943,
      260944,
      260946,
      260951,
      260961,
      260972,
      260974,
      260978,
      260992,
      261010,
      261017,
      261022,
      261033
    ],
    "whitelisted": false
  },
#[... outros peers ...]
]
```

Configurando a Carteira

A primeira coisa que você deve fazer com a sua carteira é encriptar ela com uma senha forte. Para encriptar com a senha "naouseestasenhaidiota", entre:

```
$ bitcoin-cli encryptwallet naouseestasenhaidiota
wallet encrypted; Bitcoin server stopping, restart to run with encrypted wallet. The k
eypool has been flushed, you need to make a new backup.
```

Inicie o `bitcoind` novamente e verifique que sua carteira está encriptada com `getinfo` ao observar o novo atributo "unlocked_until" que deverá ter o valor 0:

```
$ bitcoin-cli getinfo
{
  "version": 120100,
  "protocolversion": 70012,
  "walletversion": 60000,
  "balance": 0.00000000,
  "blocks": 266096,
  "timeoffset": 1,
  "connections": 8,
  "proxy": "",
  "difficulty": 267731249.4824211,
  "testnet": false,
  "keypoololdest": 1461603754,
  "keypoolsize": 100,
  "unlocked_until": 0,  # <--- a carteira está encriptada
  "paytxfee": 0.00000000,
  "relayfee": 0.00001000,
  "errors": ""
}
```

Para "destrancar" a carteira, use o comando `walletpassphrase` com a senha e o tempo que deseja que a carteira permaneça "aberta" antes de se trancar novamente:

```
$ bitcoin-cli walletpassphrase naouseestasenhaidiota 120
```

E quando chamar `getinfo` novamente, poderá verificar que a carteira está "destrancada":

```
$ bitcoin-cli getinfo
{
  "version": 120100,
  "protocolversion": 70012,
  "walletversion": 60000,
  "balance": 0.00000000,
  "blocks": 269955,
  "timeoffset": 1,
  "connections": 8,
  "proxy": "",
  "difficulty": 510929738.0161518,
  "testnet": false,
  "keypoololdest": 1461603754,
  "keypoolsize": 101,
  "unlocked_until": 1461604440, #  <--- tempo em formato UNIX até quando a carteira pe
rmanecerá encriptada
  "paytxfee": 0.00000000,
  "relayfee": 0.00001000,
  "errors": ""
}
```

Criando e Restaurando Backups da Carteira

Agora, vamos criar ver como criar um arquivo de *backup* e restaurar a sua carteira a partir deste arquivo. Para criar o backup, entre o comando `backupwallet` com o destino do arquivo de *backup*:

```
$ bitcoin-cli backupwallet /home/user/carteira-2016-04-25.backup
```

E para restaurar a partir deste backup, simplemente use `importwallet` com a localização do arquivo como parâmetro - a carteira deve estar aberta para isso, caso já esteja com outra:

```
$ bitcoin-cli importwallet carteira-2016-04-25.backup
```

Caso precise da carteira em texto num formato legível, use `dumpwallet` com o arquivo destino do texto como parâmetro:

```
$ bitcoin-cli dumpwallet wallet.txt
```

E você terá um arquivo .txt com o *dump* da carteira inteira.

Transações

O seu *software* cliente gera endereços automaticamente e os mantém em uma *pool* de endereços. O atributo *keypoolsize* que indica quantos endereços públicos já gerados a sua carteira tem pode ser obtido com *getinfo*. Para pegar um desses endereços, você simplemente usa o comando `getnewaddress`:

```
$ bitcoin-cli getnewaddress
1GswWkrWvxGhCARW4w2ibnwk3pDURM6GpC
```

Envie alguns *millibits* para o endereço que você acabou de pegar na linha de comando para você poder ver como checar a quantia recebida por endereço. No meu caso, vou enviar 10 *millibits* (0.010 bitcoins) para o endereço.

Após enviar, você pode checar o valor recebido com o comando `getreceivedbyaddress`, porém, por padrão, ele precisa de alguns números de confirmação antes de mostrar a quantia - você pode configurar isso no arquivo bitcoin.conf; para que mostre a quantia recebida mesmo sem confirmação, chamaremos este comando com o parâmetro de confirmações mínimas como 0;

```
$ bitcoin-cli getreceivedbyaddress 1GswWkrWvxGhCARW4w2ibnwk3pDURM6GpC 0
0.01000000
```

Se você tentar este mesmo comando sem o parâmetro 0, ele usará o número configurado como `minconf` no seu bitcoin.conf.

Você também pode listar todas as transações recebidas pela carteira em todos os endereços com o comando `listtransactions`:

```
$ bitcoin-cli listtransactions
[
  {
    "account": "",
    "address": "1GswWkrWvxGhCARW4w2ibnwk3pDURM6GpC",
    "category": "receive",
    "amount": 0.01000000,
    "label": "",
    "vout": 0,
    "confirmations": 0,
    "trusted": false,
    "txid": "01ecaecc96b148589be10ef3f8fffc70dcc14970ed77144a787c087dfcd0b5e2",
    "walletconflicts": [
    ],
    "time": 1462453065,
    "timereceived": 1462453065,
    "bip125-replaceable": "no"
  }
]
```

Também podemos listar todos os endereços de uma conta com o comando `getaddressesbyaccount` com o identificador da conta como parâmetro - a padrão inicial é uma *string* vazia.:

```
$ bitcoin-cli getaddressesbyaccount ""
[
  "115PHNcNbBtaWeb1iYfX96hYiehgnxamL2",
  "1GswWkrWvxGhCARW4w2ibnwk3pDURM6GpC",
  "1HFJpe3knukVwfqj22m9mVvLz9R4Ac76f9",
  "1JEqZijxDfTyXHYjiaJTy6SgSCJNdmTayr",
  "1KCoEFDXmtde2LEshZfbDZFwRwrX8pg8H9",
  "1PU7nWdqJsev4sWmgcDJwZG9EZRuS7nLGU"
]
```

E para ver a quantia total de todos endereços da carteira com o número de confirmações mínimos da configuração `minconf`, basta usar o `getbalance`:

```
$ bitcoin-cli getbalance
0.01000000
```

Vamos, agora, ver como a transação é vista pelo *client* usando o comando `gettransaction` com o txid (id da transação; o *hash* da transação) que pegamos com o comando `listtransactions` como parâmetro:

```
$ bitcoin-cli gettransaction 01ecaecc96b148589be10ef3f8fffc70dcc14970ed77144a787c087df
cd0b5e2
{
  "amount": 0.01000000,
  "confirmations": 1,
  "blockhash": "000000000000000042e065b770fb46aba51bd389d086a1a2a47c5d9e5ffe575",
  "blockindex": 2408,
  "blocktime": 1462453539,
  "txid": "01ecaecc96b148589be10ef3f8fffc70dcc14970ed77144a787c087dfcd0b5e2",
  "walletconflicts": [
  ],
  "time": 1462453065,
  "timereceived": 1462453065,
  "bip125-replaceable": "no",
  "details": [
    {
      "account": "",
      "address": "1GswWkrWvxGhCARW4w2ibnwk3pDURM6GpC",
      "category": "receive",
      "amount": 0.01000000,
      "label": "",
      "vout": 0
    }
  ],
  "hex": "0100000001e6980adab5435ca0214cb5e047c334a4d8f01d5c7b6f55f022ac903db7e75cb701
0000006b483045022100ddb1a8477d5ac64c688122f9ecb6306991ed3148d93639601f0ac145ca7bb49602
203b306c2d60555dc2ccf89636b7994b04a03960a6fc541c673a04036da641940f0121037d1588b4d483db
39a0d076f78d03181cf10dcf362cc06f2a01ff20d447c94388ffffffff0240420f00000000001976a914ae
2c30a645a14fa36425119e1d174b4d201b8f5988acc868420000000000001976a91474e79a48b73182083df3
7632bba597961ef51da788ac00000000"
}
```

Esta é a transação em sua forma simplificada, mostrando a quantia transacionada (`amount`), número de confirmações da transação (`confirmations`), o ID da transação (`txid`), a hora da transação (`time`), a hora que seu *client* ficou sabendo da transação (`timereceived`) e os detalhes da transação (`details`).

Para ver a transação no formato completo, você precisa usar o comando `getrawtransaction` com o `txid` como parâmetro que retornará a transação no formato de uma *string* hexadecimal para que você decodifique com o `decoderawtransaction` :

```
$ bitcoin-cli getrawtransaction 01ecaecc96b148589be10ef3f8fffc70dcc14970ed77144a787c08
7dfcd0b5e2
0100000001e6980adab5435ca0214cb5e047c334a4d8f01d5c7b6f55f022ac903db7e75cb7010000006b48
3045022100ddb1a8477d5ac64c688122f9ecb6306991ed3148d93639601f0ac145ca7bb49602203b306c2d
60555dc2ccf89636b7994b04a03960a6fc541c673a04036da641940f0121037d1588b4d483db39a0d076f7
8d03181cf10dcf362cc06f2a01ff20d447c94388ffffffff0240420f00000000001976a914ae2c30a645a1
4fa36425119e1d174b4d201b8f5988acc868420000000000001976a91474e79a48b73182083df37632bba597
961ef51da788ac00000000
```

```
$ bitcoin-cli decoderawtransaction 0100000001e6980adab5435ca0214cb5e047c334a4d8f01d5c7
b6f55f022ac903db7e75cb7010000006b483045022100ddb1a8477d5ac64c688122f9ecb6306991ed3148d
93639601f0ac145ca7bb49602203b306c2d60555dc2ccf89636b7994b04a03960a6fc541c673a04036da64
1940f0121037d1588b4d483db39a0d076f78d03181cf10dcf362cc06f2a01ff20d447c94388ffffffff024
0420f00000000001976a914ae2c30a645a14fa36425119e1d174b4d201b8f5988acc86820000000000197
6a91474e79a48b73182083df37632bba597961ef51da788ac00000000
{
  "txid": "01ecaecc96b148589be10ef3f8fffc70dcc14970ed77144a787c087dfcd0b5e2",
  "size": 226,
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "b75ce7b73d90ac22f0556f7b5c1df0d8a434c347e0b54c21a05c43b5da0a98e6",
      "vout": 1,
      "scriptSig": {
        "asm": "3045022100ddb1a8477d5ac64c688122f9ecb6306991ed3148d93639601f0ac145ca7b
b49602203b306c2d60555dc2ccf89636b7994b04a03960a6fc541c673a04036da641940f[ALL] 037d1588
b4d483db39a0d076f78d03181cf10dcf362cc06f2a01ff20d447c94388",
        "hex": "483045022100ddb1a8477d5ac64c688122f9ecb6306991ed3148d93639601f0ac145ca
7bb49602203b306c2d60555dc2ccf89636b7994b04a03960a6fc541c673a04036da641940f0121037d1588
b4d483db39a0d076f78d03181cf10dcf362cc06f2a01ff20d447c94388"
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.01000000,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 ae2c30a645a14fa36425119e1d174b4d201b8f59 OP_EQUALVER
IFY OP_CHECKSIG",
        "hex": "76a914ae2c30a645a14fa36425119e1d174b4d201b8f5988ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1GswWkrWvxGhCARW4w2ibnwk3pDURM6GpC"
        ]
      }
    },
    {
      "value": 0.04352200,
      "n": 1,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 74e79a48b73182083df37632bba597961ef51da7 OP_EQUALVER
IFY OP_CHECKSIG",
        "hex": "76a91474e79a48b73182083df37632bba597961ef51da788ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1Bf8qHi9q3ASGMHwf123E5RbTfMiky7TSy"
        ]
```

```
      }
    }
  ]
}
```

Ou juntando tudo em uma linha para melhorar a visualização...

```
$ bitcoin-cli decoderawtransaction $(bitcoin-cli getrawtransaction 01ecaecc96b148589be
10ef3f8fffc70dcc14970ed77144a787c087dfcd0b5e2)
{
  "txid": "01ecaecc96b148589be10ef3f8fffc70dcc14970ed77144a787c087dfcd0b5e2",
  "size": 226,
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "b75ce7b73d90ac22f0556f7b5c1df0d8a434c347e0b54c21a05c43b5da0a98e6",
      "vout": 1,
      "scriptSig": {
        "asm": "3045022100ddb1a8477d5ac64c688122f9ecb6306991ed3148d93639601f0ac145ca7b
b49602203b306c2d60555dc2ccf89636b7994b04a03960a6fc541c673a04036da641940f[ALL] 037d1588
b4d483db39a0d076f78d03181cf10dcf362cc06f2a01ff20d447c94388",
        "hex": "483045022100ddb1a8477d5ac64c688122f9ecb6306991ed3148d93639601f0ac145ca
7bb49602203b306c2d60555dc2ccf89636b7994b04a03960a6fc541c673a04036da641940f0121037d1588
b4d483db39a0d076f78d03181cf10dcf362cc06f2a01ff20d447c94388"
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.01000000,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 ae2c30a645a14fa36425119e1d174b4d201b8f59 OP_EQUALVER
IFY OP_CHECKSIG",
        "hex": "76a914ae2c30a645a14fa36425119e1d174b4d201b8f5988ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1GswWkrWvxGhCARW4w2ibnwk3pDURM6GpC"
        ]
      }
    },
    {
      "value": 0.04352200,
      "n": 1,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 74e79a48b73182083df37632bba597961ef51da7 OP_EQUALVER
IFY OP_CHECKSIG",
        "hex": "76a91474e79a48b73182083df37632bba597961ef51da788ac",
```

```
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1Bf8qHi9q3ASGMHwf123E5RbTfMiky7TSy"
        ]
      }
    }
  ]
}
```

Como você pode ver, desta forma nós conseguimos ver todos os elementos de uma transação com tudo que a rede precisa para verificar e validar ela. Nesta transação, eu usei N inputs e gerei N outputs sendo 10 *millibits* direcionado ao endereço da carteira no Bitcoin Core. Note que não existe a *fee* (ou taxa) de transação explícita; Ela nada mais é do que a diferença entre o todos os *inputs* e o total de todos *outputs*. Isso significa que qualquer fundo não gasto é considerado uma taxa de transação para recompensar o minerador responsável por incluir ela na blockchain.

Feito isso, vamos usar o Bitcoin Core para enviar estes bitcoins para outro endereço pela linha de comando. O primeiro comando que precisamos é o `listunspent` para listar todos os *outputs* de nossa carteira que já foram confirmados:

```
$ bitcoin-cli listunspent
  {
    "txid": "01ecaecc96b148589be10ef3f8fffc70dcc14970ed77144a787c087dfcd0b5e2",
    "vout": 0,
    "address": "1GswWkrWvxGhCARW4w2ibnwk3pDURM6GpC",
    "account": "",
    "scriptPubKey": "76a914ae2c30a645a14fa36425119e1d174b4d201b8f5988ac",
    "amount": 0.01000000,
    "confirmations": 1,
    "spendable": true
  }
]
```

Aqui, podemos ver que em `txid` está o *hash* da transação responsável por criar o *output* `vout` 0 contendo 0.01000000 designados ao endereço `1GswWkrWvxGhCARW4w2ibnwk3pDURM6GpC`, assim como o número de confirmações e o `scriptPubKey` (mais detalhes em Transações)

Para enviarmos estes bitcoins, teremos que criar uma transação que usa o *output* `vout` 0 da transação 01ecaecc96b148589be10ef3f8fffc70dcc14970ed77144a787c087dfcd0b5e2 e o referenciar no *input* da próxima transação que enviará para o próximo endereço.

Nós podemos ter uma visão mais detalhada do *output* que queremos usar com o comando `gettxout` dando o `txid` e o número do `vout` como parâmetros:

```
$ bitcoin-cli gettxout 01ecaecc96b148589be10ef3f8fffc70dcc14970ed77144a787c087dfcd0b5e
2 0
{
  "bestblock": "0000000000000000042e065b770fb46aba51bd389d086a1a2a47c5d9e5ffe575",
  "confirmations": 1,
  "value": 0.01000000,
  "scriptPubKey": {
    "asm": "OP_DUP OP_HASH160 ae2c30a645a14fa36425119e1d174b4d201b8f59 OP_EQUALVERIFY
OP_CHECKSIG",
    "hex": "76a914ae2c30a645a14fa36425119e1d174b4d201b8f5988ac",
    "reqSigs": 1,
    "type": "pubkeyhash",
    "addresses": [
      "1GswWkrWvxGhCARW4w2ibnwk3pDURM6GpC"
    ]
  },
  "version": 1,
  "coinbase": false
}
```

Então, como você pode ver, este *output* designou 10 *millibits* para o endereço `1GswWkrWvxGhCARW4w2ibnwk3pDURM6GpC` ter o direito de gastar como pode ser verificado junto com o *script* de transação mais comum da rede (mais detalhes sobre isso em Transações). Para continuar a operação de envio dos bitcoins, vamos criar dois novos endereços para enviar os bitcoins:

```
$ bitcoin-cli getnewaddress
1N8A7WLw4TA8Vbu6GQ9JSotbvtTPY5uKhf  # <--- endereço recipiente principal
$ bitcoin-cli getnewaddress
1EoPUX89wRFGHRNGJFTzqfNcRgdzgs5JhK  # <--- geramos um outro para o "troco" da transaçã
o
```

O próximo comando que usaremos para criar a transação é o `createrawtransaction`, porém, antes devemos entender uma coisa importante sobre transações: **todos** os *inputs* devem ser gastos **completamente**. Isso é importante saber desde já, pois qualquer erro de programação na hora de montar uma transação em que não se gaste todos os *inputs* significará perda financeira. O que acontece é que qualquer valor que não seja explicitamente gasto numa transação será considerado como *fee* de mineração para o minerador que incluir a sua transação na blockchain. Logo, a diferença da soma de todos os *inputs* menos a soma de todos os *outputs* é igual à taxa que será paga ao minerador (soma dos *inputs* - soma dos *outputs* = taxa de mineração). Entendendo isso, podemos criar a transação com o `createrawtransaction`:

```
$ bitcoin-cli createrawtransaction '[{"txid" : "01ecaecc96b148589be10ef3f8fffc70dcc149
70ed77144a787c087dfcd0b5e2", "vout" : 0}]' '{"1N8A7WLw4TA8Vbu6GQ9JSotbvtTPY5uKhf": 0.0
06, "1EoPUX89wRFGHRNGJFTzqfNcRgdzgs5JhK": 0.0039}'
0100000001e2b5d0fc7d087c784a1477ed7049c1dc70fcfff8f30ee19b5848b196ccaeec010000000000ff
ffffff02c027090000000000001976a914e7b528dec5f14d053d6f32d8508ccb287932c43188ac70f3050000
0000001976a914975f87b998f171ee6c6426e2b018414d2abec50f88ac00000000
```

O que criamos acima é uma transação enviando `0.006` bitcoins para `1N8A7WLw4TA8Vbu6GQ9JSotbvtTPY5uKhf` como simulação de um endereço para um pagamento ou qualquer outra coisa, usamos o endereço `1EoPUX89wRFGHRNGJFTzqfNcRgdzgs5JhK` como endereço de troco atribuindo `0.0039` bitcoins a ele; o que deixa `0.0001` (`0.006 - 0.0039 = 0.0001`) como taxa de transação.

Para verificar que a transação foi formada corretamente como o esperado, podemos usar o comando `decoderawtransaction` com a transação codificada como parâmetro:

```
$ bitcoin-cli decoderawtransaction 0100000001e2b5d0fc7d087c784a1477ed7049c1dc70fcfff8f
30ee19b5848b196ccaeec010000000000ffffffff02c027090000000001976a914e7b528dec5f14d053d6
f32d8508ccb287932c43188ac70f30500000000001976a914975f87b998f171ee6c6426e2b018414d2abec
50f88ac00000000
{
  "txid": "88f124d1cc2198c8103cfb707849f091589734a42413baf2323936463e905f53",
  "size": 119,
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "01ecaecc96b148589be10ef3f8fffc70dcc14970ed77144a787c087dfcd0b5e2",
      "vout": 0,
      "scriptSig": {
        "asm": "",
        "hex": ""
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.00600000,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 e7b528dec5f14d053d6f32d8508ccb287932c431 OP_EQUALVER
IFY OP_CHECKSIG",
        "hex": "76a914e7b528dec5f14d053d6f32d8508ccb287932c43188ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1N8A7WLw4TA8Vbu6GQ9JSotbvtTPY5uKhf"
        ]
      }
    },
    {
      "value": 0.00390000,
      "n": 1,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 975f87b998f171ee6c6426e2b018414d2abec50f OP_EQUALVER
IFY OP_CHECKSIG",
        "hex": "76a914975f87b998f171ee6c6426e2b018414d2abec50f88ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1EoPUX89wRFGHRNGJFTzqfNcRgdzgs5JhK"
        ]
      }
    }
  ]
}
```

Parece tudo certo. E, como pode ver, o `scriptSig` desta transação está vazio, pois não assinamos ainda. Sem a assinatura digital, esta transação é apenas um *template* que qualquer um poderia ter formado sem valor financeiro algum. Nós precisamos da assinatura para destrancar os *inputs* que estamos usando na transação e antes devemos destrancar a carteira com a senha para podermos expor a chave privada para a assinatura:

```
$ bitcoin-cli walletpassphrase naouseestasenhaidiota 200
```

E assinamos...

```
$ bitcoin-cli signrawtransaction
{
  "hex": "0100000001e2b5d0fc7d087c784a1477ed7049c1dc70fcfff8f30ee19b5848b196ccaeec0100
0000006a47304402205984d90f15c019c0804ae140613f6ab5819bd44b6e1faa08cf9aba39655d32f80220
2b3e5f40dd0f7c8b17381cfff466fae17f7c80e20190ce0caf81962389b80aaf0121039e129c7aec71c340
73909fa428982674d305f984fbbc4acda68046a9cd07598bfffffffff02c027090000000001976a914e7b5
28dec5f14d053d6f32d8508ccb287932c43188ac70f30500000000001976a914975f87b998f171ee6c6426
e2b018414d2abec50f88ac00000000",
  "complete": true
}
```

Agora temos a transação assinada, retornada pelo `signrawtransaction` como podemos verificar com o `decoderawtransaction`:

```
$ bitcoin-cli decoderawtransaction 0100000001e2b5d0fc7d087c784a1477ed7049c1dc70fcfff8f
30ee19b5848b196ccaeec01000000006a47304402205984d90f15c019c0804ae140613f6ab5819bd44b6e1
faa08cf9aba39655d32f802202b3e5f40dd0f7c8b17381cfff466fae17f7c80e20190ce0caf81962389b80
aaf0121039e129c7aec71c34073909fa428982674d305f984fbbc4acda68046a9cd07598bfffffffff02c02
709000000000001976a914e7b528dec5f14d053d6f32d8508ccb287932c43188ac70f30500000000001976a
914975f87b998f171ee6c6426e2b018414d2abec50f88ac00000000
{
  "txid": "51cf064af1e25c9a7a299bd3fec3a7a41f318febd17c661656c5134f2ca63044",
  "size": 225,
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "01ecaecc96b148589be10ef3f8fffc70dcc14970ed77144a787c087dfcd0b5e2",
      "vout": 0,
      "scriptSig": {
        "asm": "304402205984d90f15c019c0804ae140613f6ab5819bd44b6e1faa08cf9aba39655d32
f802202b3e5f40dd0f7c8b17381cfff466fae17f7c80e20190ce0caf81962389b80aaf[ALL] 039e129c7a
ec71c34073909fa428982674d305f984fbbc4acda68046a9cd07598b",
        "hex": "47304402205984d90f15c019c0804ae140613f6ab5819bd44b6e1faa08cf9aba39655d
32f802202b3e5f40dd0f7c8b17381cfff466fae17f7c80e20190ce0caf81962389b80aaf0121039e129c7a
ec71c34073909fa428982674d305f984fbbc4acda68046a9cd07598b"
      },
      "sequence": 4294967295
```

```
    }
  ],
  "vout": [
    {
      "value": 0.00600000,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 e7b528dec5f14d053d6f32d8508ccb287932c431 OP_EQUALVER
IFY OP_CHECKSIG",
        "hex": "76a914e7b528dec5f14d053d6f32d8508ccb287932c43188ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1N8A7WLw4TA8Vbu6GQ9JSotbvtTPY5uKhf"
        ]
      }
    },
    {
      "value": 0.00390000,
      "n": 1,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 975f87b998f171ee6c6426e2b018414d2abec50f OP_EQUALVER
IFY OP_CHECKSIG",
        "hex": "76a914975f87b998f171ee6c6426e2b018414d2abec50f88ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1EoPUX89wRFGHRNGJFTzqfNcRgdzgs5JhK"
        ]
      }
    }
  ]
}
```

Okay, nosso *input* agora tem o `scriptSig` preenchido e passa a ter valor real. Só assim ela pode ser verificada e validada pelos outros nós na rede.

Finalmente, vamos enviar a transação assinada para a rede com `sendrawtransaction` para que a rede conheça e propague nossa transação:

```
$ bitcoin-cli sendrawtransaction 0100000001e2b5d0fc7d087c784a1477ed7049c1dc70fcfff8f30
ee19b5848b196ccaeec01000000006a47304402205984d90f15c019c0804ae140613f6ab5819bd44b6e1fa
a08cf9aba39655d32f802202b3e5f40dd0f7c8b17381cfff466fae17f7c80e20190ce0caf81962389b80aa
f0121039e129c7aec71c34073909fa428982674d305f984fbbc4acda68046a9cd07598bffffffff02c0270
900000000001976a914e7b528dec5f14d053d6f32d8508ccb287932c43188ac70f30500000000001976a91
4975f87b998f171ee6c6426e2b018414d2abec50f88ac00000000

51cf064af1e25c9a7a299bd3fec3a7a41f318febd17c661656c5134f2ca63044
```

Agora temos o *hash* desta transação (*txid*) e podemos usar sobe ele os mesmos comandos que usamos com outras transações para olhar cada detalhe desta transação. Vale lembrar que, atualmente, este *txid* ainda pode mudar até que esta transação seja incluída na blockchain com o *txid* final.

Uma última coisa a se reparar é o fato de o Bitcoin, diferente de cartões de crédito e outros meios do sistema financeiro atual ser muito mais seguro pelo simples fato de que nenhuma informação secreta teve que ser enviada à rede - como nome, documentos, número de cartão de crédito, etc. Tudo que enviamos é uma transação com dados públicos e que não pode ser forjada por ninguém com a informação contida nela sem a chave privada para assinar digitalmente a transação. Num mundo conectado onde o roubo de identidade é um dos crimes mais lucrativos do mundo, esta característica é claramente uma gigantesca vantagem deste sistema.

Blocos

Você também pode explorar blocos com a API JSON-RPC do Core. Vejamos o bloco da transação de 0.010 btc recebida usando o comando `getblock` com o *hash* do bloco como parâmetro (presente na transação após confirmada):

```
$ bitcoin-cli getblock 000000000000000042e065b770fb46aba51bd389d086a1a2a47c5d9e5ffe57
5
{
  "hash": "000000000000000042e065b770fb46aba51bd389d086a1a2a47c5d9e5ffe575",
  "confirmations": 6,
  "size": 998202,
  "height": 410350,
  "version": 805306368,
  "merkleroot": "df4984d291da6a74e2fad8f88f2df313c153815b3a11420403f72b2707b061ef",
  "tx": [
    "3f3de9824ef4fd1f65f8335e2022db559547dc8a6377124f1ef585bddfeef63e",
    "9733a90a77833f51886884f76cb50a3ca358615d3b28193ad51e4d3668b6de83",
    #  [... Muito mais transações ...]
    "76eb383c8a198b3ab56b4d469bb759e8ac75f04dc5e8e3a59b39c55bef93afcb",
    "01ecaecc96b148589be10ef3f8fffc70dcc14970ed77144a787c087dfcd0b5e2", <--- Nossa tra
nsação recebida
    "3c48a57facd63881c4df869a680f73ed751e2a9d00ff16dd5dfcbe54b61bfa23",
    # [... Muito mais transações ...]
  ],
  "time": 1462453539,
  "mediantime": 1462448476,
  "nonce": 1818589289,
  "bits": "18062776",
  "difficulty": 178659257772.5273,
  "chainwork": "00000000000000000000000000000000000000000001858c55a5df5a426cda2d0",
  "previousblockhash": "000000000000000021ff8d9b19b4b760670d593e2958a4410880dc6ce14f0
5a",
  "nextblockhash": "000000000000000000520dd8c8ca2ce480d6bceee1d51916f9befbba0e00d0359"
}
```

Para pegarmos o *hash* de um bloco pela *height* (ou altura) dele na blockchain, usamos o comando `getblockhash` e o índice do bloco como parâmetro. A altura ou indíce do primeiro bloco é 0 e, desde então, a cada novo bloco ela é incrementada. Para vermos o famoso bloco *genesis* minerado por Satoshi Nakamoto, basta que usemos o parâmetro 0:

```
$ bitcoin-cli getblockhash 0
000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
```

E agora podemos usar o comando anterior `getblock` com o *hash* retornado como parâmetro:

```
$ bitcoin-cli getblock 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26
f
{
    "hash" : "000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",
    "confirmations" : 286388,
    "size" : 285,
    "height" : 0,
    "version" : 1,
    "merkleroot" : "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b",
    "tx" : [
        "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b"
    ],
    "time" : 1231006505,
    "nonce" : 2083236893,
    "bits" : "1d00ffff",
    "difficulty" : 1.00000000,
    "chainwork" : "0000000000000000000000000000000000000000000000000000000100010001",
    "nextblockhash" : "00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb604
8"
}
```

Os comandos `gettransaction`, `getblock` e `getblockhash` tem tudo o que você precisa para explorar a blockchain com o Bitcoin Core passando de uma transação para outra e de um bloco para outro como necessário.

Outras Implementações e Ferramentas

O cliente bitcoind do Bitcoin Core é a implementação referência atual, porém, pela natureza *open-source* do Bitcoin, há uma variedade enorme de implementações e ferramentas alternativas em diferentes linguagens e com diferentes intuitos. Algumas delas são:

libbitcoin: Ferramenta para desenvolvimento Bitcoin multi-plataforma implementada em C++

libbitcoin explorer: Ferramenta de Linha de Comando para o Bitcoin

libbitcoin server: *Fullnode* e *Query Server* Bitcoin

btcd: Implementação do Bitcoin escrita em Go (golang)

bitcoinj: *Library* Bitcoin escrita em Java

python-bitcoinlib: *Library* Bitcoin escrita em Python

picocoin: *Library* Bitcoin escrita em C

... e muitas outras.

Próximo capítulo: Endereços e Carteiras

Endereços e Carteiras

Como visto em A Criptografia no Bitcoin, a criptografia de chave pública é essencial para resolver a questão de propriedade em sistemas computacionais como no caso do Bitcoin junto com o apoio criptográfico das funções *hash* criptográficas. O que antes eram apenas 0's e 1's indistinguíveis ganham algumas propriedades como escasses, autenticidade e integridade. E isto é a base para que se entenda o que os endereços representam e como eles são tratados pela rede.

Criando Endereços

Cada endereço é um identificador único na rede; não de uma pessoa como *e-mails*, mas de uma chave pública. Para podermos criar um endereço sob nosso controle, primeiro temos que gerar uma chave privada de onde calcularemos a chave pública que será identificada como um endereço.

Gerando a Chave Privada

A chave privada é um número gerado de forma aleatória. E além da aleatoriedade na geração deste número, outra qualidade importante para a segurança na implementação deste tipo de criptografia é que a chave privada é um número gigante. Com esta chave, as transações serão assinadas e valores serão transferidos na rede; gerar e manter esta chave de forma segura é **essencial** para a segurança de valores em bitcoin.

Para a geração segura de uma chave privada, primeiro precisamos de uma fonte segura de entropia para geração de uma chave de 256 *bits*. Normalmente, os *softwares* de carteira utilizam uma fonte de coleta de entropia do próprio sistema operacional - como `/dev/urandom` em sistemas UNIX ou `cryptGenRandom()` no Windows - e isto costuma ser o suficiente, mas para o nível máximo de paranoia é possível que utilizemos métodos físicos que excluem a possibilidade de falhas ou *backdoors* de *software*/*hardware* como o *diceware* descrito especialmente para o Bitcoin aqui. No mínimo, recomendo que gere a chave privada pelo método *diceware* descrito no link anterior pelo menos uma vez na vida devido à experiência que te ajudará a entender ou fortalecer na prática algumas coisas a mais sobre entropia.

> Como programador(a) é a sua responsabilidade conhecer e entender a ferramenta que você estiver utilizando para a geração de números aleatórios criptograficamente seguros (*cryptographically secure pseudo-random number generator* - CSPRNG). Não use uma técnica própria desconhecida ao resto do mundo mesmo que você tenha a capacidade de cria uma técnica razoavelmente segura - parte fundamental na segurança de um algoritmo deste tipo é o fato de ele ter sido revisado por muitos especialistas e utilizado em "batalha" ao longo de muitos anos. Não confie nem mesmo nas ferramentas que eu apontar sem que você entenda porque elas são seguras e/ou que você possa verificar ampla confiança consolidada de outros especialistas nestas ferramentas. Dito isto, voltemos ao material...

Em Python, podemos utilizar o `os.urandom` que coleta *bytes* a partir do `/dev/urandom` em sistema UNIX e `cryptGenRandom()` no Windows para geração da chave privada em *bytes*:

```
>>> import os

>>> privkey = os.urandom(32)
>>> print(privkey)
b'\x9a\xfbch\x05\xc3|\xdf\xfc\xebJ\x89g\xdf\xfcn\xbe\x80%m\x91\x80DI\xd0hs6\x04\x84~\x
e8'
#  para termos uma ideia do tamanho do número, veja em int...
privkey_int = int.from_bytes(privkey, byteorder='little')
>>> print(privkey_int)
105160114745571781986276553867283233264261203826988260355664227503456152451994
# e em hexadecimal como é comum ser enviado e recebido pelos cabos...
# (obviamente, em sistema seguros e de seu controle caso seja REALMENTE necessário)
>>> privkey_hex = hex(privkey_int)
>>> print(privkey_hex)
0xe87e8404367368d0494480916d2580be6efcdf67894aebfcdf7cc3056863fb9a
```

Agora temos 32 *bytes* (ou 256 *bits*) coletados. Certamente nenhum dos formatos mostrados parece com o que costumamos ler ao pedir que uma carteira exporte a chave privada para nós. Isto ocorre porque, diferente de quando estamos passando informação pelos cabos, costumamos utilizar um formato especial chamado WIF.

> **WIF**: este é o formato que costumamos ler e é abreviação para *Wallet Import Format* (Formato de Importação de Carteira). Este formato é usado para facilitar a leitura e cópia das chaves por humanos. O processo é simples: pegamos a chave privada, concatenamos o *byte* `0x80` para `mainet` ou `0xef` para a `testnet` como prefixo e o *byte* `0x01` como sufixo se a chave privada for corresponder a uma chave pública comprimida - uma forma de representar chaves públicas usando menos espaço -, realizamos uma função *hash* SHA-256 duas vezes seguidas - comumente referido como *shasha* -, pegamos os 4 primeiros *bytes* do resultado - este é o *checksum* -, adicionamos estes 4 *bytes* ao final do resultado da chave pública antes do *shasha*, e então, usamos a funcão *Base58Check* para codificar o resultado final.
>
> **Base58Check**: é uma versão modificada da função de Base58 utilizada no Bitcoin para codificar chaves e endereços na rede para produzir um formato que facilita a digitação por humanos, assim como diminui drasticamente a chance de erros na digitação limitando os endereços a caracteres específicos que não sejam visualmente idênticos em algumas fontes - como 1 e I ou 0 e O - e outros problemas parecidos no envio e recebimento de endereços por humanos.

Então, vamos pegar a chave privada que criamos e transformar ela para o formato WIF. Ao finalizarmos, como teste do resultado, pegue a chave no formato WIF e importe ela em algum *software* de carteira e verá que a sua carteira criará endereços a partir desta chave privada normalmente. Aqui está uma forma como podemos transformar a chave privada que obtivemos acima para o formato WIF:

```
#!/usr/bin/env python3
import hashlib
from base58 import b58encode

# definimos a dupla rodada de sha-256 para melhorar a legibilidade...
def shasha(data):
    """SHA256(SHA256(data)) -> HASH object"""
    result = hashlib.sha256(hashlib.sha256(data).digest())
    return result

# agora criamos a função que passará a chave para formato WIF...
def privkey_to_wif(rawkey, compressed=True):
    """Converte os bytes da chave privada para WIF"""
    k = b'\x80' + rawkey  # adicionamos o prefixo da mainet

    # por padrão criamos formado comprimido
    if compressed:
        k += b'\x01'  # sufixo para indicar chave comprimida

    checksum = shasha(k).digest()[:4]  # os primeiros 4 bytes da chave como checksum
    key = k + checksum

    b58key = b58encode(key)
    return b58key

# agora podemos usar a função privkey_to_wif com a nossa chave privada
# privkey obtida no exemplo anterior
privkey_wif = privkey_to_wif(privkey)
print(privkey_wif)
# L2QyYCh5nFDe4yRX8hBRMhAGNnHYQmyrWbH1HT7YJohYULHbthxg
```

Este é o formato final que nos permite importar para as carteiras que permitem este tipo de operação. Note este mesmo formato ssendo exportado pelo Bitcoin Core, por exemplo, com o comando `dumpprivkey`:

```
$ bitcoin-cli getnewaddress
193AUxttHmHQLajJ1pnHMvk5d9WvbuvvFR
$ bitcoin-cli dumpprivkey "193AUxttHmHQLajJ1pnHMvk5d9WvbuvvFR"
KxBBVbkgku7f5XudmAizo51h8pfBTHDhL1u167EMgKS7PK3bnrwc
```

Gerando a Chave Pública

Aqui é onde faremos a primeira operação exclusiva de Criptografia de Chave Pública ao usarmos o ECDSA (*Elliptic Curve Digital Signature Algorithm*) para gerarmos a nossa chave pública a partir da chave privada que criamos anteriormente. Para este tipo de operação, recomendo que utilize *libraries* como python-ecsda, secp256k1-py (*binding* direto com a secp256k1 escrita em C) ou alguma *lib* já bem utilizada em produção na sua linguagem de

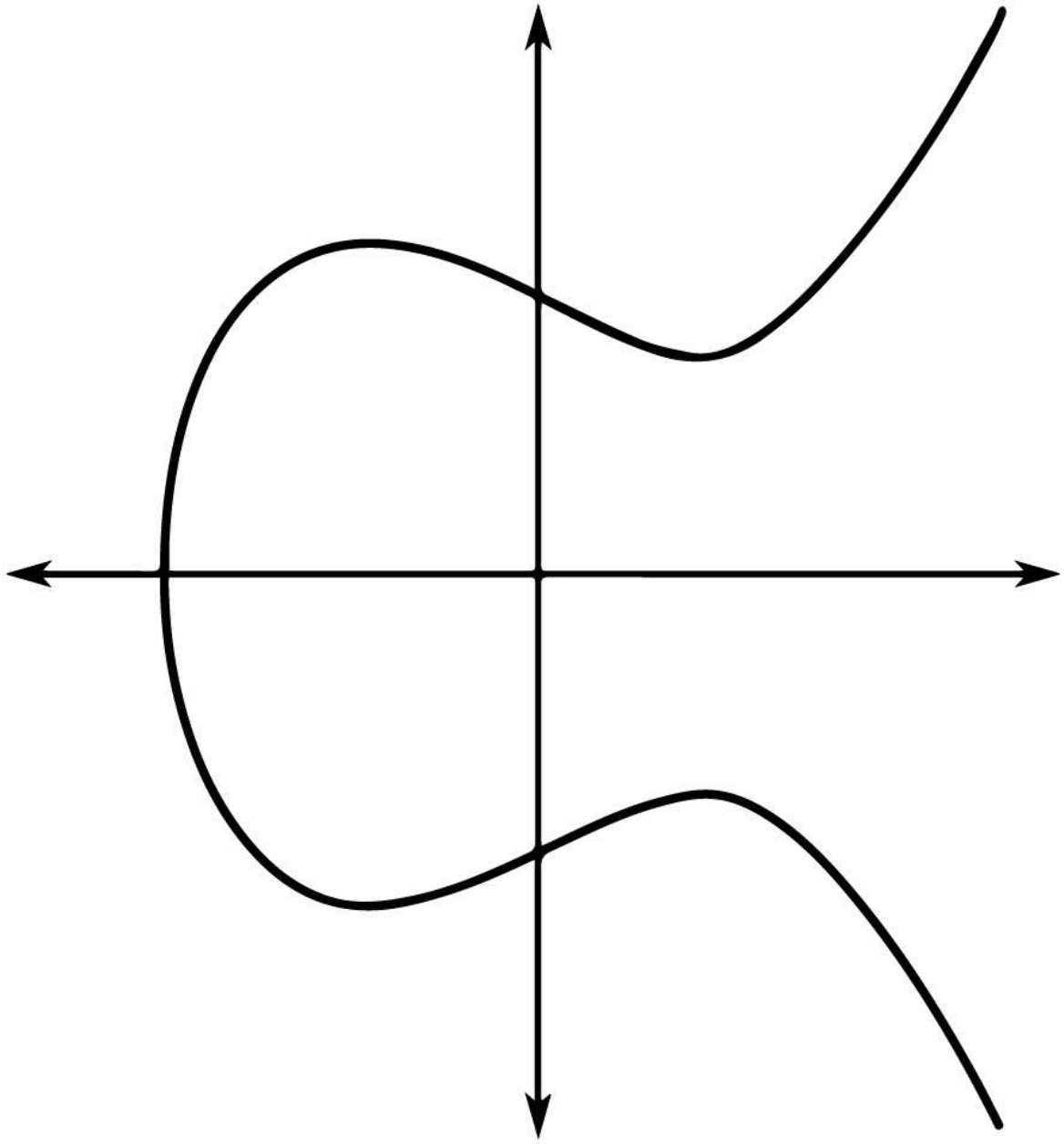
preferência - melhor ajudar a melhorar estas *libs* do que fazer uma nova implementação à toa e apresentar um novo risco sem benefício. Mas, para nosso fim didático deste material, vamos ver como calculamos a chave pública a partir da chave privada em Python para simplificar a visualização e entendimento.

> O código abaixo é uma versão útil apenas como referência didática. Não recomendo que use este código em produção de forma alguma! Este código não foi testado e revisado como necessário para os padrões de segurança compatíveis com criptografia, e provavelmente, há mais razões para não fazer operações criptográficas em puro Python do que átomos no Universo observável. **VOCÊ FOI AVISADO(A)**.

Agora que você sabe que é uma péssima ideia usar o código abaixo em qualquer ambiente de produção com valores reais, podemos continuar com a geração da chave pública. Já tendo a chave privada que criamos acima na variável `privkey` que será utilizada ao longo deste exemplo:

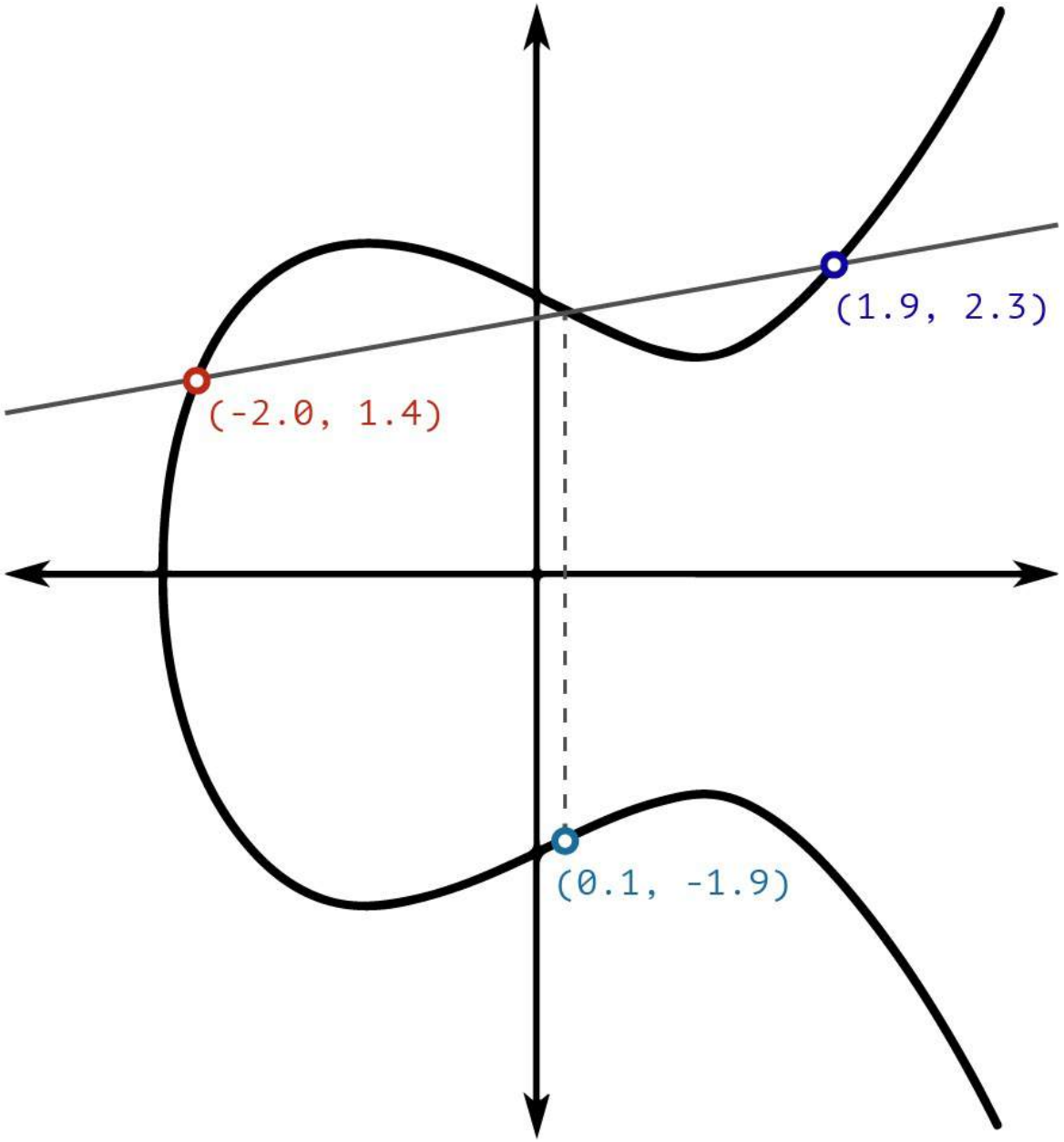
```
>>> print(privkey)
b'\x9a\xfbch\x05\xc3|\xdf\xfc\xebJ\x89g\xdf\xfcn\xbe\x80%m\x91\x80DI\xd0hs6\x04\x84~\x
e8'
```

Como comentado anteriormente, o Bitcoin utiliza a implementação de Curva Elíptica secp256k1 definida pela equação `y^2 == x^3 + 7 (mod p)` sendo `p = 2^256 - 2^32 - 977`. A curva elíptica que estamos trabalhando pode ser graficamente representada por:

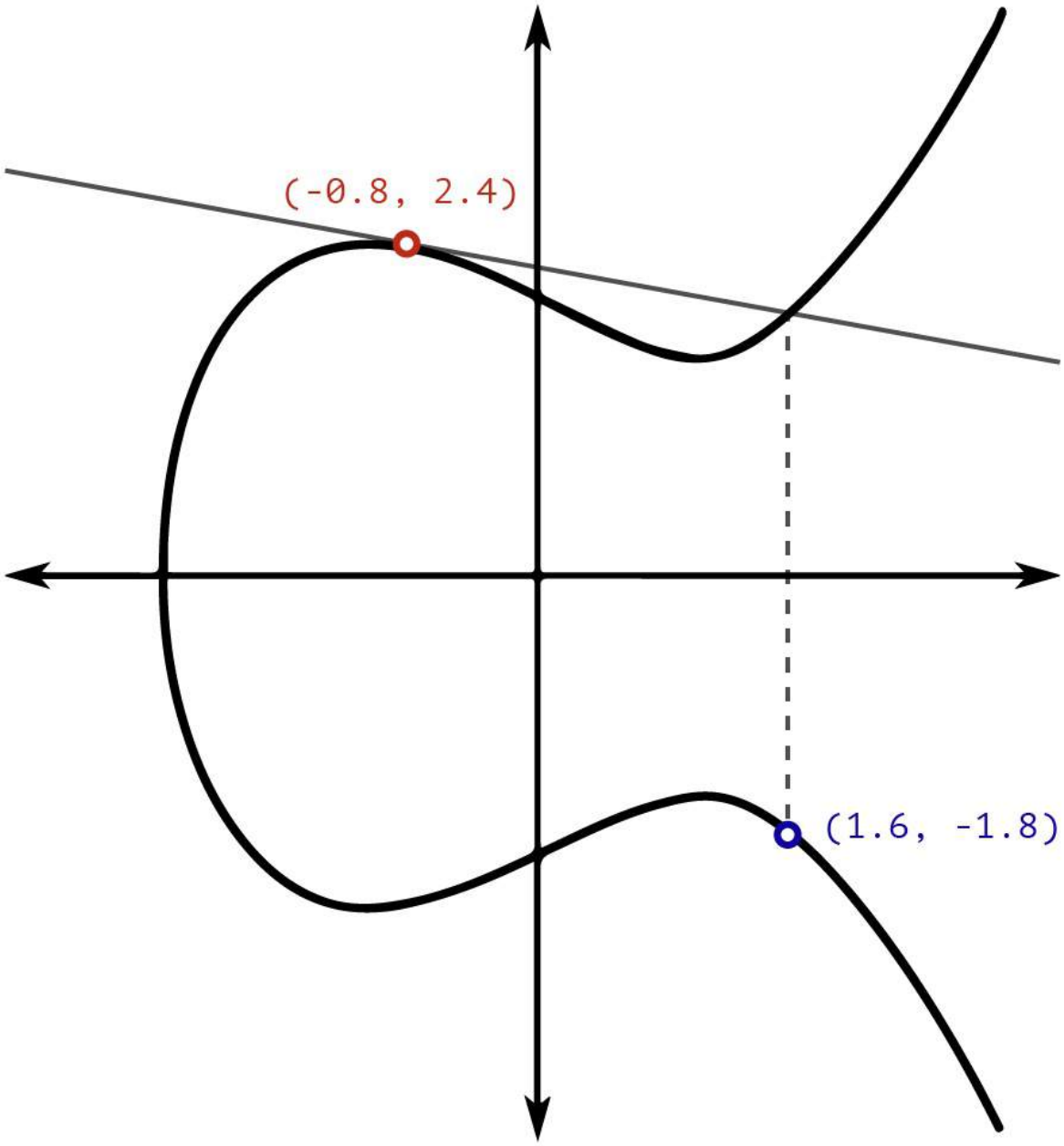


Para que calculemos a curva usada em nosso esquema de criptografia de chave pública, temos que ter o poder de duas operações: Adição e multiplicação de pontos. A utilidade deste método para o fim que pretendemos começa a ser visualizada aqui ao perceber que as únicas operações que podemos fazer com pontos são adição e multiplicação.

Para adicionarmos o Ponto A ao Ponto B de uma curva como esta, primeiro desenhamos uma linha entre os dois pontos A e B, esta linha fará uma interseção com um terceiro ponto na curva, então só precisamos refletir esta interseção passando pelo eixo x para que tenhamos o resultado da soma dos pontos A e B:



E para adicionarmos um mesmo ponto A a ele mesmo, nós desenhamos uma linha tangencial à curva tocando no Ponto A, esta linha terá uma interseção com a curva formando um segundo ponto, daí basta que façamos a reflexão como no primeiro caso e achamos o resultado de `2 * Ponto A`:



Como multiplicação nada mais é do que adicionar um mesmo valor por ele *n* vezes, nós já temos o que precisamos para realizar as operações necessárias.

Para obtermos a chave pública, a operação que faremos será a multiplicação da nossa chave privada `privkey` por um ponto inicial definido pela implementação da curva conhecido por todos. Este ponto se chama Ponto Gerador (abreviado como G em futuras referências) e na secp256k1, ele é:

```
Gx = 55066263022277343669578718895168534326250603453777594175500187360389116729240
Gy = 32670510020758816978083085130507043184471273380659243275938904335757337482424
```

Sim, os números precisam ser gigantes para que este tipo de esquema criptográfico seja seguro. Como nota de curiosidade, o número total de chaves privadas possíveis no Bitcoin é algo em torno de 2**256; isto é um número de grandeza comparável com o número**

estimado de átomos no Universo observável (sim, de verdade) que está entre 1077 e 10**82.

A nossa chave pública, então, será o ponto gerado a partir da multiplicação da chave privada por G (`privkey * (Gx, Gy)`). Assim, temos que implementar uma multiplicação escalar para multiplicarmos a `privkey` como `int` pelo vetor G implementado como um `set` . Vamos abstrair esta operação em quatro funções: uma para calcular a inversa modular de `x (mod p)` , uma para caluclar o dobro de um ponto - ou seja, a soma de um ponto por ele mesmo -, mais uma para adicionar um ponto a outro ponto qualquer e uma outra função para multiplicar um ponto por valores arbitrários (no caso, a chave privada):

```
#!/usr/bin/env python3

# primeiro vamos definir o "p" da fórmula usada no bitcoin "y^2 == x^3 + 7 (mod p)"
p = 2**256 - 2**32 - 977

def inverse(x, p):
    """
    Calcula inversa modular de x (mod p)
    A inversa modular de um número é definida:
    (inverse(x, p) * x) == 1
    """
    inv1 = 1
    inv2 = 0
    while p != 1 and p != 0:
        inv1, inv2 = inv2, inv1 - inv2 * (x // p)
        x, p = p, x % p

    return inv2


def double_point(point, p):
    """
    Calcula point + point (== 2 * point)
    """
    (x, y) = point
    if y == 0:
        return None

    # Calculate 3*x^2/(2*y)  modulus p
    slope = 3*pow(x, 2, p) * inverse(2 * y, p)

    xsum = pow(slope, 2, p) - 2 * x
    ysum = slope * (x - xsum) - y
    return (xsum % p, ysum % p)


def add_point(p1, p2, p):
    """
    Calcula p1 + p2
```

```
    """
    (x1, y1) = p1
    (x2, y2) = p2
    if x1 == x2:
        return double_point(p1, p)

    # calcula (y1-y2)/(x1-x2)  modulo p
    # slope é o coeficiente angular da curva
    slope = (y1 - y2) * inverse(x1 - x2, p)
    xsum = pow(slope, 2, p) - (x1 + x2)
    ysum = slope * (x1 - xsum) - y1
    return (xsum % p, ysum % p)


def point_mul(point, a, p):
    """
    Multiplicação escalar: calcula point * a
    """
    scale = point
    acc = None
    while a:
        if a & 1:
            if acc is None:
                acc = scale
            else:
                acc = add_point(acc, scale, p)
        scale = double_point(scale, p)
        a >>= 1
    return acc
```

Agora utilizamos estas funções com os valores que temos da `privkey` e as coordenadas de G escritas em hexadecimal em `int`:

```
>>> privkey = 0xe87e8404367368d0494480916d2580be6efcdf67894aebfcdf7cc3056863fb9a
>>> g_x = 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798
>>> g_y = 0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8
>>> g = (g_x, g_y)

>>> pub_x, pub_y = point_mul(g, privkey, p)
>>> print("x: %x, y: %x" % (pub_x, pub_y))
# Com os valores acima, print deve imprimir as coordenadas da nossa chave pública na c
urva que é:
# 273f9c55a1c8976f87032aade62b794df31e64327386b403d7438c735b2f7c89, 9848db72f0b7964636
4e508d0f591d3a80541f8138f44722ada5220608f79805
```

O resultado da final da chave pública no ECDSA é simplesmente as coordenadas da chave pública (32 *bytes* para cada uma) com um *byte* `0x04` como sufixo. Podemos simplesmente fazer algo assim:

```
>>> pubkey = '04' + format(pub_x, 'x') + format(pub_y, 'x')
>>> print(pubkey)
04273f9c55a1c8976f87032aade62b794df31e64327386b403d7438c735b2f7c899848db72f0b79646364e
508d0f591d3a80541f8138f44722ada5220608f79805
```

Agora que temos a nossa chave pública, podemos seguir com a geração do endereço Bitcoin correspondente a ela.

Gerando o Endereço Bitcoin

Continuando com os valores que conseguimos acima para a nossa chave pública, vamos gerar um endereço Bitcoin de acordo com as regras no protocolo.

Primeiro, pegamos o valor de `pubkey` e aplicamos a função *hash* SHA-256 em seu *bytes*:

```
>>> import codecs
>>> import hashlib

>>> pubkey_bytes = codecs.decode(pub.encode('utf-8'), 'hex')
>>> print(pubkey_bytes)
b'\x04\'?\x9cU\xa1\xc8\x97o\x87\x03*\xad\xe6+yM\xf3\x1ed2s\x86\xb4\x03\xd7C\x8cs[/|\x8
9\x98H\xdbr\xf0\xb7\x96F6NP\x8d\x0fY\x1d:\x80T\x1f\x818\xf4G"\xad\xa5"\x06\x08\xf7\x98
\x05'
>>> pubkey_sha256 = hashlib.sha256(pubkey_bytes)
>>> print(pubkey_sha256.hexdigest())
'd3feedd34006df75cba68a5de79228e4a6956436f46aba74332eeada8ac2ec47'
```

Segundo, aplicamos a função *hash* RIPEMD-160 no resultado `pubkey_sha256`:

```
>>> pubkey_ripe = hashlib.new('ripemd160')
>>> pubkey_ripe.update(pubkey_sha256.digest())
>>> print(pubkey_ripe.hexdigest())
'0fc8aa8b93103a388fd562514ec250be2d403a27'
```

Terceiro, adicionamos o byte `0x00` para identificar a chave para a *mainet*. De forma simples, podemos:

```
>>> raw_address = '00' + pubkey_ripe.hexdigest()
>>> print(raw_address)
000fc8aa8b93103a388fd562514ec250be2d403a27
```

E este resultado é o endereço Bitcoin. Guarde-o, pois já usaremos ele novamente. Mais uma vez não se parece com os endereços que costumamos ver e escrever. Isto ocorre porque este é o endereço puro em hexadecimal sem Base58Check. Para termos um

endereço com proteção contra erros de digitação como a maioria dos endereços que lidamos no dia-a-dia, seguimos os seguintes passos.

Primeiro vamos pegar 4 *bytes* para usar como *checksum* fazendo o *shasha* nos bytes do endereço. Como dito acima costumamos chamar de *shasha* quando tiramos um *hash* SHA-256 e passamos este resultado novamente na função *hash* SHA256. Para isso, fazemos:

```
>>> raw_addr_bytes = codecs.decode(raw_address.encode('utf-8'), 'hex')
>>> print(raw_addr_bytes)
b"\x00\x0f\xc8\xaa\x8b\x93\x10:8\x8f\xd5bQN\xc2P\xbe-@:'"
>>> addr_shasha = hashlib.sha256(hashlib.sha256(raw_addr_bytes).digest())
>>> print(addr_shasha.hexdigest())
455a7335c8c121fbb90e22e0dbb77ff3d7137908a052b895d6933f53032b2d27
# E pegamos os 4 primeiros bytes como checksum
>>> checksum = addr_shasha.digest()[:4]
>>> print(checksum)
b'EZs5'
# Em hexadecimal...
>>> checksum = addr_shasha.hexdigest()[:8]
>>> print(checksum)
'455a7335'
```

Agora adicionamos o *checksum* ao final do endereço `000fc8aa8b93103a388fd562514ec250be2d403a27` que pegamos um pouco mais acima, e finalmente, passamos os *bytes* pela função para codificar em Base58:

```
>>> from base58 import b58encode
>>> address = raw_address + checksum
>>> addr_bytes = codecs.decode(address.encode('utf-8'), 'hex')
>>> address = b58encode(addr_bytes)
>>> print(address)
12STXQicaWRh4RFfjW6T6y6ZAqQVytTKXa
```

E este é o endereço que podemos usar para receber transações nos *softwares* de carteira controlado pela chave privada criada no início.

> Repare que com o uso de 4 *bytes* como *checksum* junto om o Base58Check, nós temos um endereço com proteção contra erros de digitação contanto que o *software* de carteira o implemente corretamente. O Base58Check serve para que os endereços gerados sejam de mais facilidade visual e o *checksum* serve para garantir que o endereço realmente foi digitado corretamente antes de enviar uma transação. Sem esta proteção, a chance de erros com perdas financeiras seria muito maior para os usuários.

Carteiras

Carteiras ou *wallets* são *softwares* ou arquivos - geralmente, refere-se a *softwares* - que contém chaves privadas e/ou disponibilizam funcionalidades para administração destas chaves, como: *backup*, envio de transações, monitoramento de recebimento de transações, *UTXOs*, geração de endereços, etc. As carteiras podem guardar outras informações sobre as transações para melhor usabilidade como anotações sobre as transações, e outras utilidades como assinatura de textos ou arquivos com suas chaves privadas.

O *software* de carteira mais simples costuma fazer, pelo menos, as operações: gerar chaves privadas, derivar chaves públicas correspondentes, monitorar *UTXOs* para para estas chave, e criar e assinar transações. Existem outras carteiras com menos ou mais funcionalidades de acordo com o caso de uso (exemplo: uma empresa pode utilizar uma carteira com o código enxuto para ter uma menor superfície de contato em um computador desconectado da Internet com a única funcionalidade de proteger as chaves privadas e assinar transações para transmitir estas transações por outra máquina seguindo um protocolo de segurança mais rigoroso). No entanto, a diferença fundamental nas carteiras mais utilizadas está no esquema de geração de chaves que elas utilizam. Abaixo podemos ver os tipos de carteiras mais comuns e um pouco do contexto técnico que justifica a utilidade de cada tipo. Vale notar que os tipos listados abaixo não são necessariamente exclusivos entre si.

Tipos de Carteiras: Método de Segurança das Chaves

Hot Wallets: são todas as carteiras que, em algum momento, tem as suas chaves privadas expostas em um ambiente com conexão disponível à Internet - especialmente, a Internet. Estas são, provavelmente, são as carteiras mais usadas e que apresentam o menor nível de segurança para o usuário. Pelo menos, toda vez em que uma nova transação precisa ser assinada, o usuário digita uma senha ou PIN para desencriptar a chave privada e assinar a transação em um ambiente com conexão à Internet, fazendo com que a chave fique exposta mesmo que por alguns momentos.

Cold Wallets: são todas as carteiras rodando em um ambiente sem conexão à Internet. Outro nome comum para definir este esquema é *Cold Storage* e a chave privada NUNCA é exposta em um computador com acesso à internet. Normalmente são criadas e usadas para acumular bitcoins e, quando usadas, descartadas e trocadas por novas chaves com menos chance de terem sido comprometidas. Também são usadas como carteira em ambiente totalmente *offline* apenas para assinar transações e ter as transações levadas de alguma outra forma ao conhecimento da rede bitcoin. O ideal para a segurança é que as chaves sejam sempre renovadas quando uma transação for feita.

Paper Wallets: podem ser consideradas um tipo de *cold storage*, já que são carteiras com as chaves privadas impressas em papel para proteção física da chave privada. A efetividade da segurança deste método depende de como as chaves foram geradas, em que ambiente elas foram geradas (ex.: um computador sem conexão) e como elas são armazenadas com segurança contra roubo ou destruição. Um método interessante para este tipo de carteira é o M-de-N que permite que você tenha um número N de papéis e precise apenas de M destes papéis para gastar seus bitcoins. Assim, se você tiver uma 3-de-5 que é um esquema bem comum, você pode ter 5 papéis separados fisicamente em locais seguros e precisará de 3 destas partes para gastar os bitcoins; Desta forma além da proteção das chaves não estarem conectadas, poderá existir a proteção contra um único ponto de falha já que, caso um papel seja destruído ou roubado, os bitcoins ainda estarão seguros para serem enviados para uma nova carteira segura.

Hardware Wallets: também podem ser consideradas um tipo de *cold storage*. O interessante desta carteira é a possibilidade de ter um nível de segurança razoável em comparação aos outros métodos ao mesmo tempo que se pode ter uma carteira boa para ser utilizada com mais frequência no dia-a-dia. Técnicamente, este tipo de carteira se vale de uma separação de *hardware*, sendo um dispositivo seguro, desconectado de qualquer rede e que não confia no computador a que ela é conectada. Tudo que ela faz é assinar transações com as chaves privadas armazenadas nela e verificar a validade de endereços de recebimento para proteção contra certos tipos de ataque *man-in-the-middle*. Normalmente, os passos para assinatura de uma transação são como seguem: O usuário, com a carteira conectada ao computador, cria uma transação no *software* de carteira, o *software* compatível com a *hardware wallet* envia esta transação para ser assinada pela *hardware wallet*, o usuário confirma a transação na própria *hardware wallet* em um visor separado (ou por algum outro esquema como cartões PIN) e confirma, logo em seguida tudo que sai da *hardware wallet* é a transação assinada e nada mais para que o *software* de carteira possa transmitir esta transação à rede.

Tipos de Carteira: Método de Geração de Chaves

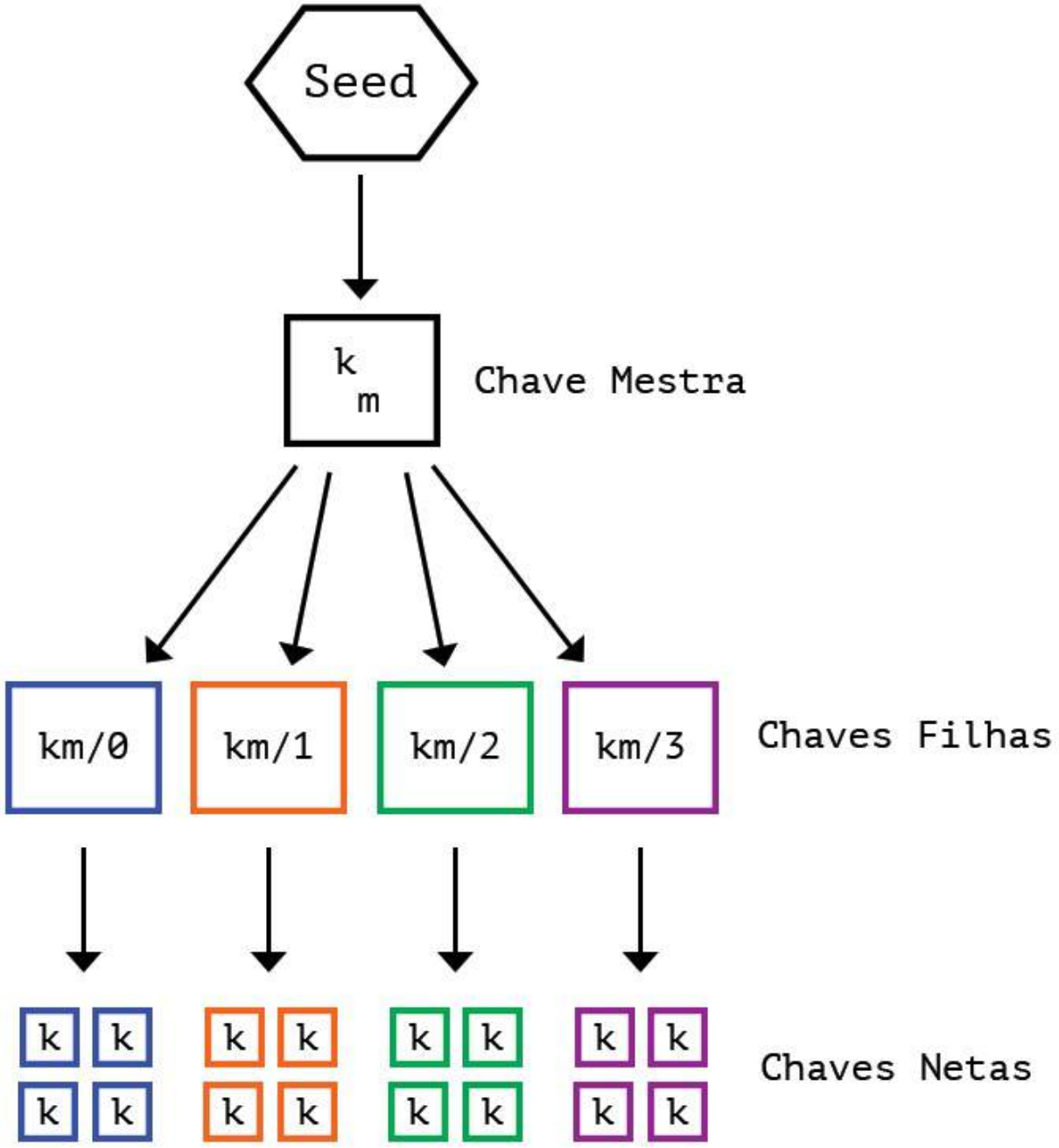
Single-Address Wallets: são carteiras que utilizam um único endereço para recebimento de transações e trocos, envio, etc. Não são um bom tipo de carteira para privacidade e segurança, e são cada vez menos utilizadas substituídas por métodos superiores para a maioria dos casos de uso.

Nondeterministic (Random) Address Wallets: carteiras não-determinísticas tem um armazenamento de tamanho fixo de endereços gerados aleatoriamente. As chaves privadas são geradas de forma aleatória, tem seus endereços correspondentes gerados e ficam armazenadas. Este esquema tem alguns problemas para a segurança dos bitcoins caso *backups* não sejam feitos regularmente. Para entender o problema, digamos que você

tenha uma *pool* de no máximo 100 endereços na sua carteira, você faz um *backup* da carteira antes de começar a utilizar ela, então, depois de um tempo utilizando com centenas ou milhares de transações, você tem o seu disco rígido destruído perdendo o acesso a carteira. Ótimo, você pega o seu *backup* e restaura em um novo computador... mas, de repente, você nota que está sem parte ou todos os seus bitcoins e, sim, você os perdeu. O que aconteceu é que quando você fez o backup da primeira vez, o seu backup tinha guardado as 100 primeiras chaves privadas da *pool* da sua carteira, mas, com o tempo, você foi utilizando e criando novos endereços, recebendo em outros e ec, e estas chaves estavam sendo armazenadas em seu computador danificado, mas não no *backup*. Neste tipo de carteira, você gera as chaves de forma aleatória e as armazena, você não tem como saber quais chaves foram geradas no computador caso não tenha armazenado as novas chaves no *backup*.

Deterministic Address Wallets: carteiras determinísticas geram chaves privadas derivadas a partir de uma *seed* (semente) em comum. Para a recuperação de seus bitcoin neste tipo de carteira, você só precisa estar em posse de sua *seed* inicialmente gerada. Neste tipo de carteira, diferente das não-determinísticas, há a garantia de que todas as chaves privadas serão geradas na mesma ordem não importando o dispositivo em que ela estiver rodando. Isto facilita a segurança e usabilidade já que não é mais necessário repetir o processo de *backup* da carteira original de tempos em tempos.

Hierachical Deterministic (HD) Wallets: carteiras hierárquicas determinísticas como definidas pelas BIP0032 e BIP0044 são um tipo avançado de carteira determinística em que as chaves também são geradas a partir de uma *seed* e todas são geradas em uma ordem não aleatória. A diferença aparece na *feature* que este tipo de carteira apresenta ao gerar suas chaves formando uma estrutura de árvore com cada folha na árvore tendo a possibilidade de gerar as chaves filhas e não as acima delas. Isto possibilita uma organização estrutural superior da carteira com facilidades de organização, divisão de carteiras e, inclusive, a possibilidade de ter uma carteira dividida por suas folhas por pessoas de uma empresa de acordo com a estrutura organizacional sem comprometer as chaves privadas mestras. Para um melhor entendimento, veja esta imagem mostrando da *seed* até as folhas filhas:



Próximo capítulo: Transações

Transações

Uma das melhores introduções sobre transações vem diretamente do livro "Mastering Bitcoin" que, em tradução livre, diz:

> "Transações são a parte mais importante do sistema bitcoin. Todo o resto é desenhado para garantir que as transações possam ser criadas, propagadas na rede, validadas, e finalmente adicionadas ao livro-razão de transações (a blockchain). Transações são estruturas de dados que codificam a transferência de valor entre participantes do sistema bitcoin. Cada transação é uma entrada pública na blockchain do bitcoin, o livro-razão de dupla entrada global."

Ciclo de Vida de uma Transação

O ciclo de vida de uma transação é todo o processo desde a criação de uma transação até a inclusão dela na blockchain por um minerador. De forma resumida, uma transação bem-sucedida é criada, assinada com a chave privada correspondente para "destrancar" os *outputs* da transação anterior referenciados nos *inputs* da transação atual, enviada à rede, verificada e validada pelos nós na rede P2P Bitcoin, e propagada por eles até chegar ao minerador que incluirá a transação na blockchain (primeira confirmaçao).

Para mais detalhes sobre cada parte do ciclo de vida de uma transação do tipo mais comum, recomendo que leia O Ciclo de Vida de uma Transação Bitcoin (arquivado aqui caso não esteja disponível na URL original).

A melhor forma de entender como as transações funcionam na rede Bitcoin é conhecendo e "hackeando" cada peça desta estrutura. Comecemos por conhecer estas peças...

Estrutura de uma Transação

Uma transação é a estrutura de dados responsável por formalizar em código a transferência de valor de um ou mais *inputs* (fonte dos fundos) para um ou mais *outputs* (destino dos fundos). Aqui está o primeiro nível e uma transação:

Campo	Descrição	Tamanho
version	Identifica as regras que a transação segue	4 *bytes*
tx_in count	Identifica quantos *inputs* a transação tem	1-9 *bytes*
tx_in	O(s) *input(s)* da transação	Tamanho variável
tx_out count	Identifica quantos *outputs* a transação tem	1-9 *bytes*
tx_out	O(s) *output(s)* da transação	Tamanho variável
lock_time	Um *timestamp UNIX* ou um número de bloco a partir de quando/qual a transação poderá ser destrancada	4 *bytes*

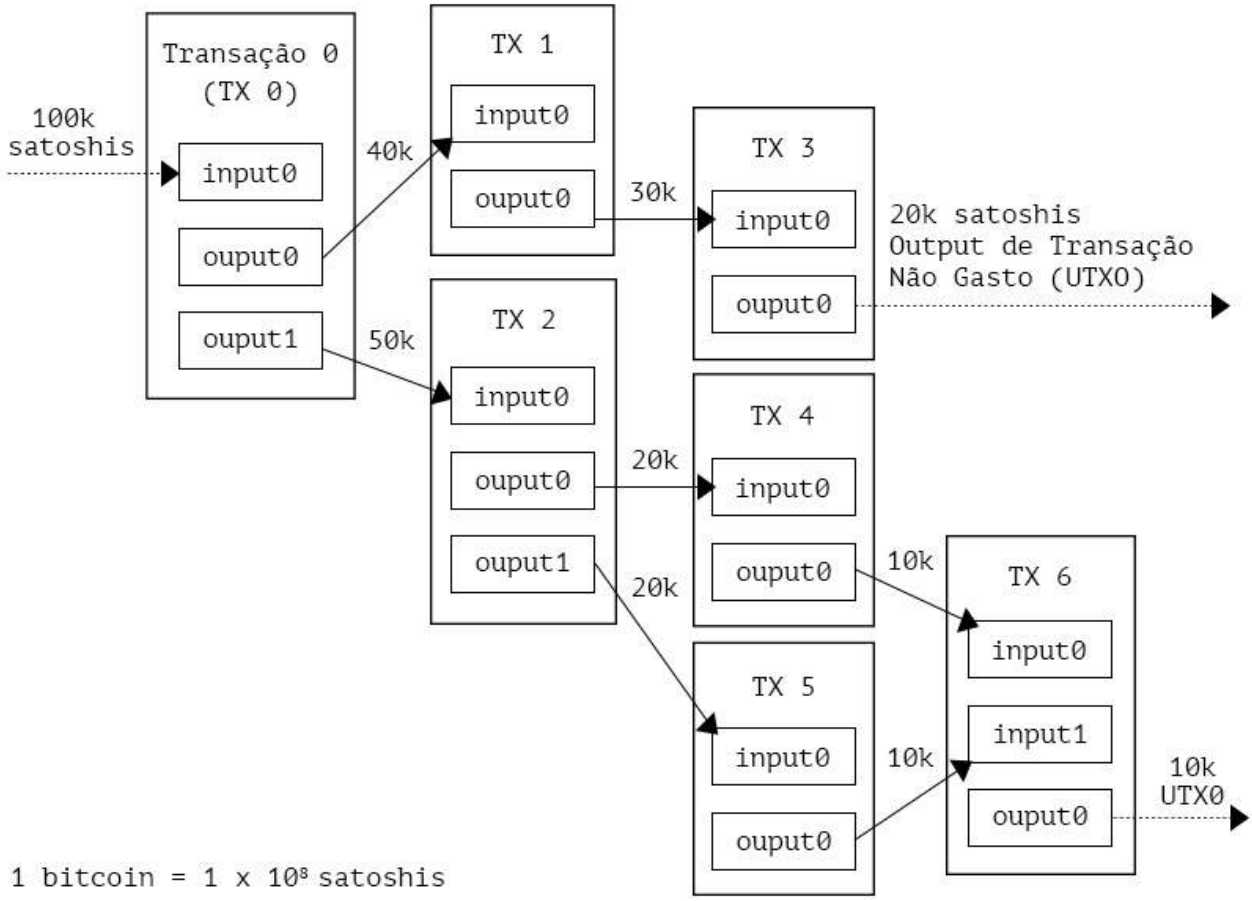
Para que o nosso entendimento do papel de cada um destes campos fique bem claro: O campo `version` diz a versão desta transação para que todos nós na rede saberem que regras ela segue para que possam decidir como verificá-la ou, até, descartá-la em caso de incompatibilidade; os `tx_in count` e `tx_out count` servem como contadores dos *inputs* e *outputs*, respectivamente, para que a iteração por ambos seja simples ao se saber o número de elementos a se esperar; os `tx_in` e `tx_out` tem um ou mais *inputs* e *outputs* de transação, respectivamente em sua estrutura; E, finalmente, o `lock_time` é o tempo em formato *UNIX* ou número de bloco a partir do qual a transação será considerada válida pela rede - normalmente, este valor é `0` para indicar que a transação pode ser gasta assim que recebida e validada.

Outputs e *Inputs*

Os *inputs* e *outputs* são os elementos fundamentais de uma transação. As transações são ligadas umas às outras por estes dois elementos; Os *inputs* de uma transação são, simplemente, referências aos *outputs* de uma transação anterior. Estes *outputs* prontos para serem usado por uma nova transação são chamados de *UTXO* (*unspent transaction output*/*output* de transação não gasto).

A quantia de bitcoins que você tem são apenas uma abstração dos *UTXO* que você tem atribuídos a endereços sob o seu controle. Quando alguém diz que "tem 15 bitcoins" significa que esta pessoa tem uma quantidade de *UTXO* que atribuem, em seu total, 15 bitcoins para serem destrancados por endereços cujas chaves privadas estão sob controle dela. O sistema não tem um contador que diz algo como "15 bitcoins em sua conta"; ele apenas conhece *UTXOs* que são, por sua vez, abstraídos para valores como "15 bitcoins".

O modo como as transações funcionam, faz com que elas formem uma sucessiva corrente de *inputs* e *outputs* trancando e destrancando valores na rede.



Tendo como única exceção as chamadas transações *coinbase* que são as transações criadas pelos mineradores ao incluirem um novo bloco na blockchain (mais detalhes em Mineração) e recolherem o prêmio pelo trabalho.

Como já dito, é importante lembrar que os *UTXO* quando usados numa nova transação sempre devem ser gastos **completamente**, o que faz que seja comum ter transações com um ou mais endereços de troco da transação já que nem sempre você terá *UTXOs* disponíveis formando o valor exato que deseja enviar para alguma transferência de valor.

Outputs

Todas transações criam um ou mais *outputs* para serem destrancados posteriormente quando usados em outra transação e a maioria deles podem ser gastos. Agora, nosso entendimento sobre transações fica mais interessante e preciso ao dissecarmos a estrutura do *output* para entendermos cada uma de suas peças:

Campo	Descrição	Tamanho
value	Número de *satoshis* (BTC/10^8) a serem transferidos	8 *bytes*
locking-script length	O tamanho do *locking script* em *bytes*	1-9 *bytes*
locking-script	Um *script* com as condições necessárias para o *output* ser gasto	Tamanho variável

"Mas o que é um locking-script?"

Os *locking-scripts* são uma peça fundamental para que você possa contemplar como as transações Bitcoin realmente funcionam. Eles são escritos na linguagem *Script* do Bitcoin (um pouco mais sobre a linguagem adiante) e ditam a condição necessária para que aquele *output* possa ser gasto; é como um desafio que precisa de uma solução para que o *output* seja destrancado. As transações mais comuns são transações em que se transfere um certo número de bitcoins que poderão ser gastos por quem provar que tem o controle sob a chave privada de algum outro endereço, mas as condições podem ser bastante variadas permitindo, até mesmo, a criação de *outputs* provadamente impossíveis de serem gastos; o que atende a certos casos de uso como veremos adiante.

Inputs

Os *inputs* são referências aos *UTXOs* (*outputs* não gastos) contendo a resposta à condição necessária para gastar os *UTXOs*. A estrutura de um *input* de transação é:

Campo	Descrição	Tamanho
tx Hash	Identificador da transação que contém os *UTXOs* a serem gastos	32 *bytes*
output index	Índice do *UTXO* a da transação a ser gasto	4 *bytes*
unlocking-script length	Tamanho do *unlocking-script* em *bytes*	1-9 *bytes
unlocking-script	O *unlocking-script* que responde as condições do *locking-script* do *UTXO* a ser gasto	Tamanho variável
*sequence number	Sequência para ser usada por *feature* de substituição de transação (atualmente, não utilizado)	4 *bytes*

O *unlocking-script*, como você já deve estar concluindo, é a solução para o desafio colocado no *locking-script* do *output* ao qual o *input* atual fizer referência. Se o *UTXO* sendo gasto dizia *"Apresente assinatura pertencente ao endereço 1exemplo"*, o *unlocking-script* deve responder com a assinatura do endereço "1exemplo" para que a transação seja considerada válida.

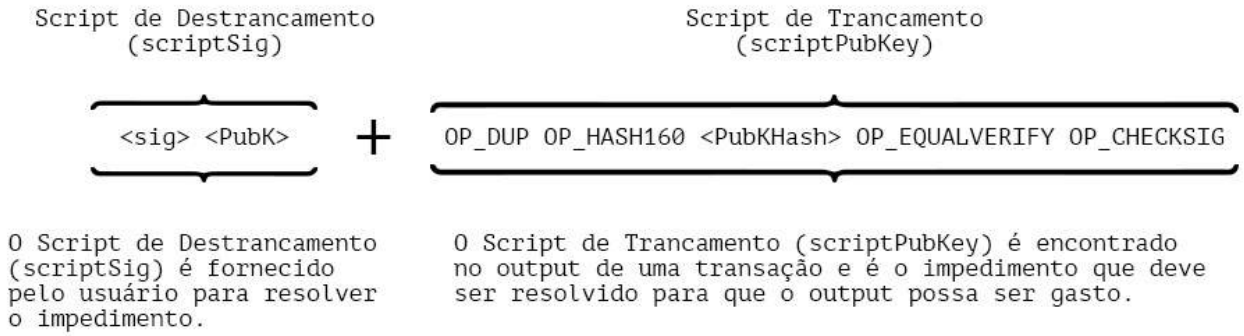
O que nos leva a entender como esta linguagem funciona...

Linguagem *Script* Bitcoin

Como já percebido, a alma da validação de uma transação Bitcoin está nos *locking script* - responsável por trancar os *outputs* - e *unlocking script* - responsável por destrancar os *UTXOs* utilizados num *input*. Mas como isso é lido e interpretado por cada nó?

A resposta está na linguagem *Script* do Bitcoin. Chamada apenas de *Script* é uma linguagem similar à Forth com notação em Polonesa Reversa/Inversa com método de execução em pilha interpretada da esquerda para a direita e foi intencionalmente limitada - por exemplo, ela é *Turing* incompleta - para evitar problemas inesperados na rede e garantir a robustez das transações. A explicação ficará simples quando visualizada.

Primeiro, vejamos como ambos *scripts* ficam organizados para serem interpretados na validação de uma transação. Cada nó responsável por validar uma transação utiliza o `scriptPubKey` do *UTXO* referenciado no *input* atual e utiliza o `scriptSig` deste *input* para formar a expressão a ser verificada. Aqui está um exemplo do tipo mais comum de transacão - a *Pay to Pubkey Hash* (*P2PKH*) - em que "enviamos" um certo número de bitcoins para ser gasto pelo detentor da chave privada de outro endereço:



Mas antes de resolvermos esta expressão, vamos visualizar a resolução de uma expressão com operações matemáticas básicas para que entendamos como os *scripts* são interpretados e validados na rede Bitcoin. Tomemos como exemplo este *locking script* contendo apenas operações aritméticas:

```
7 OP_ADD 3 OP_SUB 6 OP_EQUAL
```

Contemple o fato de que este *locking script* é sintaticamente correto e poderia ser o *script* trancando um *output* caso alguém quisesse uma "doação" ao primeiro que quisesse destrancar este *output*. As palavras com prefixo "OP_" são chamados de **opcodes** e são funções da linguagem de *script* do Bitcoin utilizadas nas operações; Uma lista de **opcodes** existentes pode ser encontrada aqui.

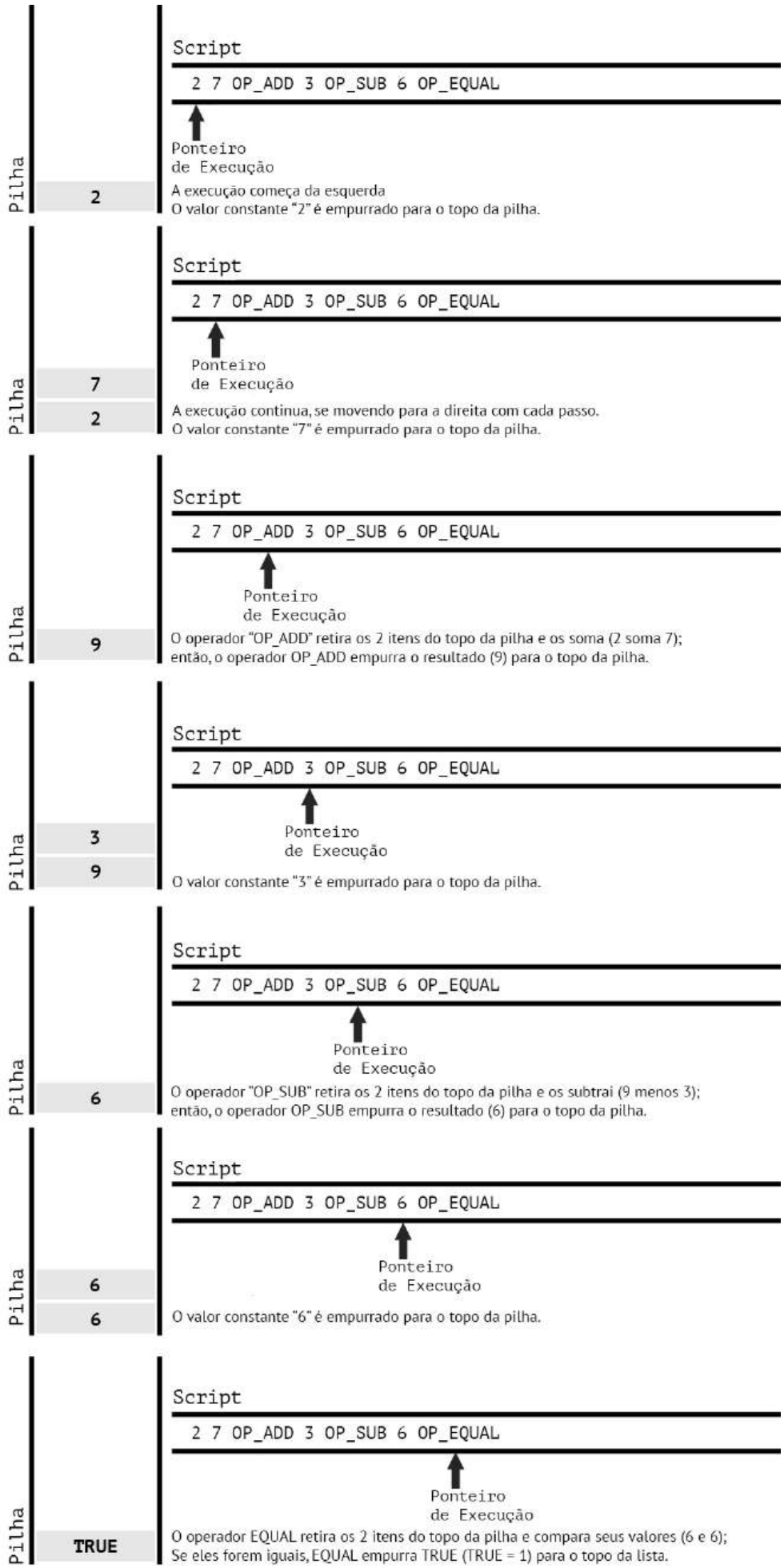
A solução para este *script* pode ser dada apenas com:

```
2
```

A explicação para isso é que o programa de validação pega o *locking script* e o *unlocking script* e os coloca juntos numa expressão assim:

```
2 7 OP_ADD 3 OP_SUB 6 OP_EQUAL
```

E a execução desta expressão acontece como na imagem a seguir:


Pilha
2
Script
2 7 OP_ADD 3 OP_SUB 6 OP_EQUAL
Ponteiro de Execução
A execução começa da esquerda
O valor constante "2" é empurrado para o topo da pilha.
Pilha
7
2
Script
2 7 OP_ADD 3 OP_SUB 6 OP_EQUAL
Ponteiro de Execução
A execução continua, se movendo para a direita com cada passo.
O valor constante "7" é empurrado para o topo da pilha.
Pilha
9
Script
2 7 OP_ADD 3 OP_SUB 6 OP_EQUAL
Ponteiro de Execução
O operador "OP_ADD" retira os 2 itens do topo da pilha e os soma (2 soma 7);
então, o operador OP_ADD empurra o resultado (9) para o topo da pilha.
Pilha
3
9
Script
2 7 OP_ADD 3 OP_SUB 6 OP_EQUAL
Ponteiro de Execução
O valor constante "3" é empurrado para o topo da pilha.
Pilha
6
Script
2 7 OP_ADD 3 OP_SUB 6 OP_EQUAL
Ponteiro de Execução
O operador "OP_SUB" retira os 2 itens do topo da pilha e os subtrai (9 menos 3);
então, o operador OP_SUB empurra o resultado (6) para o topo da pilha.
Pilha
6
6
Script
2 7 OP_ADD 3 OP_SUB 6 OP_EQUAL
Ponteiro de Execução
O valor constante "6" é empurrado para o topo da pilha.
Pilha
TRUE
Script
2 7 OP_ADD 3 OP_SUB 6 OP_EQUAL
Ponteiro de Execução
O operador EQUAL retira os 2 itens do topo da pilha e compara seus valores (6 e 6);
Se eles forem iguais, EQUAL empurra TRUE (TRUE = 1) para o topo da lista.

Este *unlocking script* satisfaz o "desafio" proposto pelo *locking script* ao retornar TRUE que é representado pelo número 1 em hexadecimal `0x01`. Caso a solução fosse inválida, `OP_EQUAL` retornaria FALSE ou `0x00` para a pilha.

Agora, entendendo a forma de execução podemos ver o que cada *opcode* na expressão inicial muito comum em transações Bitcoin faz com cada valor anterior.

`OP_DUP`: Duplica o valor no topo da pilha, ou seja, se `x` estiver no topo `xx` será retornado. `OP_HASH160`: Aplica duas funções *hash* no *input*. Primeiro um SHA-256 e depois um RIPEMD-160, o que a partir de uma chave pública formará o endereço Bitcoin correspondente como visto em Endereços e Carteiras. `OP_EQUALVERIFY`: Aplica `OP_EQUAL` e `OP_VERIFY` logo em seguida, o que significa que checa se os *inputs* são iguais (e retorna TRUE se forem) e, então, marca o *script* como inválido se o valor no topo da pilha for FALSE e nada se for TRUE. `OP_CHECKSIG`: Aplica uma função *hash* sobre os *outputs*, *inputs* e *script* da transação e verifica se a assinatura realmente pertence à chave esperada.

A execução do *script* inicialmente proposto retornará TRUE se a assinatura corresponder com a esperada e fará com que a transação seja considerada válida se todo o resto estiver certo - como possíveis tentativas de *double spend* ou valores errados.

Outros Scripts Comuns

Para finalizar, agora que entendemos que estes *scripts* de trancamento e destrancamento de transações são tão variados quanto a possibilidade de combinação dos *opcodes* que couberem numa transação, não podemos deixar de ver a notação de alguns dos outros *scripts* mais comuns na rede.

Multi-assinatura (*multisig*): Este *script* é utilizado para criar uma conidição de destrancamento da transação na qual um número M de assinaturas devem ser apresentadas de N assinaturas especificadas no *locking-script*. Normalmente, chamado de M de N, um exemplo simples e comum é um esquema de *multisig* 2-de-3 no qual ao menos 2 assinaturas devem ser apresentadas para a transação de 3 chaves previamente especificadas para que os fundos possam ser movidos. Um exemplo de um *script* de trancamento e um de destrancamento para um esquema *multisig* seria como a seguir:

```
# Primeiro o script de trancamento 2-de-3...
2 <Chave Pública A> <Chave Pública B> <Chave Pública C> 3 OP_CHECKMULTISIG
```

```
# E para solucionar o script...
OP_0  <Assinatura A> <Assinatura C>
```

```
# Juntos ficam...
OP_0 <Assinatura A> <Assinatura C> 2 <Chave Pública A> <Chave Pública B> <Chave Públic
a C> 3 OP_CHECKMULTISIG
```

E esta expressão retornará TRUE se as 2 assinaturas corresponderem a 2 assinaturas **diferentes** das 3 especificadas no *script* de trancamento.

OP_RETURN: Um *script* normalmente chamado apenas de `OP_RETURN` devido ao uso deste *opcode* de mesmo nome é utilizado para criar um *output* chamado de *provably unspendable* (ou provadamente não gastável) e normalmente associado com algum dado. Um dos exemplos comuns de uso deste operador é a prova de existência, em que se grava o *hash* de aguma informação para ser adicionado na blockchain e comprovar que tal informação existia, pelo menos, a partir daquele ponto em que foi escrita na blockchain; criando, assim, uma prova matematicamente verificável por qualquer um com acesso à blockchain de que uma informação realmente existia naquele ponto da história em que esta transação foi adicionada à blockchain. Um exemplo deste *script* ficaria assim:

```
OP_RETURN <informação a ser gravada; normalmente um *hash* feito a partir de uma infor
mação (como um arquivo) devido ao limite de 40 *bytes*>
```

Dois exemplos de utilização desta capacidade da linguagem de *script* são os sites Proof of Existence e o brasileiro OriginalMy que geram provas de existência da informação suprida.

Pay to Script Hash (P2SH): Este tipo de *script* facilitou muito a criação de condições mais complexas para a liberação de *outputs* e, inclusive, substitui muitos esquemas de *multisig* devido à sua facilidade para envio de pagamentos para este tipo de esquema de assinatura. Para entendermos, vamos pegar um exemplo de *script* apresentado em *multisig*:

```
2 <Chave Pública A> <Chave Pública B> <Chave Pública C> 3 OP_CHECKMULTISIG
```

O problema que se encontra com este tipo de esquema é que, além do espaço necessário para adicionar um certo número de chaves, como pode notar, qualquer pessoas que esteja criando o pagamento teria que criar uma transação que gerasse um *output* desta forma. Isso, em larga escala, especialmente quando lidando com usuários comuns na rede é inviável visto que a maioria dos *softwares* de carteira nem mesmo geram este tipo de pagamento. A solução para isso introduzida em 2012 com *P2SH* é criar um *hash* a partir do *script* que desejamos usar como condição de destrancamento dos fundos e quando quisermos gastar estes fundos deveremos apresentar o *script* que formou o *hash* junto com a condição de destrancamento para ele. Logo, um *script* de trancamento como o de cima seria reduzido para o resultado do *hash* SHA-256 e, logo após, a aplicação do *hash* RIPEMD-160 por cima deste, fazendo com que possamos criar uma condição assim:

```
OP_HASH160 ab8648c91206a3770b8aaf6f5153b6b35423caf0 OP_EQUAL
```

O que garantiria que o *script* de destrancamento seria verificado e bateria com o *hash* esperado.

Isso resolve alguns problemas, mas a capacidade mais interessante vem com a *feature* do P2SH codificar o *hash* do *script* como um endereço Bitcoin como pode ser visto na BIP0013 que é chamada de **Pay to Script Hash Address*. Com isso, após o trabalho de criação do *hash* do script, a única coisa que precisa ser feita é criar um endereço a partir deste *script* aplicando a codificação *Base58Check* utilizada para endereços Bitcoin resultando em um endereço Bitcoin normal com o prefixo legível 3 como visto em Endereços e Carteiras sobre o significado de prefixos de endereços.

Finalmente, a partir de agora, qualquer transação enviada para este endereço com prefixo 3 respeitará as regras de destrancamento ditadas pelo *script* utilizado para formar este endereço. Resolvendo, entre outros problemas descritos acima, a questão de facilidade de pagamento já que ninguém que envia a transação para este endereço precisa conhecer o *script* que o formou.

Próximo capítulo: Rede P2P

Rede P2P

A rede Bitcoin tem uma arquitetura *peer-to-peer* onde todos os participantes da rede são nós (ou *nodes*) que funcionam tanto como servidores quanto como clientes sem hierarquia especial entre um nó ou outro. Todos são considerados hierarquicamente iguais e devem servir a rede com certos recursos. Este modelo aberto e descentralizado faz com que a rede não tenha uma barreira de entrada arbitrariamente escolhida; basta que você esteja rodando um nó e você será parte da rede.

Os nós na rede trocam mensagens contendo transações, blocos e endereços (IP) de outros *peers* entre si. Ao conectar à rede seu *client* realiza o processo chamado *bootstrap* para achar e se conectar a outros *peers* na rede, e começa a baixar blocos para a cópia local da blockchain de *peers* mais adiantados na rede. Ao mesmo tempo o seu *client* já pode começar a prover dados para os *peers* conectados e ajudar a rede enquanto se atualiza com o consenso da rede. Quando você cria uma transação e a envia para a rede - como fizemos em Bitcoin Core: API JSON-RPC - o seu *client* envia esta transação para alguns *peers*, estes *peers* enviam para outros e assim, sucessivamente, até que um minerador valide a sua transação e a inclua num bloco, e, então, transmita este bloco aos *peers* de forma sucessiva até que seu *client* receba o novo bloco e confirme que a transação foi confirmada.

Quando nos referimos à "rede Bitcoin", estamos falando do protocolo P2P Bitcoin. No entanto, também existem outros protocolos como o *Stratum*, por exemplo, que é usado por mineradores e carteiras *lightweight*, e forma parte do que chamamos de rede estendida Bitcoin.

Tipos de Nós na Rede

Existem 4 características/funções básicas que um nó pode ter na rede: carteira, minerador, blockchain completa e roteamento na rede P2P. Nem todos os nós apresentam todas as quatro características/funções, podendo apresentar desde apenas uma - rede - ou mais delas até todas elas. A falta de hierarquia especial na rede continua a mesma; a única coisa que muda é a funcionalidade que o operador do nó **escolhe** ter.

Os *softwares* dos nós podem ser modificados nas funcionalidades que apresentam e de que forma as implementam de acordo com a necessidade do operador. Por exemplo, uma *exchange* - e a maioria das pessoas - não precisa ter a funcionalidade de mineração em seu nó. O mínimo que se precisa para ser considerado um nó na rede Bitcoin é a

participação com a funcionalidade de rede passando e recebendo mensagens, porém, a rede estendida Bitcoin apresenta outros tipos e nó fora desta rede original com outros protocolos e eles estarão listados abaixo. Aqui está uma lista dos tipos mais comuns:

Tipo	Funções
Cliente Referência (Bitcoin Core)	**carteira**, **minerador**, cópia da **blockchain completa** e roteamento na **rede** P2P.
Nó de Blockchain Completa	cópia da **blockchain completa**, e roteamento na **rede** P2P.
Minerador Solo	**minerador**, cópia da **blockchain completa** e roteamento na **rede** P2P.
Carteira Leve	**carteira** e roteamento na rede.
Servidores Stratum	roteamento na **rede** P2P e funcionam como *gateways* para nós rodando *Stratum*.
Minerador Leve	**minerador** e **rede** conectada a algum nó ou *pool Stratum*.
Carteira Leve Stratum	**carteira** e **rede** conectada a algum nó via protocolo *Stratum*.

> *Full-node* se refere aos nós na rede P2P Bitcoin que mantém uma cópia completa da *blockchain* e, assim, tem independência completa para verificar blocos e transações.

Conectando-se à Rede

O Bitcoin Core - e a maioria dos *clients* seguindo a referência - costumam seguir o mesmo processo de conexão à rede P2P Bitcoin. No caso do Core, assim que você começa a rodar com o comando `bitcoind` este processo é iniciado e passa por verificação dos últimos blocos - 288 por *default* configurável pela *flag* `-checkblocks` -, descoberta de *peers* para que a troca de inventório possa começar a ser feita junto com o roteamento de mensagens (transações, blocos, **peers**...).

Acho uma boa ideia que você veja os *logs* referentes a este processo acontecendo para captar mais informações por você mesmo. Para isso, basta seguir os arquivo `debug.log` no diretório que seu *client* estiver utilizando. Você pode simplesmente usar o comando tail:

```
$ tail -f ~/.bitcoin/debug.log
```

Agora vamos aos passos-a-passo do processo de conexão...

Descoberta de Peers: O primeiro método utilizado na primeira inicialização de seu Bitcoin Core para achar outros *peers* na rede é puxar DNS's usando as chamadas *DNS seeds* que são servidores DNS que disponibilizam uma rede de IPs de outros nós na rede. Esta lista de *DNS seeds* é escrita diretamente no código e, no momento, pode ser encontrada no arquivo chainparams.cpp (linha 112) do código fonte do Bitcoin Core. Como teste, podemos usar o comando `nslookup` no terminal para ver a lista que um destes servidores retornará:

```
$ nslookup seed.bitcoin.sipa.be
Server:         10.137.4.1
Address:    10.137.4.1#53

Non-authoritative answer:
Name:    seed.bitcoin.sipa.be
Address: 88.198.60.110
Name:    seed.bitcoin.sipa.be
Address: 73.71.114.219
Name:    seed.bitcoin.sipa.be
Address: 213.91.211.17
Name:    seed.bitcoin.sipa.be
Address: 85.218.136.63
Name:    seed.bitcoin.sipa.be
Address: 188.226.188.160
Name:    seed.bitcoin.sipa.be
Address: 185.25.49.184
Name:    seed.bitcoin.sipa.be
Address: 204.68.122.11
#[... Mais outros IPs... ]
```

> O Bitcoin Core, atualmente, vem com 6 *DNS seeds* escritas diretamente no código e você pode escolher se utilizará estes servidores com a opção `-dnsseed` que, por padrão, vem configurada com o valor `1`.

Você também pode escolher passar a informação `-dnsseed` definindo o IP do *peer* para que o *nslookup* seja feito ou usar diretamente a opção `-addnode` para se conectar diretamente a um outro nó que esteja disponível na rede.

No entanto, caso o seu *client* já tenha se conectado à rede antes, ele usará como primeiro método a tentativa de se conectar à lista de nós que ele estava conectado antes de sair da rede.

Handshake: Ao se conectar a um nó na rede, a primeira coisa que ambos devem fazer é um *handshake* para se certificarem de que eles conseguem se comunicar de acordo com as mesmas regras. O primeiro nó transmitirá uma mensagem de versão contendo:

Campo	Descrição
PROTOCOL_VERSION	constante que define a versão do protocolo P2P que o *client* utiliza para se comunicar.
nLocalServices	lista de serviços suportados pelo nó.
nTime	hora atual
addrYou	endereço de IP do outro nó como visto por ele.
addrMe	endereço IP do nó local.
subver	Uma sub-versão que identifica o tipo de *software* que o nó está rodando.
BestHeight	A altura do bloco mais recente da blockchain deste nó.

O outro nó, então, continuará com o *handshake* retornando uma mensagem *verack* em resposta ao recebimento de versão junto com a própria mensagem de versão dele em caso de ter aceitado continua a conexão, reconhecendo que ambos estão "falando" de formas compatíveis.

O primeiro nó envia, agora, as mensagens *getaddr* e *addr* para pegar endereços de outros *peers* conectados ao segundo nó e começar o mesmo processo de estabelecimento de conexão com estes outros e ter o seu próprio endereço repassado a outros nós conectados ao seu *peer* atual. E, este mesmo processo de conexão ocorrerá com mais outros nós a fim de se estabelecer na rede já que nenhum nó tem a garantia de que permanecerá conectado e o processo de descoberta de novos nós deve continuar periodicamente para garantir que os nós que parem de responder sejam descartados e novos sejam adicionados como *peers*.

Você pode usar a API JSON-RPC para ver quais nós estão conectados ao seu *client* no momento junto com algumas informações sobre eles:

```
$ bitcoin-cli getpeerinfo
[
{
    "id": 2,
    "addr": "54.152.216.47:8333",
    "addrlocal": "179.43.176.98:47524",
    "services": "0000000000000001",
    "relaytxes": true,
    "lastsend": 1463261684,
    "lastrecv": 1463261685,
    "bytessent": 2153665,
    "bytesrecv": 455684694,
    "conntime": 1463258351,
    "timeoffset": 3,
    "pingtime": 9.469450999999999,
    "minping": 0.419712,
    "version": 70002,
```

```
    "subver": "/Satoshi:0.10.0/",
    "inbound": false,
    "startingheight": 411774,
    "banscore": 0,
    "synced_headers": 411777,
    "synced_blocks": 199986,
    "inflight": [
      200190,
      200191,
      200208,
      200214,
      200219,
      200243,
      200253,
      200259,
      200266,
      200271,
      200276,
      200297,
      200298,
      200299,
      200303,
      200305

    ],
    "whitelisted": false

},
{
    "id": 3,
    "addr": "45.32.185.154:8333",
    "addrlocal": "179.43.176.98:50096",
    "services": "0000000000000001",
    "relaytxes": true,
    "lastsend": 1463261684,
    "lastrecv": 1463261685,
    "bytessent": 2485767,
    "bytesrecv": 533974518,
    "conntime": 1463258352,
    "timeoffset": 1,
    "pingtime": 8.236776000000001,
    "minping": 0.260138,
    "version": 70002,
    "subver": "/Satoshi:0.11.2/",
    "inbound": false,
    "startingheight": 411774,
    "banscore": 0,
    "synced_headers": 411777,
    "synced_blocks": 199986,
    "inflight": [
      200222,
      200223,
      200241,
```

```
        200244,
        200260,
        200262,
        200270,
        200272,
        200273,
        200274,
        200275,
        200282,
        200284,
        200288,
        200301,
        200307

      ],
      "whitelisted": false

  },
  # [... Mais outros nós... ]
  ]
```

Troca de Inventário: Agora, para que o nó que acabou de entrar na rede possa ter uma cópia atualizada da blockchain, uma das primeiras coisas que ele faz ao se conectar com outros nós é a troca de inventários. A primeira mensagem enviada ao se conectar ao outro nó inclui o *bestHeight* que informa até que bloco ele tem conhecimento. Assim que ele receber a mensagem de versão de seu *peer* contendo o *bestHeight* dele, ele saberá se está precisando sincronizar a sua cópia local da blockchain com o resto do consenso da rede ou se é seu *peer* que está mais atrás. O *peer* que tiver a blockchain com mais longa (normalmente, a com mais trabalho), já sabendo quantos blocos o *peer* com a blockchain mais curta precisa, identificará os 500 primeiros blocos que o *peer* está precisando para alcançar o resto da rede e enviará uma mensagem *inv* com os *hashes* dos blocos para que, então, o *peer* que precisa se atualizar envie uma mensagem *getdata* pedingo a informação completa de cada bloco para que ele possa baixar os blocos e verificar um por um de forma independente.

Este nó, agora, faz parte da rede P2P Bitcoin e contribui recebendo, verificando e transmitindo transações, blocos e outras informações ao resto da rede.

> Uma dica para caso ainda esteja muitos blocos atrás na rede e queira acelerar a atualização é buscar por nós bem conectados e que estejam contribuindo bastante para a rede em locais como esta lista e iniciar o seu client com a *flag* `-addnode` com o IP de alguns destes nós.

Transmitindo e Propagando Transações

Toda vez que um nó conectado à rede cria uma transação, ele envia uma mensagem *inv* informando a nova transação para todos os seus *peers*. A mensagem *inv* é uma mensagem de inventório que apenas notifica outros nós sobre um novo objeto descoberto ou envia dados sobre algo que está sendo requisitado. O nó que recebe uma mensagem *inv* sabe se tem ou não um objeto porque esta mensagem inclui o *hash* do objeto. Este é o formato de uma mensagem *inv*:

Campo	Tamanho	Descrição
type	4 *bytes*	tipo do objeto identificado neste inventório
hash	32 *bytes*	*hash* do objeto

Os outros nós que receberem a mensagem *inv* e não conhecerem este novo objeto - neste caso, uma nova transação - enviarão uma mensagem *getdata* incluindo o *hash* do objeto pedindo a informação completa. Ao receberem a transação, poderão verificar a validade dela por si próprios e, em caso da transação ser válida, repetirão o mesmo processo com os outros *peers* propagando a transação sucessivamente pela rede.

Pools de Transação

Enquanto estas transações recebidas e validadas pelos nós não são incluidas por algum minerador à blockchain, elas, geralmente, permanecem em um espaço de memória volátil de cada *full-node* da rede. A maioria dos nós conectados à rede implementam a *pool* chamada *mempool* e, alguns outros, implementam *pools* separadas da *mempool* como as *orphan pools* para as chamadas transações orfãs e as *UTXO pools* que pode ser implementada em uma memória local persistente dependendo do tipo do *software* utilizado pelo nó.

mempool: guarda todas as transações recebidas e validadas pelo nó que não tenham sido escritas na blockchain por algum minerador. Quanto mais transações sendo propagadas pela rede, maior a *mempool* ficará até que ela vá sendo gradualmente esvaziada com as transações sendo aceitas pelos mineradores. Muitas implementações de carteiras usam a *mempool* para calcular as taxas de transação ideais para que a transação seja aceita o mais rápido quanto possível, criando uma sugestão de *fee* flutuante.

orphan pool: é uma *pool* separada da *mempool* existente em algumas versões de *full-node* na rede que guardam as transações que referenciem e dependam de uma transação anterior a elas, chamadas de *parent transactions* - transações pai/mãe - que ainda não tenham sido vistas por este nó.

UTXO pool: diferente da *mempool*, é uma *pool* de transações presente em algumas implementações que guarda em memória volátil ou persistente local milhões de entradas de *outputs* de transação criados e nunca gastos com *UTXOs* que podem datar, em alguns

casos, de transações criadas em 2009. E, enquanto a *mempool* e a *orphan pool* apenas contém transações não confirmadas, esta *pool* apenas contém transações confirmadas na rede.

Próximo capítulo: Blockchain

Blockchain

A blockchain - ou block chain - é uma estrutura de dados ordenada composta por blocos de transações criptograficamente ligados por uma referência direta ao bloco anterior, servindo como livro-razão público no Bitcoin. A propriedade mais interessante da blockchain para o consenso de um sistema descentralizado é este elo criptográfico que liga os seus blocos entre si, que é criado pelo esforço computacional do algoritmo de *proof-of-work*. Sendo apenas mais uma estrutura de dados sem propriedades excepcionalmente úteis por si só fora do contexto apropriado; em um sistema descentralizado é onde esta estrutura de dados engenhosa se torna uma peça-chave na manutenção de consenso e segurança. A blockchain foi dada à luz por Satoshi Nakamoto como uma abordagem inteligente a um problema inerente a sistemas descentralizados: confiança.

O primeiro bloco da blockchain do Bitcoin é o bloco 0 chamado de bloco genesis. Este é o bloco minerado por Satoshi Nakamoto que serve como ponto de partida comum a todas as implementações do Bitcoin e é escrito diretamente no código referência para este fim. Ele contém a famosa frase escolhida por Satoshi Nakamoto:

```
The Times 03/Jan/2009 Chancellor on brink of second bailout for banks
```

Esta frase é serve o propósito de ser uma prova da data mínima em que a rede Bitcoin foi iniciada por se tratar da manchete de um jornal e, também, como mensagem clara sobre a motivação para a concepção de uma tecnologia como o Bitcoin para os dias de hoje. Ela pode ser encontrada diretamente no código-fonte do Bitcoin Core no arquivo chainparams.cpp linha 53.

Blocos

Cada bloco é uma estrutura de dados que contém as transações a serem incluidas na blockchain por meio do trabalho dos mineradores na rede.

De uma forma geral, o bloco é composto por um cabeçalho contendo metadados sobre o bloco e uma lista de transações:

Campo	Tamanho	Descrição
Tamanho do Bloco	4 *bytes*	tamanho do bloco (*block size*) em *bytes* a partir deste campo
Cabeçalho do Bloco	80 *bytes*	o cabeçalho do bloco (*block header*) contendo metadados
Contador de Transações	1-9 *bytes*	número de transações neste bloco
Transaçòes	Variável	as transações deste bloco

Como na descrição, o cabeçalho do bloco é responsável por conter os metadados referentes ao bloco e é nele que está a "cola" criptográfica fundamental para a segurança da blockchain. A estrutura do cabeçalho do bloco contém os seguintes campos:

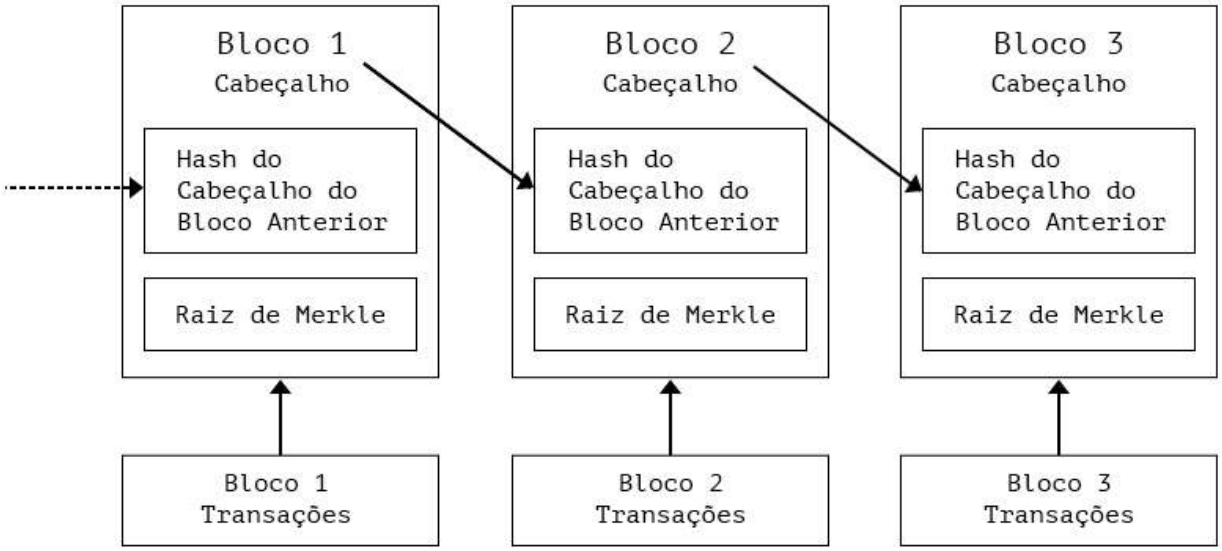
Campo	Tamanho	Descrição
Versão	4 *bytes*	número de versão do bloco indica que regras este bloco segue
Hash do Cabeçalho do Bloco Anterior	32 *bytes*	*hash* do cabeçalho do bloco anterior (*parent block*) a este na blockchain
Raiz de Merkle	32 *bytes*	*hash* da raiz de Merkle das transações deste bloco
Timestamp	4 *bytes*	hora aproximada da criação deste tempo em segundos no padrão UNIX
Dificuldade Alvo	4 *bytes*	dificuldade alvo do algoritmo de *proof-of-work* para este bloco
Nonce	4 *bytes*	contador utilizado como *nonce* no algoritmo de *proof-of-work*

Como pode ver, os carregam todas as informações necessárias para servirem como páginas de um livro-razão das transações confirmadas na rede. Porém, sem o processo de mineração, os blocos e a blockchain estão "mortos". A partir da mineração a vida e utilidade desta estrutura ganha sentido ao se encaixar perfeitamente à esta corrida de processamento (e um pouco de sorte) acontecendo neste exato instante entre milhares de computadores ao redor do mundo. Alguns itens diretamente ligados ao processo de mineração - dificuldade, *nonce* e *timestamp* - serão revistos no capítulo sobre mineração; No momento, estamos interessados em ver os elementos que compõem blockchain para que o processo de atualizaçã dela possa fazer sentido.

O Elo entre os Blocos

Como dito, no cabeçalho dos blocos está a chave para a segurança da blockchain proporcionada pelo algoritmo de *proof-of-work*. Este elo é *hash* do cabeçalho do bloco anterior.

O que ocorre é que cada bloco tem um identificador único que é criado a partir do *hash* de seu cabeçalho que serve tanto como identificador único deste objeto na rede quanto como prova de toda informação - incluindo as transações - contida nele. Para esta identificacão, apenas precisamos do cabeçalho de cada bloco já que a Raiz de Merkle (explicada um pouco adiante) serve como prova de todas as transações incluidas no bloco da qual ela faz parte. Esta característica faz com que a blockchain possa ser visualizada como uma corrente de blocos com os *hashes* do bloco anterior como elo criptográfico entre cada bloco:



Blockchain do Bitcoin Simplificada

Na prática, cada novo bloco tem ligação matematicamente comprovada com todos os blocos anteriores a ele, pois cada bloco tem um *hash* como identificador **único** que inclui, em seu resultado, o *hash* do bloco anterior a ele na blockchain. E, junto com o esforço computacional comprovado pelo *proof-of-work* responsável por gerar cada bloco, esta propriedade cria um elo que torna a forjabilidade da blockchain exponencialmente mais difícil a cada novo bloco.

Para visualizarmos a criação do *hash* do cabeçalho de um bloco - comumente chamado apenas de *hash* do bloco ou *block hash* - podemos usar um pequeno *script* Python3 para calcularmos o *hash* do bloco #125552. Para este fim, utilizamos os 6 campos do cabeçalho do bloco descritos acima concatenados em hexadecimal e ordem dos bytes *little-endian* para fazermos um *hash* SHA-256 seguido de outro *hash* SHA-256 com o resultado do anterior (*shasha*):

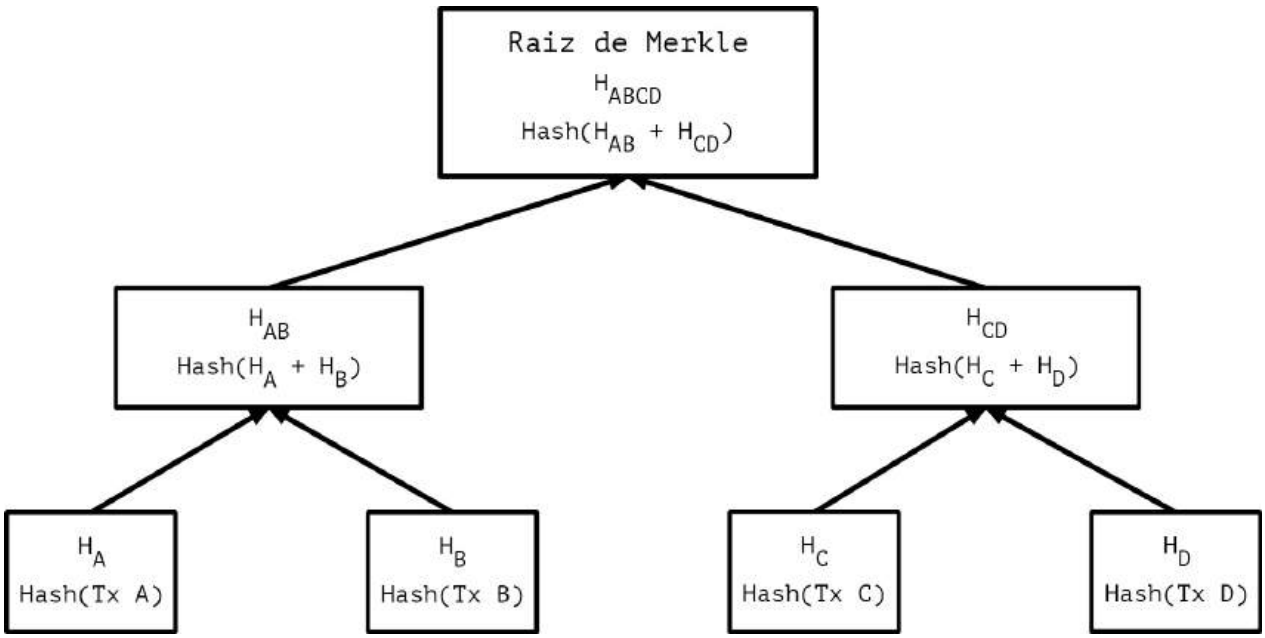
```
>>> import hashlib
>>> import codecs
>>> import binascii

#  blockheader com os 6 campos em hexadecimal concatenados...
>>> blockheader_hex = ("01000000" +  # <- versão
 "81cd02ab7e569e8bcd9317e2fe99f2de44d49ab2b8851ba4a308000000000000" + #  <- hash do bl
oco anterior
 "e320b6c2fffc8d750423db8b1eb942ae710e951ed797f7affc8892b0f1fc122b" +  # <- raiz de Me
rkle
 "c7f5d74d" +  # <- timestamp unix
 "f2b9441a" +  # <- dificuldade alvo em formato compacto
 "42a14695")  # <- nonce

# passando de hex para binário
>>> blockheader_bin = codecs.decode(blockheader_hex.encode('utf-8'), 'hex')
# crio o hash com uma dupla rodada de sha256 (shasha)
>>> blockheader_hash = hashlib.sha256(hashlib.sha256(blockheader_bin).digest()).digest
()[::-1]  # <- trocando a ordem dos bytes para big-endian
# e de volta a hexadecimal com os bytes já na ordem certa
>>> blockheader_hash = binascii.hexlify(blockheader_hash)
>>> print(blockheader_hash)
b'0000000000000001e8d6829a8a21adc5d38d0a473b144b6765798e61f98bd1d'
```

E este é o *hash* do bloco 125552. Note a passagem da ordem dos *bytes* de *little-endian* para *big-endian* que é como as informações são transmitidas na rede e, geralmente, salvas desta forma.

Árvores e Raízes de Merkle

Árvores de Merkle são estruturas de dados utilizadas para criar um resumo de dados com integridade criptograficamente verificável de forma eficiente quando em poder da raiz de Merkle - que vai no cabeçalho de cada bloco - e de um caminho de Merkle.

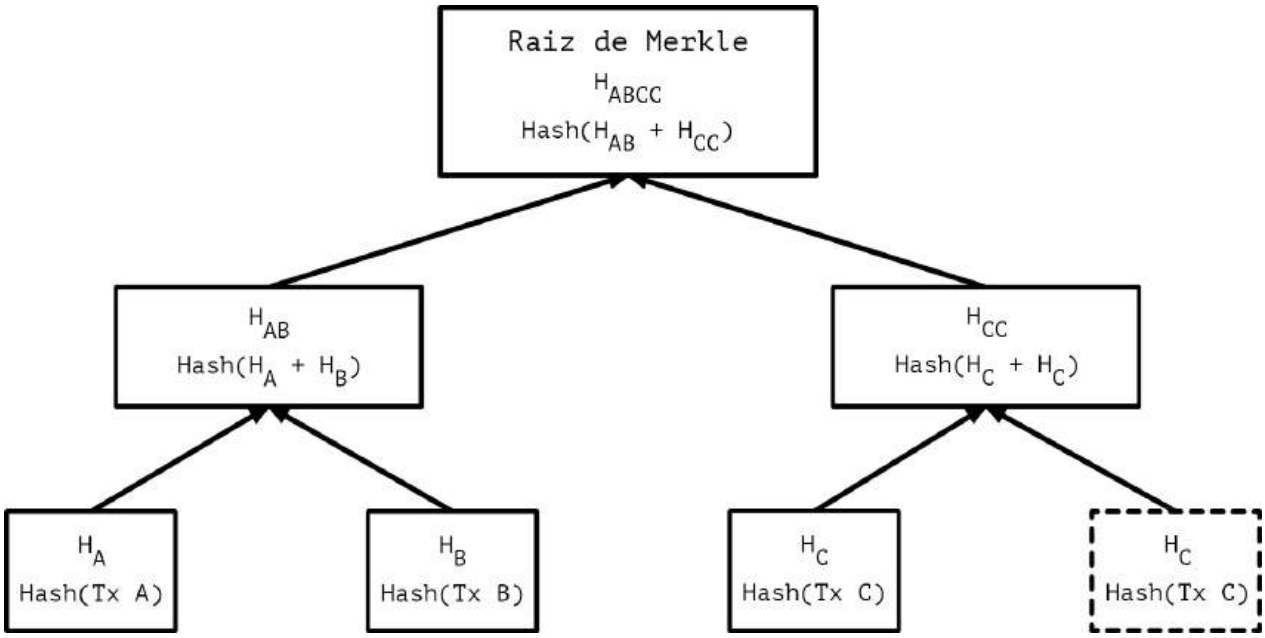
Para formar a raiz desta árvore binária com as transações, cada transação tem o seu *id* (o *hash* da transação) concatenado ao *id* da transação vizinha na árvore é submetida a uma dupla rodada da função *hash* SHA-256 sucessivamente até chegar à raiz. A visualização torna simples:



Cada um dos *hashes* passados é a mesma dupla rodada de SHA-256 já conhecida por nós. Logo, para formar o nó `H AB` e supondo que já temos o *id* das transações atribuídos às variáveis `tx_a` e `tx_b` basta que façamos isso:

```
>>> import hashlib
>>> H_ab = hashlib.sha256(hashlib.sha256(tx_a + tx_b).digest()):w
```

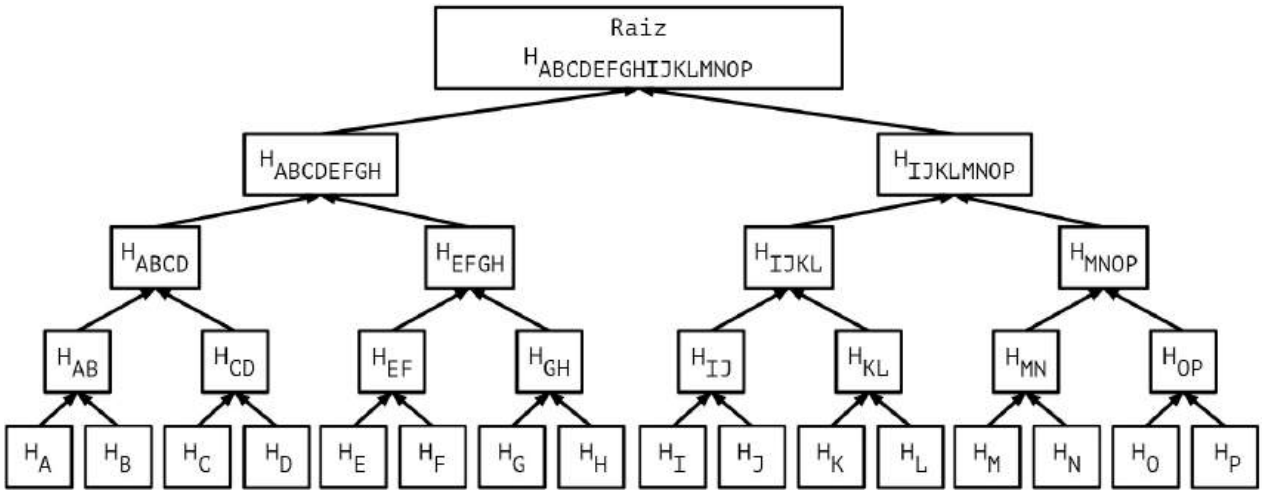
Para isso, todas as transações devem ser colocadas no mesmo nível como na imagem e formarem duplas para criarem os níveis acima sucessivamente. Em caso do bloco não ter um número par de transações, tudo que se faz é repetir a última transação:



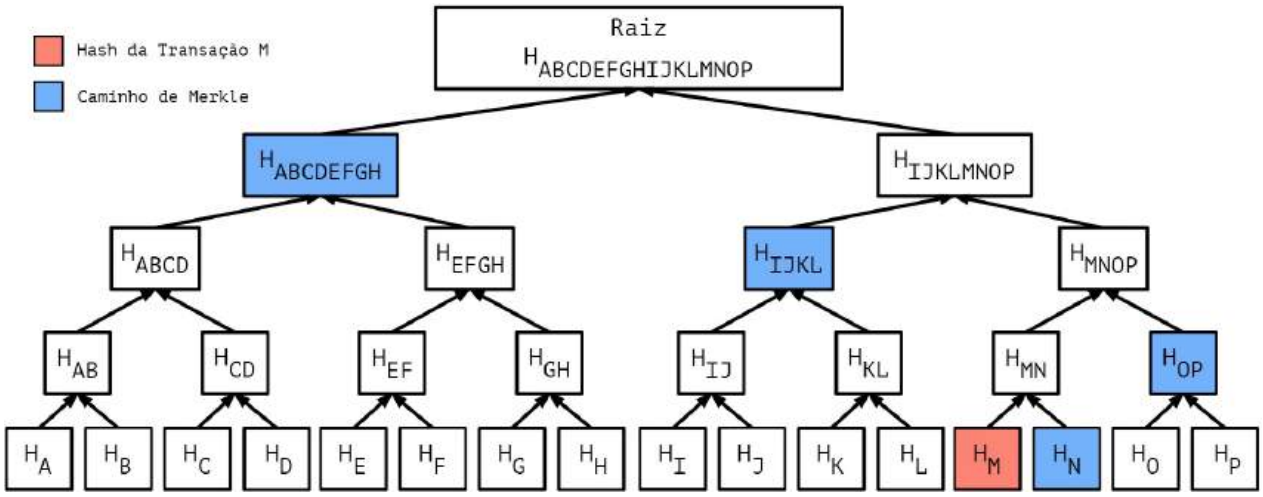
Óbviamente, no Bitcoin, estas árvores são muito maiores e proporcionais ao número de transações de cada bloco e em cenários desta magnitude que esta estrutura se torna uma solução para comprovação eficiente da existência e integridade de uma transação num

dado bloco. Cada *hash* deste tem o tamanho de 32 *bytes* e a complexidade de busca na árvore de Merkle cresce `O(log2(N))` na notação "Big-O" com N sendo o número de transações.

Vejamos um exemplo um pouco maior de uma árvore de Merkle criada a partir de 16 transações:



Caso precisemos comprovarmos que a transação M está incluida no bloco, precisamos de apenas 4 *hashes* de 32 *bytes* num total de 128 *bytes* para formar o noso caminho de Merkle. Veja como o caminho de Merkle junto com a raiz de Merkle é tudo que precisamos para comprovarmos que um bloco inclui a transação M:



A eficiência da árvore de Merkle para este objetivo vai se tornando mais óbvia de acordo com que aumentamos o número de transações e comparamos com o número de *bytes* necessários para comprovar a existência de uma transação nestes números maiores:

Número de Transações	Tamanho Aproximado do Bloco	Tamanho do Caminho de Merkle (*hashes*)	Tamanho do Caminho de Merkle (*bytes*)
16 transações	4 *kilobytes*	4 *hashes*	128 *bytes*
512 transações	128 *kilobytes*	9 *hashes*	288 *bytes*
2048 transações	512 *kilobytes*	11 *hashes*	352 *bytes*
65.525 transações	16 *megabytes*	16 *hashes*	512 *bytes*

O que observamos é que com poucas transações não parece fazer muito sentido o uso da árvore de Merkle, mas logo que o número de transações começa a saltar podemos ver claramente a otimização que esta estrutura traz ao sistema do Bitcoin.

Próximo capítulo: Mineração

Mineração

A mineração é o processo responsável por atualizar a blockchain e, até atingir o limite de cerca de 21 milhões *satoshis*, trazer novas moedas à rede por meio de uma competição de processamento intenso com o intuito de alcançar um *hash* de um bloco com transações válidas menor ou igual ao resultado esperado pelo resto da rede. Esta competição propositalmente pesada para os recursos computacionais produz o *proof-of-work* - ou prova de trabalho - essencial para a segurança do consenso na rede. O algoritmo usado no Bitcoin é o *hashcash* criado em 1997 por Adam Back.

O Propósito da Mineração

Segurança

Em Blockchain podemos ver que cada bloco é diretamente ligado ao anterior pelo campo que contém o *hash* do cabeçalho do bloco anterior, servindo de prova de integridade da blockchain. A mineração, por sua vez, adiciona o toque final à segurança da blockchain trazendo a necessidade de um esforço computacional mínimo para a adição de cada novo bloco e, assim, fazendo com que o esforço de qualquer tentativa de mudança aos blocos da blockchain cresça exponencialmente a cada bloco anterior que se tentar realizar a mudança; ao mesmo tempo em que a rede continua trabalhando adicionando novos blocos com mais *proof-of-work* em cada um deles, fazendo a blockchain cada vez mais segura e praticavelmente imutável. Graças ao trabalho da mineração, se eu quiser alterar um bloco de altura 410500, eu terei que provar para a rede o trabalho computacional de todos os blocos subsequentes a este desde o bloco mais alto aceito pela rede; No momento, isso é simplesmente impossível em meu tempo de vida, pois dependeria de um trabalho computacional que, mesmo com muito processamento disponível, levaria um tempo fora de minha compreensão humana de espaço-tempo.

Novas Moedas

Outro propósito da mineração está em um dos incentivos oferecido aos mineradores como recompensa pelo esforço computacional gasto na segurança da blockchain contra alterações maliciosas. Junto com as taxas de mineração recebidas de todas as transações incluidas no bloco, o minerador também recebe uma recompensa que tem a dupla função de servir de subsídio ao trabalho despendido e de trazer novas moedas à existência na

rede. Este subsídio, atualmente, é de 25 bitcoins e, em breve diminuirá para 12.5 bitcoins; Esta diminuição no subsídio da rede é conhecida como *halving* e acontece a cada 210.000 blocos - aproximadamente 4 anos -, quando esta recompensa é cortada pela metade.

Todo minerador que escreve um novo bloco na blockchain ganha o direito de criar uma transação chamada *coinbase* que é uma exceção por não ter *inputs* e ter apenas o *output* com a recompensa atual da rede para o endereço escolhido pelo minerador. Esta geração de moedas a partir das transações *coinbase* apresenta um crescimento logarítimico no número total de moedas circulando na rede ao longo do tempo:

O número total de bitcoins que serão criados na rede é aproximadamente 21 milhões de bitcoins; precisamente 2099999997690000 *satoshis*. Podemos visualizar este crescimento com um pequeno *script* em Python3:

```
#!/usr/bin/env python3
# Ano em que a rede foi iniciada por Satoshi Nakamoto
START_YEAR = 2009
# Intervalo de anos com blocos de 10 minutos
YEAR_INTERVAL = 4
# A recompensa inical de 50 bitcoins em satoshis
START_BLOCK_REWARD = 50 * 10**8
# Intervalo de blocos entre o halving da recompensa
REWARD_INTERVAL = 210 * 10**3


def show_mine_progress():
    curr_year = START_YEAR
    curr_reward = START_BLOCK_REWARD
    total_coins = 0
    while curr_reward > 0:
        print("Ano: %d, Recompensa atual: %d, Moedas em circulação: %d satoshis" %
              (curr_year, curr_reward, total_coins))
        # O número de moedas é somado com todas as recompensas de 210000 blocos
        # gerados em 4 anos
        total_coins += curr_reward * REWARD_INTERVAL
        curr_year += YEAR_INTERVAL
        # empurra os bits para a direita uma vez, efetivamente dividindo
        # a recompensa pela metade.
        # Ex.: 4 em binário (100) com os bits empurrados para a
        # direita será 2 (010)
        curr_reward >>= 1
    print("\nTotal de moedas em circulação: %d satoshis" % total_coins)
```

O *output* desta função será:

```
>>> show_mine_progress()
Ano: 2009, Recompensa atual: 5000000000, Moedas em circulação: 0 satoshis
Ano: 2013, Recompensa atual: 2500000000, Moedas em circulação: 1050000000000000 satosh
is
Ano: 2017, Recompensa atual: 1250000000, Moedas em circulação: 1575000000000000 satosh
is
Ano: 2021, Recompensa atual: 625000000, Moedas em circulação: 1837500000000000 satoshi
s
Ano: 2025, Recompensa atual: 312500000, Moedas em circulação: 1968750000000000 satoshi
s
Ano: 2029, Recompensa atual: 156250000, Moedas em circulação: 2034375000000000 satoshi
s
Ano: 2033, Recompensa atual: 78125000, Moedas em circulação: 2067187500000000 satoshis
Ano: 2037, Recompensa atual: 39062500, Moedas em circulação: 2083593750000000 satoshis
Ano: 2041, Recompensa atual: 19531250, Moedas em circulação: 2091796875000000 satoshis
Ano: 2045, Recompensa atual: 9765625, Moedas em circulação: 2095898437500000 satoshis
Ano: 2049, Recompensa atual: 4882812, Moedas em circulação: 2097949218750000 satoshis
Ano: 2053, Recompensa atual: 2441406, Moedas em circulação: 2098974609270000 satoshis
Ano: 2057, Recompensa atual: 1220703, Moedas em circulação: 2099487304530000 satoshis
Ano: 2061, Recompensa atual: 610351, Moedas em circulação: 2099743652160000 satoshis
Ano: 2065, Recompensa atual: 305175, Moedas em circulação: 2099871825870000 satoshis
Ano: 2069, Recompensa atual: 152587, Moedas em circulação: 2099935912620000 satoshis
Ano: 2073, Recompensa atual: 76293, Moedas em circulação: 2099967955890000 satoshis
Ano: 2077, Recompensa atual: 38146, Moedas em circulação: 2099983977420000 satoshis
Ano: 2081, Recompensa atual: 19073, Moedas em circulação: 2099991988080000 satoshis
Ano: 2085, Recompensa atual: 9536, Moedas em circulação: 2099995993410000 satoshis
Ano: 2089, Recompensa atual: 4768, Moedas em circulação: 2099997995970000 satoshis
Ano: 2093, Recompensa atual: 2384, Moedas em circulação: 2099998997250000 satoshis
Ano: 2097, Recompensa atual: 1192, Moedas em circulação: 2099999497890000 satoshis
Ano: 2101, Recompensa atual: 596, Moedas em circulação: 2099999748210000 satoshis
Ano: 2105, Recompensa atual: 298, Moedas em circulação: 2099999873370000 satoshis
Ano: 2109, Recompensa atual: 149, Moedas em circulação: 2099999935950000 satoshis
Ano: 2113, Recompensa atual: 74, Moedas em circulação: 2099999967240000 satoshis
Ano: 2117, Recompensa atual: 37, Moedas em circulação: 2099999982780000 satoshis
Ano: 2121, Recompensa atual: 18, Moedas em circulação: 2099999990550000 satoshis
Ano: 2125, Recompensa atual: 9, Moedas em circulação: 2099999994330000 satoshis
Ano: 2129, Recompensa atual: 4, Moedas em circulação: 2099999996220000 satoshis
Ano: 2133, Recompensa atual: 2, Moedas em circulação: 2099999997060000 satoshis
Ano: 2137, Recompensa atual: 1, Moedas em circulação: 2099999997480000 satoshis

Total de moedas em circulação: 2099999997690000 satoshis
```

E assim fica fácil observar que o número de bitcoins que existirão na rede será aproximadamente 20 milhões. A transação *coinbase* tem a sua recompensa calculada no arquivo miner.cpp linha 279 no Bitcoin Core e é verificada pelos nós como visto no arquivo main.cpp linha 2393; ambos utilizando as funções `GetBlockSubsidy` com a `chainparams.GetConsensus` como parametro.

Como funciona

A rede tem um parametro chamado `difficulty` que indica o *hash* do bloco esperado pela rede para que o minerador seja autorizado a atualizar a blockchain. Esta dificuldade é ajustada, se necessário, a cada 2016 blocos com o objetivo de manter um alvo que necessite de 10 minutos **em média** para ser alcançado por algum minerador via *proof-of-work*. Você pode verificar a dificuldade atual pela API JSON-RPC do Bitcoin Core com o comando `getdifficulty`:

```
$ bitcoin-cli getdifficulty
194254820283.444
```

A dificuldade é uma medida do quão difícil é para achar um *hash* abaixo de um certo alvo. O alvo inicial da rede é `0x00000000FFFF0000000000000000000000000000000000000000000000000000` (em *float*, truncado) e é o máximo alvo possível no Bitcoin que representa a mínima dificuldade possível `1` ajustada, desde então, a cada 2016 blocos. O alvo esperado pela rede representa o número mínimo de 0's que o *hash* do próximo bloco deve ter para que seja aceito na atualização da blockchain, ou seja, o minerador deve conseguir criar um *hash* que represente um número menor ou igual ao alvo atual. Este mecanismo no *hashcash* serve como prova de computação devido ao fato de que quantos mais 0's um *hash* tem em seu início, mais trabalho computacional deve ser despendido para que se consiga achar este *hash* e este trabalho computacional é previsível. Para calcular a notação do número retornado pelo comando `getdifficulty`, usamos esta fórmula como base:

```
dificuldade_atual = dificuldade_inicial / alvo_atual
```

No Bitcoin Core, a você pode ver o cálculo da dificuldade no arquivo main.cpp linha 3304 com o uso da função `GetNextWorkRequired` definida no arquivo pow.cpp linha 13.

Para conseguir criar *hashes* diferentes com o mesmo bloco, os mineradores podem alterar o campo *nonce* arbitrariamente para criarem *hashes* completamente diferentes do mesmo bloco. Então, para um minerador provar que alcançou o resultado esperado pela rede basta que apresente o bloco com todos seus elementos incluindo este *nonce* para que qualquer um possa verificar a validade deste *hash* e da validade das transações contidas naquele bloco. No entanto, um valor de 32 *bytes* com suas 4 bilhões de possibilidades diferentes já não é suficiente para um *proof-of-work* que desde 2011 já necessita de mais de 1 quadrilhão de *hashes* para ser resolvido por algum minerador, logo, os mineradores passaram a usar otimizações ao algoritmo de *proof-of-work* original para alterarem o *timestamp* em alguns segundos, a própria transação *coinbase* e a ordem ou composição da lista de transações.

Agora, para facilitar a nossa visualização do trabalho computacional despendido no *proof-of-work*, podemos implementar um script em Python3 para minerar um bloco razoavelmente fácil para uma CPU comum, o bloco 1 após logo após o bloco gênesis. Não utilizarei o próprio bloco gênesis para o exemplo para podermos ver um bloco comum como outros, já que o gênesis é um caso especial e não tem referência ao bloco anterior. Este exemplo é bastante ineficiente para a tarefa e é muito mais útil para entendermos melhor o processo por meio de visualização, no entanto ele pode ser alterado para outros blocos como quiser:

```
#!/usr/bin/env python3
import hashlib
import struct
import codecs
import time

# versao para o bloco
version = 1
# o bloco anteiror (aqui, o bloco genesis)
prev_block = "000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f"
# a raiz de merkle formada das transacoes... nao estamos formando a nossa;
# apenas pegamos a formada no bloco original
merkle_root = "0e3e2357e806b6cdb1f70b54c3a3a17b6714ee1f0e68bebb44a74b1efd512098"
start_time = int(time.time())
bits = 486604799
p = ''

# calculando a string do alvo para checarmos
exp = bits >> 24
mant = bits & 0xffffff
target = mant * (1 << (8 * (exp - 3)))
target_hexstr = '%064x' % target
target_str = codecs.decode(target_hexstr, 'hex')
nonce = 100000000

# apenas printando algumas informacoes sobre o bloco, bloco anterior e alvo
print("Hash do bloco anterior:", prev_block)
print("Raiz de Merkle:", merkle_root)
print("Alvo atual (hex):", target_hexstr)
print()

print("Iniciando mineração...")
while True:
    nonce += 1
    # este e o cabecalho do bloco
    header = (struct.pack('<L', version) +
              codecs.decode(prev_block, 'hex')[::-1] +
              codecs.decode(merkle_root, 'hex')[::-1] +
              struct.pack('<LLL', start_time, bits, nonce))

    # passando o cabecalho na shasha; shasha = sha256(sha256().digest())
    blockhash = hashlib.sha256(hashlib.sha256(header).digest()).digest()
```

```
    blockhash_hex = codecs.encode(blockhash[::-1], 'hex')

    # printando a cada hash com um 0 a mais conseguido
    if blockhash_hex.startswith(p.encode('utf-8')):
        print('\nnonce:', nonce,
              '\nblockhash (hex):', blockhash_hex.decode('utf-8'),
              '\ntempo corrido: %.2f segundos' % (time.time() - start_time),
              '\nnumero de zeros no inicio do hash:', len(p))
        p += '0'

    # se o hash do bloco for menor ou igual ao target, finalizamos
    if blockhash[::-1] <= target_str:
        print("Sucesso!")
        print("Bloco minerado em %d segundos." % (time.time() - start_time))
        break
```

No *output* deste último programa, podemos ver a progressão do esforço computacional até conseguirmos minerar o bloco... o que levou algum tempo:

```
Hash do bloco anterior: 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce2
6f
Raiz de Merkle: 0e3e2357e806b6cdb1f70b54c3a3a17b6714ee1f0e68bebb44a74b1efd512098
Alvo atual (hex): 00000000ffff0000000000000000000000000000000000000000000000000000

Iniciando mineração...

nonce: 100000001
blockhash (hex): 771be0368aaaa3fc1521f4d6db2fd1dfc6e9ef0952fd4bd822ad3b4610f7aebe
tempo corrido: 0.34 segundos
numero de zeros no inicio do hash: 0

nonce: 100000003
blockhash (hex): 0fa9339e51cefb79cc6c61d602cbf549fbe29e8bf17427d840277f2b5037769a
tempo corrido: 0.34 segundos
numero de zeros no inicio do hash: 1

nonce: 100000230
blockhash (hex): 00b911eccc58ef739f7e87cc45fcd445b746ca95f4eec0f3b4ea7854bcdd850e
tempo corrido: 0.34 segundos
numero de zeros no inicio do hash: 2

nonce: 100001054
blockhash (hex): 000046253349900d528f82b17365378925e80e08e4404ce86d0a81be592332ba
tempo corrido: 0.36 segundos
numero de zeros no inicio do hash: 3

nonce: 100075230
blockhash (hex): 0000e9a6a93a2d8c19bc35a797bc558afecb0a4d53247946fc2460dd749a7948
tempo corrido: 1.80 segundos
numero de zeros no inicio do hash: 4

nonce: 100236426
blockhash (hex): 00000a8d845ea7b92ea3f589b0c8e3210e4351aa988daac57bc9b5b46cab26fc
tempo corrido: 4.82 segundos
numero de zeros no inicio do hash: 5

nonce: 121523146
blockhash (hex): 00000048b1a635b6b7fd726bbf7be991dbe2ced3e894df08619e1c081e93ded8
tempo corrido: 425.43 segundos
numero de zeros no inicio do hash: 6

nonce: 148058503
blockhash (hex): 0000000a8256eb559ad433b492a5fbee291a457d4f522631afd5a086a18cbcbd
tempo corrido: 959.89 segundos
numero de zeros no inicio do hash: 7

# [... um tempo de espera até conseguir o alvo... ]
```

Como pode ver pelo *output* com um computador pessoal, levamos um tempo considerável para minerar este bloco; no total * segundos.

E, para finalizarmos esta parte, podemos ver uma implementação simplificada do algoritmo de *proof-of-work* para visualizarmos a progressão dos números com a dificuldade em *bits* aumentando progressivamente:

```
#!/usr/bin/env python3
import hashlib
import time

max_nonce = 2**32


def proof_of_work(header, difficulty_bits):
    # calcula o alvo da dificuldade
    target = 2**(256-difficulty_bits)

    for nonce in range(max_nonce):
        utf8_header = header.encode('utf-8')
        utf8_nonce = str(nonce).encode('utf-8')
        hash_result = hashlib.sha256(utf8_header + utf8_nonce).hexdigest()

        # checa se e um resultado valido
        if int(hash_result, 16) < target:
            print("Sucesso com o nonce %d" % nonce)
            print("Hash é %s" % hash_result)
            return(hash_result, nonce)
    print("Falhou após %d (max_nonce) tentativas" % nonce)
    return nonce

if __name__ == '__main__':
    nonce = 0
    hash_result = ''

    # dificuldade de 0 a 31 bits
    for difficulty_bits in range(32):
        difficulty = 2**difficulty_bits
        print("Dificuldade: %d (%d bits)" % (difficulty, difficulty_bits))

        print("Começando busca...")

        # marca a hora inicial
        start_time = time.time()

        # cria um novo bloco que inclui o hash do anterior
        # usamos apenas uma string vazia como um mock das transacoes
        new_block = 'bloco teste com transações' + hash_result

        hash_result, nonce = proof_of_work(new_block, difficulty_bits)

        # marca a hora final
        end_time = time.time()

        elapsed_time = end_time - start_time
```

```
        print("Tempo Corrido: %.4f segundos" % elapsed_time)

        if elapsed_time > 0:
            # uma estimativa de hashes por segundo
            hash_power = float(nonce/elapsed_time)
            print("Poder de Hashing: %d hashes por segundo" % hash_power)
        print("\n")
```

Rodando este código, podemos observar o aumento da dificuldade em *bits* que representa o número de 0's ao início do valor alvo e observar quanto tempo nosso computador demora para achar uma solução em cada dificuldade. Logo vemos que o tempo aumenta de bastante - com espaço para alguma sorte - de acordo com o aumento do número que o alvo vai ficando menor (com mais 0's ao início) como neste output em um computador pessoal:

```
Dificuldade: 2 (1 bits)
Começando busca...
Sucesso com o nonce 0
Hash é 065243447c61432cce2e6a047e9ee07d104a1ce9a28f9f7a70a12cf84de09b02
Tempo Corrido: 0.0001 segundos
Poder de Hashing: 0 hashes por segundo

# [... outras dificuldades... ]
Dificuldade: 4194304 (22 bits)
Começando busca...
Sucesso com o nonce 7782755
Hash é 00000318dddbe8ba0b1272f58c3d4273640fd26413a41766de3a15f416cd5ddd
Tempo Corrido: 28.6132 segundos
Poder de Hashing: 271998 hashes por segundo

# [...]
Dificuldade: 67108864 (26 bits)
Começando busca...
Sucesso com o nonce 63590018
Hash é 00000018645fee7587700bff0af594aeeb2f0973831df1b8c26ea5a589fcd0a2
Tempo Corrido: 244.7094 segundos
Poder de Hashing: 259859 hashes por segundo

# [...]
```

Com números de uma magnitude muito maior do que a que estamos fazendo em nossos processadores para entendermos o processo, os mineradores estão, neste exato momento, competindo restringidos pelas leis da física para alcançar estes mesmos tipos de resultados. Entendendo isto, observe os *hashes* dos últimos blocos e contemple o processamento colossal necessário para calcular cada um deles para atualizar a blockchain de forma a aumentar exponencialmente a segurança da blockchain contra alterações forjadas, ocupando um papel essencial no estabelecimento de consenso numa rede descentralizada de transferência de valores como o Bitcoin.

Errata

Esta seção é dedicada a listar erros reportados por terceiros. Muito obrigado aos que colaboram.

0

Bug no código exemplo na "Introdução" no item "Funções *Hash* Criptográficas" reportado por **JC GreenMind**, em que um erro no nome da variável causa falha na execução.

Diff da correção:

```
diff --git a/intro.md b/intro.md
index 9241f5f..4b08016 100644
--- a/intro.md
+++ b/intro.md
@@ -53,9 +53,9 @@ Exemplo de ambas funções sendo usadas em Python com a *string* "bitc
oin" como
 1ed7259a5243a1e9e33e45d8d2510bc0470032df964956e18b9f56fa65c96e89

 >>> word2_ripemd160 = hashlib.new('ripemd160')
->>> word_ripemd160.update(word2.encode('utf-8'))
->>> print(word_ripemd160.hexdigest())
-5f67cd0e647825711eac0b0bf78e0487b149bc3a
+>>> word2_ripemd160.update(word2.encode('utf-8'))
+>>> print(word2_ripemd160.hexdigest())
+9d2028ac5216d10b85d1a3ab389ebcc57a3ee6eb
```