←Home

Building Blockchain in Go. Part 5: Addresses

SEPTEMBER 11, 2017
GOLANG BLOCKCHAIN BITCOIN

Introduction

In the previous article, we started implementing transactions. You were also introduced to the impersonal nature of transactions: there are no user accounts, your personal data (e.g., name, passport number or SSN) is not required and not stored anywhere in Bitcoin. But there still must be something that identifies you as the owner of transaction outputs (i.e. the owner of coins locked on these outputs). And this is what Bitcoin addresses are needed for. So far we've used arbitrary user defined strings as addresses, and the time has come to implement real addresses, as they're implemented in Bitcoin.

> This part introduces significant code changes, so it makes no sense explaining all of them here. Please refer to this page to see all the changes since the last article.

Bitcoin Address

Here's an example of a Bitcoin address: 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa. This is the very first Bitcoin address, which allegedly belongs to Satoshi Nakamoto. Bitcoin addresses are public. If you want to send coins to someone, you need to know their address. But addresses (despite being unique) are not something that identifies you as the owner of a "wallet". In fact, such addresses are a human readable representation of public keys. In Bitcoin, your identity is a pair (or pairs) of private and public keys stored on your computer (or stored in some other place you have access to). Bitcoin relies on a

combination of cryptography algorithms to create these keys, and guarantee that no one else in the world can access your coins without getting physical access to your keys. Let's discuss what these algorithms are.

Public-key Cryptography

Public-key cryptography algorithms use pairs of keys: public keys and private keys. Public keys are not sensitive and can be disclosed to anyone. In contrast, private keys shouldn't be disclosed: no one but the owner should have access to them because it's private keys that serve as the identifier of the owner. You are your private keys (in the world of cryptocurrencies, of course).

In essence, a Bitcoin wallet is just a pair of such keys. When you install a wallet application or use a Bitcoin client to generate a new address, a pair of keys is generated for you. The one who controls the private key controls all the coins sent to this key in Bitcoin.

Private and public keys are just random sequences of bytes, thus they cannot be printed on the screen and read by a human. That's why Bitcoin uses an algorithm to convert public keys into a human readable string.

> If you've ever used a Bitcoin wallet application, it's likely that a mnemonic pass phrase was generated for you. Such phrases are used instead of private keys and can be used to generate them. This mechanism is implemented in BIP-039.

Ok, we now know what identifies users in Bitcoin. But how does Bitcoin check the ownership of transaction outputs (and coins stored on them)?

Digital Signatures

In mathematics and cryptography, there's a concept of digital signature – algorithms that guarantee:

1. that data wasn't modified while being transferred from a sender to a recipient;
2. that data was created by a certain sender;
3. that the sender cannot deny sending the data.

By applying a signing algorithm to data (i.e., signing the data), one gets a signature, which can later be verified. Digital signing happens with the usage of a private key, and verification requires a public key.

In order to sign data we need the following things:

1. data to sign;
2. private key.

The operation of signing produces a signature, which is stored in transaction inputs. In order to verify a signature, the following is required:

1. data that was signed;
2. the signature;
3. public key.

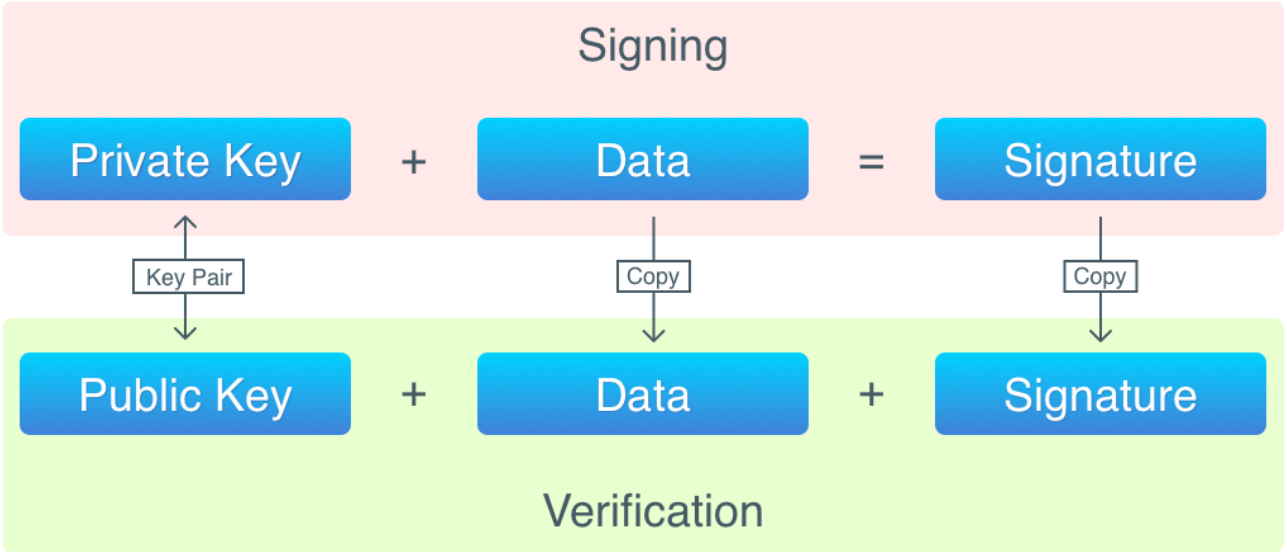
In simple terms, the verification process can be described as: check that this signature was obtained from this data with a private key used to generate the public key.

> Digital signatures are not encryption, you cannot reconstruct the data from a signature. This is similar to hashing: you run data through a hashing algorithm and get a unique representation of the data. The difference between signatures and hashes is key pairs: they make signature verification possible.
> But key pairs can also be used to encrypt data: a private key is used to encrypt, and a public key is used to decrypt the data. Bitcoin doesn't use encryption algorithms though.

Every transaction input in Bitcoin is signed by the one who created the transaction. Every transaction in Bitcoin must be verified before being put in a block. Verification means (besides other procedures):

1. Checking that inputs have permission to use outputs from previous transactions.
2. Checking that the transaction signature is correct.

Schematically, the process of signing data and verifying signature looks likes this:



Let's now review the full lifecycle of a transaction:

1. In the beginning, there's the genesis block that contains a coinbase transaction. There are no real inputs in coinbase transactions, so signing is not necessary. The output of the coinbase transaction contains a hashed public key (`RIPEMD16(SHA256(PubKey))` algorithms are used).
2. When one sends coins, a transaction is created. Inputs of the transaction will reference outputs from previous transaction(s). Every input will store a public key (not hashed) and a signature of the whole transaction.
3. Other nodes in the Bitcoin network that receive the transaction will verify it. Besides other things, they will check that: the hash of the public key in an input matches the hash of the referenced output (this ensures that the sender spends only coins belonging to them); the signature is correct (this ensures that the transaction is created by the real owner of the coins).
4. When a miner node is ready to mine a new block, it'll put the transaction in a block and start mining it.
5. When the blocked is mined, every other node in the network receives a message saying the block is mined and adds the block to the blockchain.
6. After a block is added to the blockchain, the transaction is completed, its outputs can be referenced in new transactions.

Elliptic Curve Cryptography

As described above, public and private keys are sequences of random bytes. Since it's private keys that are used to identify owners of coins, there's a required condition: the randomness algorithm must produce truly random bytes. We don't want to accidentally generate a private key that's owned by someone else.

Bitcoin uses elliptic curves to generate private keys. Elliptic curves is a complex mathematical concept, which we're not going to explain in details here (if you're curious, check out this gentle introduction to elliptic curves WARNING: Math formulas!). What we need to know is that these curves can be used to generate really big and random numbers. The curve used by Bitcoin can randomly pick a number between 0 and 2^{256} (which is approximately 10^{77}, when there are between 10^{78} and 10^{82} atoms in the visible universe). Such a huge upper limit means that it's almost impossible to generate the same private key twice.

Also, Bitcoin uses (and we will) ECDSA (Elliptic Curve Digital Signature Algorithm) algorithm to sign transactions.

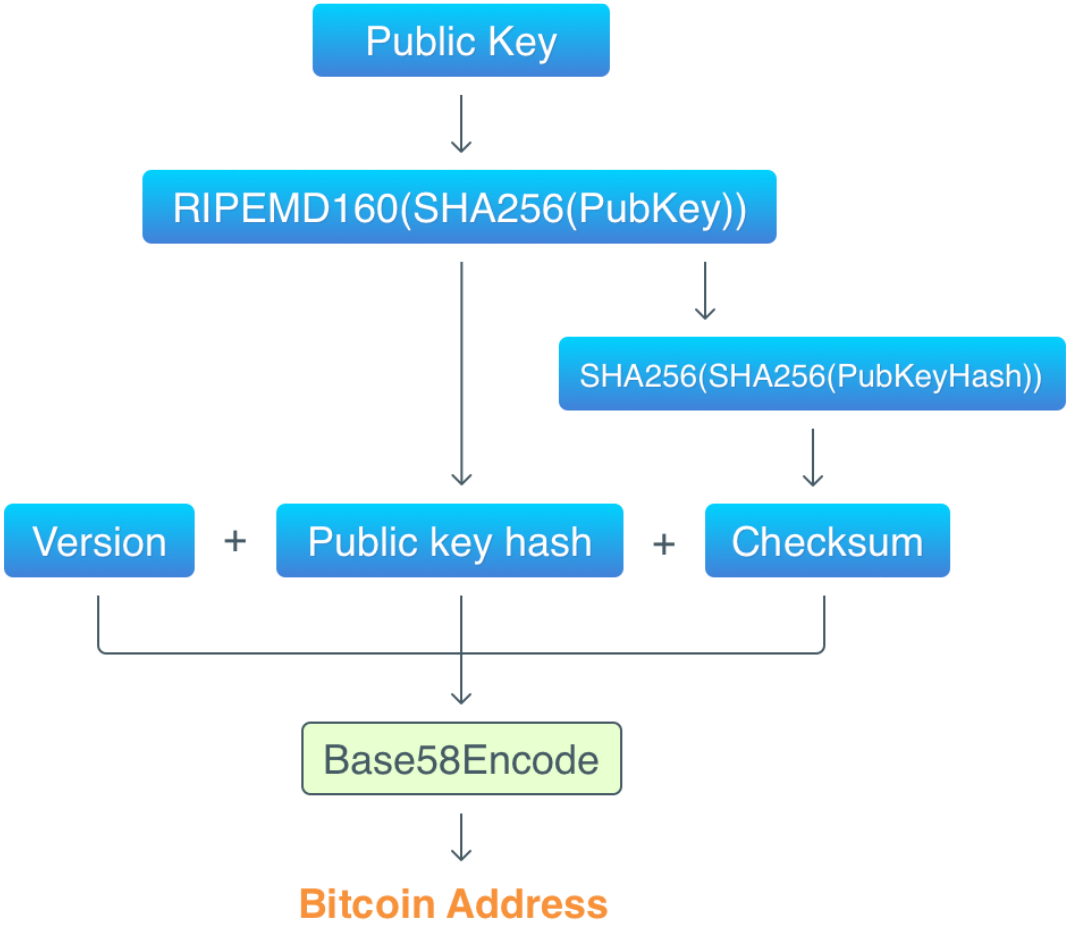
Base58

Now let's get back to the above mentioned Bitcoin address: 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa. Now we know that this is a human-readable representation of a public key. And if we decode it, here's what the public key looks like (as a sequence of bytes written in the hexadecimal system):

```
0062E907B15CBF27D5425399EBF6F0FB50EBB88F18C29B7D93
```

Bitcoin uses the Base58 algorithm to convert public keys into human readable format. The algorithm is very similar to famous Base64, but it uses shorter alphabet: some letters were removed from the alphabet to avoid some attacks that use letters similarity. Thus, there are no these symbols: 0 (zero), O (capital o), I (capital i), l (lowercase L), because they look similar. Also, there are no + and / symbols.

Let's schematically visualize the process of getting an address from a public key:



Thus, the above mentioned decoded public key consists of three parts:

```
Version  Public key hash                            Checksum
00       62E907B15CBF27D5425399EBF6F0FB50EBB88F18  C29B7D93
```

Since hashing functions are one way (i.e., they cannot be reversed), it's not possible to extract the public key from the hash. But we can check if a public key was used to get the hash by running it thought the save hash functions and comparing the hashes.

Ok, now that we have all the pieces, let's write some code. Some of the concepts should be more clear when written in code.

Implementing Addresses

We'll start with the `Wallet` structure:

```
type Wallet struct {
        PrivateKey ecdsa.PrivateKey
        PublicKey  []byte
}

type Wallets struct {
        Wallets map[string]*Wallet
}

func NewWallet() *Wallet {
        private, public := newKeyPair()
        wallet := Wallet{private, public}

        return &wallet
}

func newKeyPair() (ecdsa.PrivateKey, []byte) {
        curve := elliptic.P256()
        private, err := ecdsa.GenerateKey(curve, rand.Reader)
        pubKey := append(private.PublicKey.X.Bytes(),
private.PublicKey.Y.Bytes()...)

        return *private, pubKey
}
```

A wallet is nothing but a key pair. We'll also need the `Wallets` type to keep a collection of wallets, save them to a file, and load them from it. In the construction function of `Wallet` a new key pair is generated. The `newKeyPair` function is straightforward: ECDSA is based on elliptic curves, so we need one. Next, a private key is generated using the curve, and a public key is generated from the private key. One thing to notice: in elliptic curve based algorithms, public keys are points on a curve. Thus, a public key is a combination of X, Y coordinates. In Bitcoin, these coordinates are concatenated and form a public key.

Now, let's generate an address:

```
func (w Wallet) GetAddress() []byte {
        pubKeyHash := HashPubKey(w.PublicKey)

        versionedPayload := append([]byte{version}, pubKeyHash...)
        checksum := checksum(versionedPayload)

        fullPayload := append(versionedPayload, checksum...)
```

```
        address := Base58Encode(fullPayload)

        return address
}

func HashPubKey(pubKey []byte) []byte {
        publicSHA256 := sha256.Sum256(pubKey)

        RIPEMD160Hasher := ripemd160.New()
        _, err := RIPEMD160Hasher.Write(publicSHA256[:])
        publicRIPEMD160 := RIPEMD160Hasher.Sum(nil)

        return publicRIPEMD160
}

func checksum(payload []byte) []byte {
        firstSHA := sha256.Sum256(payload)
        secondSHA := sha256.Sum256(firstSHA[:])

        return secondSHA[:addressChecksumLen]
}
```

Here are the steps to convert a public key into a Base58 address:

1. Take the public key and hash it twice with `RIPEMD160(SHA256(PubKey))` hashing algorithms.
2. Prepend the version of the address generation algorithm to the hash.
3. Calculate the checksum by hashing the result of step 2 with `SHA256(SHA256(payload))`. The checksum is the first four bytes of the resulted hash.
4. Append the checksum to the `version+PubKeyHash` combination.
5. Encode the `version+PubKeyHash+checksum` combination with Base58.

As a result, you'll get a **real Bitcoin address**, you can even check its balance on blockchain.info. But I can assure you that the balance is 0 no matter how many times you generate a new address and check its balance. This is why choosing proper public-key cryptography algorithm is so crucial: considering private keys are random numbers, the chance of generating the same number must be as low as possible. Ideally, it must be as low as "never".

Also, pay attention that you don't need to connect to a Bitcoin node to get an address. The address generation algorithm utilizes a combination of open algorithms that are implemented in many programming languages and libraries.

Now we need to modify inputs and outputs for them to use addresses:

```
type TXInput struct {
        Txid      []byte
        Vout      int
        Signature []byte
        PubKey    []byte
}

func (in *TXInput) UsesKey(pubKeyHash []byte) bool {
        lockingHash := HashPubKey(in.PubKey)

        return bytes.Compare(lockingHash, pubKeyHash) == 0
}

type TXOutput struct {
        Value      int
        PubKeyHash []byte
}

func (out *TXOutput) Lock(address []byte) {
        pubKeyHash := Base58Decode(address)
        pubKeyHash = pubKeyHash[1 : len(pubKeyHash)-4]
        out.PubKeyHash = pubKeyHash
}

func (out *TXOutput) IsLockedWithKey(pubKeyHash []byte) bool {
        return bytes.Compare(out.PubKeyHash, pubKeyHash) == 0
}
```

Notice, that we're no longer using `ScriptPubKey` and `ScriptSig` fields, because we're not going to implement a scripting language. Instead, `ScriptSig` is split into `Signature` and `PubKey` fields, and `ScriptPubKey` is renamed to `PubKeyHash`. We'll implement the same outputs locking/unlocking and inputs signing logics as in Bitcoin, but we'll do this in methods instead.

The `UsesKey` method checks that an input uses a specific key to unlock an output. Notice that inputs store raw public keys (i.e., not hashed), but the function takes a hashed one. `IsLockedWithKey` checks if provided public key hash was used to lock the output. This is a complementary function to `UsesKey`, and they're both used in `FindUnspentTransactions` to build connections between transactions.

`Lock` simply locks an output. When we send coins to someone, we know only their address, thus the function takes an address as the only argument. The address is then decoded and the public key hash is extracted from it and saved in the `PubKeyHash` field.

Now, let's check that everything works correctly:

```
$ blockchain_go createwallet
Your new address: 13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt

$ blockchain_go createwallet
Your new address: 15pUhCbtrGh3JUx5iHnXjfpyHyTgawvG5h

$ blockchain_go createwallet
Your new address: 1Lhqun1E9zZZhodiTqxfPQBcwr1CVDV2sy

$ blockchain_go createblockchain -address
13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt
0000005420fbfdafa00c093f56e033903ba43599fa7cd9df40458e373eee724d

Done!

$ blockchain_go getbalance -address
13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt
Balance of '13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt': 10

$ blockchain_go send -from 15pUhCbtrGh3JUx5iHnXjfpyHyTgawvG5h -to
13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt -amount 5
2017/09/12 13:08:56 ERROR: Not enough funds

$ blockchain_go send -from 13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt -to
15pUhCbtrGh3JUx5iHnXjfpyHyTgawvG5h -amount 6
00000019afa909094193f64ca06e9039849709f5948fbac56cae7b1b8f0ff162

Success!

$ blockchain_go getbalance -address
13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt
Balance of '13Uu7B1vDP4ViXqHFsWtbraM3EfQ3UkWXt': 4

$ blockchain_go getbalance -address
15pUhCbtrGh3JUx5iHnXjfpyHyTgawvG5h
Balance of '15pUhCbtrGh3JUx5iHnXjfpyHyTgawvG5h': 6

$ blockchain_go getbalance -address
1Lhqun1E9zZZhodiTqxfPQBcwr1CVDV2sy
Balance of '1Lhqun1E9zZZhodiTqxfPQBcwr1CVDV2sy': 0
```

Nice! Now let's implement transaction signatures.

Implementing Signatures

Transactions must be signed because this is the only way in Bitcoin to guarantee that one cannot spend coins belonging to someone else. If a signature is invalid, the transaction is considered invalid too and, thus, cannot be added to the blockchain.

We have all the pieces to implement transactions signing, except one thing: data to sign. What parts of a transaction are actually signed? Or a transaction is signed as a whole? Choosing data to sign is quite important. The thing is that data to be signed must contain information that identifies the data in a unique way. For example, it makes no sense signing just output values because such signature won't consider the sender and the recipient.

Considering that transactions unlock previous outputs, redistribute their values, and lock new outputs, the following data must be signed:

1. Public key hashes stored in unlocked outputs. This identifies "sender" of a transaction.
2. Public key hashes stored in new, locked, outputs. This identifies "recipient" of a transaction.
3. Values of new outputs.

> In Bitcoin, locking/unlocking logic is stored in scripts, which are stored in `ScriptSig` and `ScriptPubKey` fields of inputs and outputs, respectively. Since Bitcoins allows different types of such scripts, it signs the whole content of `ScriptPubKey`.

As you can see, we don't need to sign the public keys stored in inputs. Because of this, in Bitcoin, it's not a transaction that's signed, but its trimmed copy with inputs storing `ScriptPubKey` from referenced outputs.

> A detailed process of getting a trimmed transaction copy is described here. It's likely to be outdated, but I didn't manage to find a more reliable source of information.

Ok, it looks complicated, so let's start coding. We'll start with the `Sign` method:

```
func (tx *Transaction) Sign(privKey ecdsa.PrivateKey, prevTXs
map[string]Transaction) {
        if tx.IsCoinbase() {
                return
        }

        txCopy := tx.TrimmedCopy()

        for inID, vin := range txCopy.Vin {
                prevTx := prevTXs[hex.EncodeToString(vin.Txid)]
                txCopy.Vin[inID].Signature = nil
                txCopy.Vin[inID].PubKey =
prevTx.Vout[vin.Vout].PubKeyHash
                txCopy.ID = txCopy.Hash()
                txCopy.Vin[inID].PubKey = nil

                r, s, err := ecdsa.Sign(rand.Reader, &privKey,
txCopy.ID)
                signature := append(r.Bytes(), s.Bytes()...)

                tx.Vin[inID].Signature = signature
        }
}
```

The method takes a private key and a map of previous transactions. As mentioned above, in order to sign a transaction, we need to access the outputs referenced in the inputs of the transaction, thus we need the transactions that store these outputs.

Let's review this method step by step:

```
if tx.IsCoinbase() {
        return
}
```

Coinbase transactions are not signed because there are no real inputs in them.

```
txCopy := tx.TrimmedCopy()
```

A trimmed copy will be signed, not a full transaction:

```
func (tx *Transaction) TrimmedCopy() Transaction {
        var inputs []TXInput
        var outputs []TXOutput

        for _, vin := range tx.Vin {
                inputs = append(inputs, TXInput{vin.Txid, vin.Vout,
nil, nil})
        }

        for _, vout := range tx.Vout {
                outputs = append(outputs, TXOutput{vout.Value,
vout.PubKeyHash})
        }

        txCopy := Transaction{tx.ID, inputs, outputs}

        return txCopy
}
```

The copy will include all the inputs and outputs, but `TXInput.Signature` and `TXInput.PubKey` are set to nil.

Next, we iterate over each input in the copy:

```
for inID, vin := range txCopy.Vin {
        prevTx := prevTXs[hex.EncodeToString(vin.Txid)]
        txCopy.Vin[inID].Signature = nil
        txCopy.Vin[inID].PubKey = prevTx.Vout[vin.Vout].PubKeyHash
```

In each input, `Signature` is set to `nil` (just a double-check) and `PubKey` is set to the `PubKeyHash` of the referenced output. At this moment, all transactions but the current one are "empty", i.e. their `Signature` and `PubKey` fields are set to nil. Thus, **inputs are signed separately**, although this is not necessary for our application, but Bitcoin allows transactions to contain inputs referencing different addresses.

```
        txCopy.ID = txCopy.Hash()
        txCopy.Vin[inID].PubKey = nil
```

The `Hash` method serializes the transaction and hashes it with the SHA-256 algorithm. The resulted hash is the data we're going to sign. After getting the hash we should reset the `PubKey` field, so it doesn't affect further iterations.

Now, the central piece:

```
        r, s, err := ecdsa.Sign(rand.Reader, &privKey, txCopy.ID)
        signature := append(r.Bytes(), s.Bytes()...)

        tx.Vin[inID].Signature = signature
```

We sign `txCopy.ID` with `privKey`. An ECDSA signature is a pair of numbers, which we concatenate and store in the input's `Signature` field.

Now, the verification function:

```
func (tx *Transaction) Verify(prevTXs map[string]Transaction) bool {
        txCopy := tx.TrimmedCopy()
        curve := elliptic.P256()

        for inID, vin := range tx.Vin {
                prevTx := prevTXs[hex.EncodeToString(vin.Txid)]
                txCopy.Vin[inID].Signature = nil
                txCopy.Vin[inID].PubKey =
prevTx.Vout[vin.Vout].PubKeyHash
                txCopy.ID = txCopy.Hash()
                txCopy.Vin[inID].PubKey = nil

                r := big.Int{}
                s := big.Int{}
                sigLen := len(vin.Signature)
                r.SetBytes(vin.Signature[:(sigLen / 2)])
                s.SetBytes(vin.Signature[(sigLen / 2):])

                x := big.Int{}
                y := big.Int{}
                keyLen := len(vin.PubKey)
                x.SetBytes(vin.PubKey[:(keyLen / 2)])
                y.SetBytes(vin.PubKey[(keyLen / 2):])

                rawPubKey := ecdsa.PublicKey{curve, &x, &y}
                if ecdsa.Verify(&rawPubKey, txCopy.ID, &r, &s) ==
false {
                        return false
```

```
                }
            }

            return true
    }
```

The method is quite straightforward. First, we need the same transaction copy:

```
    txCopy := tx.TrimmedCopy()
```

Next, we'll need the same curve that is used to generate key pairs:

```
    curve := elliptic.P256()
```

Next, we check signature in each input:

```
    for inID, vin := range tx.Vin {
            prevTx := prevTXs[hex.EncodeToString(vin.Txid)]
            txCopy.Vin[inID].Signature = nil
            txCopy.Vin[inID].PubKey = prevTx.Vout[vin.Vout].PubKeyHash
            txCopy.ID = txCopy.Hash()
            txCopy.Vin[inID].PubKey = nil
```

This piece is identical to the one in the `Sign` method, because during verification we need the same data what was signed.

```
            r := big.Int{}
            s := big.Int{}
            sigLen := len(vin.Signature)
            r.SetBytes(vin.Signature[:(sigLen / 2)])
            s.SetBytes(vin.Signature[(sigLen / 2):])

            x := big.Int{}
            y := big.Int{}
            keyLen := len(vin.PubKey)
            x.SetBytes(vin.PubKey[:(keyLen / 2)])
            y.SetBytes(vin.PubKey[(keyLen / 2):])
```

Here we unpack values stored in `TXInput.Signature` and `TXInput.PubKey`, since a signature is a pair of numbers and a public key is a pair of coordinates. We concatenated them earlier for storing, and now we need to unpack them to use in `crypto/ecdsa` functions.

```
		rawPubKey := ecdsa.PublicKey{curve, &x, &y}
		if ecdsa.Verify(&rawPubKey, txCopy.ID, &r, &s) == false {
			return false
		}
	}

	return true
```

Here it is: we create an `ecdsa.PublicKey` using the public key extracted from the input and execute `ecdsa.Verify` passing the signature extracted from the input. If all inputs are verified, return true; if at least one input fails verification, return false.

Now, we need a function to obtain previous transactions. Since this requires interaction with the blockchain, we'll make it a method of `Blockchain`:

```
func (bc *Blockchain) FindTransaction(ID []byte) (Transaction, error)
{
	bci := bc.Iterator()

	for {
		block := bci.Next()

		for _, tx := range block.Transactions {
			if bytes.Compare(tx.ID, ID) == 0 {
				return *tx, nil
			}
		}

		if len(block.PrevBlockHash) == 0 {
			break
		}
	}

	return Transaction{}, errors.New("Transaction is not found")
}
```

```
func (bc *Blockchain) SignTransaction(tx *Transaction, privKey
ecdsa.PrivateKey) {
        prevTXs := make(map[string]Transaction)

        for _, vin := range tx.Vin {
                prevTX, err := bc.FindTransaction(vin.Txid)
                prevTXs[hex.EncodeToString(prevTX.ID)] = prevTX
        }

        tx.Sign(privKey, prevTXs)
}

func (bc *Blockchain) VerifyTransaction(tx *Transaction) bool {
        prevTXs := make(map[string]Transaction)

        for _, vin := range tx.Vin {
                prevTX, err := bc.FindTransaction(vin.Txid)
                prevTXs[hex.EncodeToString(prevTX.ID)] = prevTX
        }

        return tx.Verify(prevTXs)
}
```

These functions are simple: `FindTransaction` finds a transaction by ID (this requires iterating over all the blocks in the blockchain); `SignTransaction` takes a transaction, finds transactions it references, and signs it; `VerifyTransaction` does the same, but verifies the transaction instead.

Now, we need to actually sign and verify transactions. Signing happens in the `NewUTXOTransaction`:

```
func NewUTXOTransaction(from, to string, amount int, bc *Blockchain)
*Transaction {
        ...

        tx := Transaction{nil, inputs, outputs}
        tx.ID = tx.Hash()
        bc.SignTransaction(&tx, wallet.PrivateKey)

        return &tx
}
```

Verification happens before a transaction is put into a block:

```
func (bc *Blockchain) MineBlock(transactions []*Transaction) {
        var lastHash []byte

        for _, tx := range transactions {
                if bc.VerifyTransaction(tx) != true {
                        log.Panic("ERROR: Invalid transaction")
                }
        }
        ...
}
```

And that's it! Let's check everything one more time:

```
$ blockchain_go createwallet
Your new address: 1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR

$ blockchain_go createwallet
Your new address: 1NE86r4Esjf53EL7fR86CsfTZpNN42Sfab

$ blockchain_go createblockchain -address
1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR
000000122348da06c19e5c513710340f4c307d884385da948a205655c6a9d008

Done!

$ blockchain_go send -from 1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR -to
1NE86r4Esjf53EL7fR86CsfTZpNN42Sfab -amount 6
0000000f3dbb0ab6d56c4e4b9f7479afe8d5a5dad4d2a8823345a1a16cf3347b

Success!

$ blockchain_go getbalance -address
1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR
Balance of '1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR': 4

$ blockchain_go getbalance -address
1NE86r4Esjf53EL7fR86CsfTZpNN42Sfab
Balance of '1NE86r4Esjf53EL7fR86CsfTZpNN42Sfab': 6
```

Nothing is broken. Awesome!

Let's also comment out the `bc.SignTransaction(&tx, wallet.PrivateKey)` call in `NewUTXOTransaction` to ensure that unsigned transactions cannot be mined:

```
func NewUTXOTransaction(from, to string, amount int, bc *Blockchain)
*Transaction {
    ...
        tx := Transaction{nil, inputs, outputs}
        tx.ID = tx.Hash()
        // bc.SignTransaction(&tx, wallet.PrivateKey)

        return &tx
}
```

```
$ go install
$ blockchain_go send -from 1AmVdDvvQ977oVCpUqz7zAPUEiXKrX5avR -to
1NE86r4Esjf53EL7fR86CsfTZpNN42Sfab -amount 1
2017/09/12 16:28:15 ERROR: Invalid transaction
```

Conclusion

It's really awesome that we've got so far and implemented so many key features of Bitcoin! We've implemented almost everything outside networking, and in the next part, we'll finish transactions.

Links:

1. Full source codes
2. Public-key cryptography
3. Digital signatures
4. Elliptic curve
5. Elliptic curve cryptography
6. ECDSA
7. Technical background of Bitcoin addresses
8. Address
9. Base58
10. A gentle introduction to elliptic curve cryptography

Ivan Kuznetsov
Write things

Read more