thesis

Table of contents

```
 style: number
 min_depth: 1
 max_depth: 6
Copy
```

Introduction

Thesis goals

The idea behind this thesis was to expand my practical knowledge in the field of the advanced cryptography implemented in coins that feature greater than normal transactional confidentiality such as Monero or Zcash. It ended up consisting of two separate parts: a basic p2p network and a cryptographic module allowing for fully confidential transactions with blinded receiver and sender using one-time addresses and ring signatures with key images. The programming language I chose was Go. The main reasons for choosing Go were:

- compilation speed for fast code iteration,
- execution speed,
- portability of binaries,
- really good standard libraries supporting network programming,
- plethora of user submitted libraries (especially for elliptic curve cryptography (ECC)).

Document structure

This thesis can be divided into 3 main parts:

- First, a short summary of cryptocurrency history is presented.
- Secondly, theoretical concepts for non-trivial algorithms and mathematical structures are introduced.
- Lastly, design rationales related to the code are presented along an explainer for the programs output.

At the very end some conclusions are presented together with ideas of improvements and further developments for the project.

Brief history of cryptocurrencies

In this section I would like to present a short history of concepts that came before Bitcoin and led to its creation, as well as show some interesting developments that came afterwards.

The idea of digital cash had been broadly discussed by people in the Comp-sci field, especially the crypto-punk community, for the best part of the last 60 years. Various iterations and concepts were introduced and slowly built upon. The invention of Hashcash was a huge milestone and directly led to making the final step in this process (for more information see below, Satoshi quoted the algorithm as a solution to the notoriously hard problem of proof-of-work).
In 2009 Satoshi Nakamoto (pseudonym for an anonymous author) published the white-paper [@{Bitcoin}] "Bitcoin: A Peer-to-Peer Electronic Cash System" [citation number x, ref: http://satoshinakamoto.me/bitcoin.pdf], which was the first real and working implementation of the concept later called cryptocurrency. It relied on elliptic curve cryptography (ECC) and `sha256` based proof-of-work, as well as some game theory concepts to allow users to perform completely decentralised and permissionless payments.

What came later was a true renaissance of digital cash systems with iteration cycles for new technologies getting faster and faster with each generation. Different ledger technologies were proposed, the idea of proof-of-stake was conceived to reduce the energy impact of proof-of-work and multiple cryptographic breakthroughs were made to allow for faster and more private transactions. Below, listed by name, are the most important developments that came before and after Bitcoin.

DigiCash

The first company implementing a "digital cash" system was probably DigiCash founded in the late 80 by David Chaum (*DigiCash*, 2021). It promised to implement anonymous online payments, but failed because of the centralisation of its components. Nonetheless it introduced the concept of **blind signatures** which later got to be known as addresses in more recent cryptocurrencies.

Bit-gold

Bit-gold was an idea introduced by cryptographer Nick Szabo in 1998 (*Bit Gold / Satoshi Nakamoto Institute*, 2019). It was the first real, decentralised, cryptographically sound digital cash concept which never got implemented. It introduced the concept of Proof-of-Work which was arguably it's biggest break-through. The project was so similar to Bitcoin that many suspected Satoshi Nakamoto to be Nick Szabo. He later denied those allegations.

Hashcash

Hashcash is a proof-of-work algorithm invented by Adam Back in order to combat spam and DDoS attacks (Wikipedia Contributors, 2019). It was used in the Bitcoin white-paper as the default algorithm for mining because of it's simplicity and elegance. Adam Back has been heavily involved in the development of Bitcoin since the very beginning through his company: Blockstream.

Ethereum

Vitalik Buterin created Ethereum in 2015 after years of work with his team. Ethereum is a cryptocurrency that implements a general purpose, Turing-complete language inside its binary (the Ethereum Virtual Machine - EVM). This allows for a more complicated computation than Bitcoin, which introduced the concept of smart contracts (programs stored on the blockchain that execute code when given conditions are met). Use-cases for this involve decentralised currency-exchanges without third parties and non-fungible tokens.

Monero

The origin of Monero (Esperanto for Coin) was a white-paper from 2013 called "CryptoNote v2" authored by Nicolas van Saberhagen - a fictional character. The original paper expanded on the ideas introduced by Satoshi Nakamoto by introducing confidential transactions. Monero built on those concepts with further research, hiring doctorate and post-doc students to perform paid work on developing its cryptography, and is now the leading project in terms of privacy preserving e-cash systems along with Z-Cash.

Theoretical concepts

Zero knowledge proofs

A zero knowledge proof (ZKP) is a method by which one party can prove to another party that a statement is true, at the same time not disclosing any additional sensitive information (such as, for example, private keys or secret values).

Ali Baba cave example The Ali Baba cave is often quoted as a good example for the ideas behind ZKPs
Let's assume two characters: Alice and Bob. While on an adventure, they walk through a cave inside which they find two tunnels, A and B. At the end of the tunnels, there is a door connecting both of them, to which Bob knows the code.

Alice would like to purchase it from him, but first she wants to make sure he actually knows it.
In this situation it's clear that Bob is the proving party while Alice is the verifier. The proof will work in the following fashion: Alice asks Bob to enter the cave without looking. After he enters one of the tunnels, she then asks him to exit the cave using one of them (A or B). If he knows the code to the door connecting the tunnels, if necessary, Bob will use the code and switch tunnels, exiting by the one requested by Alice. Let's assume he got lucky and entered through tunnel A, while being requested to also exit via this tunnel (a 50% probability of this outcome). Alice can conduct the test as many times as required, reducing the probability of Bob cheating with each iteration.
This example nicely shows general principles behind zero-knowledge proofs: Alice won't know the code until she buys it and Bob doesn't have to provide any additional information to prove he knows it.

Elliptic curve cryptography (ECC)

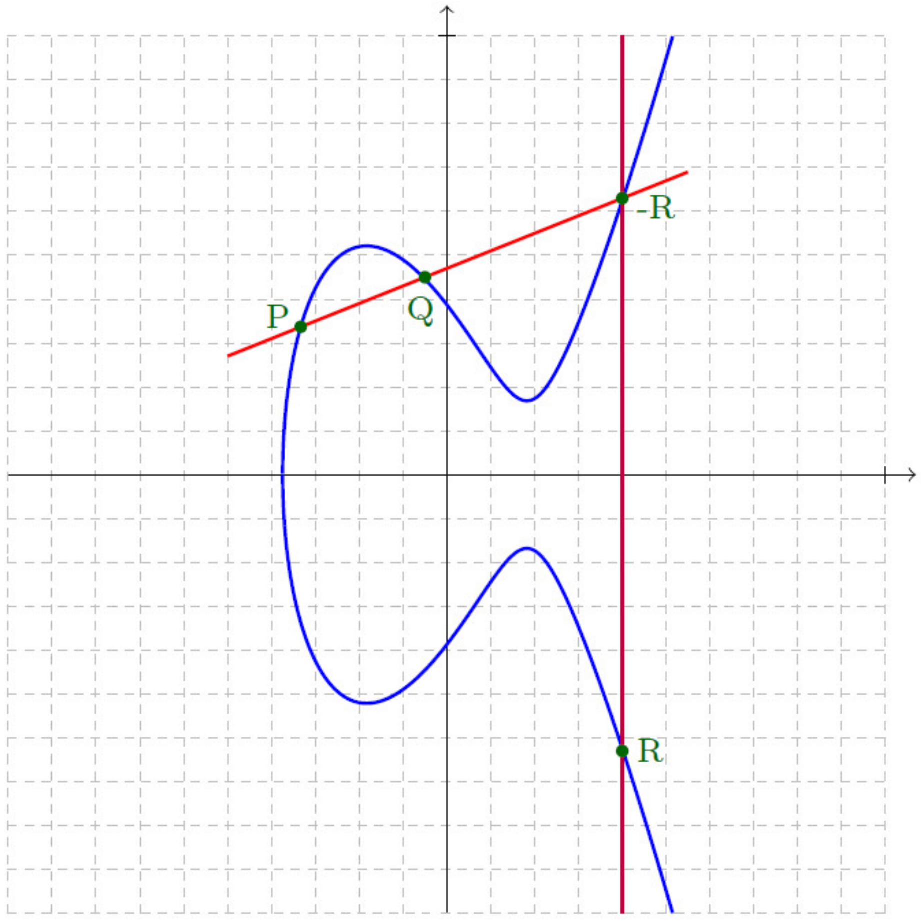
Elliptic curve cryptography is a variant of public-key cryptography schemes based on elliptic curves over finite fields. They provide stronger algebraical guarantees than systems based on prime number factorisation such as RSA, which allows for more intricate protocols. An elliptic curve is a plane curve described by the following equation

$$y^2 = f(x)$$

where $f(x)$ is a cubic polynomial with no repeated roots. We can now define a group structure on any smooth curve of this type. The curve is symmetric with respect to reflections along the x-axis; for a point $P = (a, b)\$, \$ - P$ will have the coordinations $(a, -b)$.
Given two points P, Q, let's draw a line that passes through both of them. We can now describe point R as being at the intersection of the drawn line and the curve.
Let's define the operation + in the following fashion

$$P + Q = -R$$

a visual representation is shown below.



https://www.researchgate.net/figure/Point-addition-on-an-elliptic-curve-defined-over-the-field-R_fig1_338648714

We can now define the operation $a \cdot B$ (multiplication) as the $+$ operation repeated a times on point B and an agreed upon beforehand generator point G. To define a cryptographic scheme over a curve we first have to agree on it's specific parameters, an example of which is shown for Curve25519 (see below). After choosing a generator point we define a scalar value x - the number of subsequent additions we will perform. The discrete logarithm problem can be defined as follows:
Given x it is trivial to perform the operation

$$P = xG$$

where G is the generator point (every addition passes through it) and P is the point at which we end up after performing repeated multiplication. On the other hand it is computationally infeasible to compute x (given a large enough

value for x) while only having P, G.
In summary:

- We define x - the number of repeated multiplications as our **private key**.
- We define P - the resulting point as our **public key**.

Prime field

A field is a fundamental algebraic structure. It is a set on which addition, subtraction, multiplication, and division behave as the corresponding operations on rational and real numbers.
We can define B as a subfield of A when it is a subset of A with respect to its field operations. A field is called a **prime field** if it has no strictly smaller subfields. In than case it is isomorphic either to $\mathbb{Q}$ or a finite field $\mathbb{F}_p$ of prime order.

Curve25519

It is one of the fastest ECC curves, which reference implementation is a public domain software. By design, it accepts any 32-byte integer as a valid public key. It is given with the formula:

$$y^2 = x^3 + 486663x^2 + x$$

over the prime field defined by the prime number $2^{255} - 19$ with the base point $x = 9$. This generates a cyclic subgroup with order $2^{252} + 27742317777372353535851937790883648493$ and a co-factor equal 8. This results in the number of elements in the subgroup being equal to $^1/_8$ of the elliptic curve group. Using a subgroup prevents prevents mounting cryptographic attacks. (*Curve25519*, 2021)

Cryptographic hash functions

A cryptographic hash function is one that maps arbitrarily sized data to a fixed-sized value. It guarantees that the transformation:

$$F(data) \rightarrow H$$

is fast and H is uniformly distributed in the hash function result space. Simultaneously

$$F'(H) \rightarrow data$$

should be impossible to compute in any other way than brute-forcing every single possible combination.

Below are examples of inputs for `md5, sha1 and sha256` (standardised and provably secure cryptographic hash functions) with their output presented as hex strings.

```
md5("the lazy dog jumps over the quick brown fox")    -> faf3071c93f0426384a6c20363c4ada1
sha1("the lazy dog jumps over the quick brown fox")   -> f2055aef1868865a131170dd1b9d99fba74
sha256("the lazy dog jumps over the quick brown fox") -> cab3cf1fe52f03f75f698781825e0e7421c
Copy
```

It is important to note that since every sha256 output contains 256 bits, each hash is also a valid curve25519 public key. For this reason it will be used extensively in algorithms in the project.

Base58

Base58 is a binary-to-text encoding designed to reduce the ambiguity between letters in the resulting encoding (a flaw of base64). It has been created for users that manually read and verify data and as such is perfect for encoding cryptographic addresses. Below an un-formal definition is presented.
Base58 alphabet: 123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz
Pseudo-code:

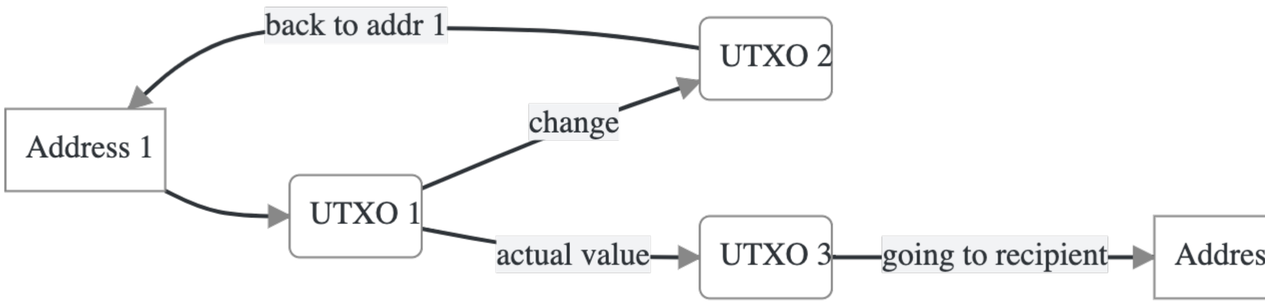
```
carry = 0
output = []
for byte in input:
    carry += byte*58
    output.append(carry % 256)
    carry //= 256 # integer division
Copy
```

Example of base58 encoding:

```
the lazy dog jumps over the quick brown fox -> 2k46VaHwYpGsJyJJ9mMR2vdsm1a9cajq7JMU2mJRHcMVI
Copy
```

UTxO

A UTxO (Unspent Transaction Output) is the building block of all transactions. As the name suggests, it contains value which can be further split into more outputs if needed. Here is a diagram for a typical transaction with a change:



Each node in the network holds a set of all UTXOs with corresponding one-time confidential addresses(see below), which they update with every new valid transaction. In summary this set represents the current state of the network (”monetary” value in each address).

Public key encoding

To facilitate exchange of keys between users, I propose an algorithm to encode public keys into human readable format. It can be outlined as follows:

- Create byte buffer - `buf` for computing output
- Take a public key (A, B)
- Insert bytes of both public keys into `buf`
- Compute the `sha256` sum of `buf`
- Add the first 8 bytes of the checksum to the end of `buf`
- Compute `base58` sum of given buffer for human readability
- Add `lrn1` (a string similar to name of the project - `learncoin`) at the beginning of resulting `base58` string to further enhance readability
  The decoding algorithm is defined as the inverse of this one. Example of encoding a key:

```
A = 13cdf09200021f90cb3ad6a695e5e01667f77d5ef0139660051e9271ca205eee
B = f93caf372e18a48040090c1581fcd5b355d592e6d5883b28b79075ece2a49f74

Encoded key: lrn1HxMwT57Bbg79x2zHAS1AA9D9nZPNTMEWKXJ27kAaYdSPCeBtwnwH6fGJfYBDNQeYmHjUMRJjUX
Copy
```

Implementation

The goal of the project was to understand the inner workings of modern cryptocurrencies and develop an independent implementation based on ZKPs. It has 2 distinct layers:

- ***A peer-to-peer networking layer, involving a consensus algorithm that’s used to agree on a shared chain state.***
- ***A cryptographic layer allowing users to perform transactions and exchange value.***

This project develops both of those parts without joining them. Such an endeavour exceeds the scope of this work and is left for the future.

Cryptography

Outline of goals My goal with this component was to create a cryptographic scheme allowing for anonymous and untraceable exchange of value between addresses on the blockchain. The design was based on the paper *CryptoNote v2.0* by Nicolas van Saberhagen. The author of the paper set out to improve

on the Bitcoin white-paper in various aspects, one of them being the cryptographic aspect of the cryptocurrency. Provided is analysis of flawed bitcoin privacy in academic papers: ref: https://css.csail.mit.edu/6.858/2013/projects/jeffchan-exue-tanyaliu.pdf ref: https://arxiv.org/pdf/2002.06403.pdf ref: https://arxiv.org/pdf/1502.01657.pdf
ref: https://www.bytecoin.org/old/whitepaper.pdf

Improvements can be roughly summarised into 2 components:

One-time confidential addresses One major flow of Bitcoin is traceability of transactions. Without much care it is more than likely to receive two UTXOs into one address, in effect linking them forever as going to one recipient. This is due to the fact that a bitcoin address is just a hash of the users public key (and not a dynamically generated address by design like the one presented below). Ideas like BIP-32 (Bitcoin Improvement Proposal 32)[https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki] are a possibility but since they are not imposed by default, most users do not make use this enhancement. This, in turn, leads to worse privacy for the entire network. With careful statistical analysis of the transaction graph one could, with a high percentage probability, uncover someone's entire transactional history, which could be disastrous for private individuals and entities.

The author of the *CryptoNote v2.0* white-paper proposed a scheme which allows for a non-interactive generation of one-time addresses. It requires every user to generate:

- a private user key (a, b)
- a corresponding public user key (A, B)
  subsequently publishing the public user key to allow for incoming transactions. Address generation proceeds as follows
- Sender unpacks(see above for a proposal for encoding addresses in human readable format) the provided public key pair (A, B)

```
lrn13FdrxyjAshFzkeZscWEEGnKxeX49bdgkbWKvv5gxHRo2gFg3YyLmEBtTTA7zunGiUBpXFkcJX6Sc7SvBkKbMmxj
Copy
```

- Sender generates a random *ed25519* scalar value r and computes P, the one-time destination address. Since `sha256` hashes are valid *ed25519* scalars, we can perform regular ECC operations (additions, multiplication, subtraction) with them

$$P = \text{sha256}(rA)G + B$$

- Sender computes an *ed25519* point R given by the equation $R = rG$, where G is the generator point of *Ed25519* elliptic curve
- Sender sends the transaction to the one time address defined as `{P,R}`.
- Receiver checks every incoming transaction with his private user key (a, b) and computes

$$P' = \text{sha256}(aR)G + B$$

- If $P = P'$ it means this transaction is meant for the receiver. This works because in the elliptic curve algebra we have:

$$rA = raG = arG = aR$$

- The receiver is the only one who can recover x - the one-time private key corresponding to our one-time address P ,

$$x = \text{sha256}(aR) + b$$

  This means the receiver can spend this output by using x in a ring-signature verifying his/hers ownership of this address.

Ring signatures While using one-time addresses exclusively highly reduces security risks because of preventing address reuse, it still allows for constructing a deterministic transactional graph that could be used for statistical analysis with sufficient accuracy. For that reason a second component for fully confidential transacting is necessary: ring signatures. Those allow to break the link between sender and receiver. As such the only public information left is the transaction amount. The generation of such a signature can be divided into 4 steps:

Generation Sender chooses the corresponding private key x for the UTxO it wants to spend and computes the corresponding public key $P = xG$. The sender also computes the key image $I = \text{Hp}(P)$, which will be used to prevent reusing the same x in another transaction. Here, Hp is a deterministic, not necessarily cryptographically sound hash function that takes an *ed25519* point and returns a *ed25519* point. Below, I present relevant blocs of code.

```
func hashPointToScalar(point *edwards25519.Point) (*edwards25519.Scalar, error) {
    // Bytes resulting from performing sha256 on the point
    rBytes, err := crypto.HashData(point.Bytes())
    if err != nil {
        return nil, err
    }
    return edwards25519.NewScalar().SetBytesWithClamping(rBytes)
}

func HashPoint(p *edwards25519.Point) *edwards25519.Point {
    // Compute sha256 of point and convert it to scalar with clamping -> deterministic
    x, _ := hashPointToScalar(p)
    // Return xP where x = Hs(P)
    return edwards25519.NewIdentityPoint().ScalarMult(x, p)
}
func KeyImage(addr Address, dest OneTimeAddress) (x *edwards25519.Scalar, img *edwards25519
    // Compute the txn private key x to use in the key image computation
    var err error
    if x, err = addr.ComputePrivateKey(dest); err != nil {
        panic(err)
    }
    img = new(edwards25519.Point).ScalarMult(x, HashPoint(dest.P))
    return
}
Copy
```

Signature Sender generates a one-time ring signature of size N with a non-interactive zero-knowledge proof. He/she selects a subset of UTxOs with identical amounts. He now has N UTXOs each with a corresponding key P_i. The sender then mixes his/hers true UTxO into the set of decoys $\{P_i \mid i \in [0, N]\}$, where $P_s;\ s \in [0, N]$ is the real public key.

```
func ShuffleAndAdd[T any](addition T, array []T) (pos int, res []T) {
    // Seed the random function
    rand.Seed(time.Now().UnixNano())
    // Shuffle incoming array to get more uniform txn distribution
    rand.Shuffle(len(array), func(i, j int) { array[i], array[j] = array[j], array[i] })
    // Randomly choose the position of the true txn
    pos = rand.Intn(len(array))

    // Add the txn to the required position in the array
    res = append(res, array[:pos]...)
    res = append(res, addition)
    res = append(res, array[pos:]...)
    return
}

truePos, txns := utility.ShuffleAndAdd(realTxn, decoyTxns)
Copy
```

The sender now picks random sets $\{q_i \mid i \in [0, N]\}$ and $\{w_i \mid i \in [0, N], w_s = 0\}$ each of valid *ed25519* scalars, and computes the sets L_i, R_i:

$$\{L_i \mid L_i = q_i \cdot G + w_i \cdot P_i\}$$

$$\{Ri \mid R_i = q_i \cdot \mathrm{Hp}(P_i) + w_i \cdot I\}$$

```
q := make([]*edwards25519.Scalar, n)
    for i := 0; i < n; i++ {
        q[i], _ = randomScalar()
    }

    // No need for w_i where i == s; s -> our priv key
    w := make([]*edwards25519.Scalar, n)
    for i := 0; i < n; i++ {
        // Zero our w co-efficient for our key
        if i == truePos {
            w[i] = edwards25519.NewScalar()
        } else {
            w[i], _ = randomScalar()
        }
    }

    L := make([]*edwards25519.Point, n)
    for i := 0; i < n; i++ {

        L[i] = new(edwards25519.Point).Add(
            new(edwards25519.Point).ScalarBaseMult(q[i]),
            new(edwards25519.Point).ScalarMult(w[i], txns[i].Keypair.P),
        )
    }
    R := make([]*edwards25519.Point, n)
    for i := 0; i < n; i++ {
        R[i] = new(edwards25519.Point).Add(
            new(edwards25519.Point).ScalarMult(q[i], HashPoint(txns[i].Keypair.P)),
            new(edwards25519.Point).ScalarMult(w[i], I),
        )
    }
Copy
```

Now using previously computed values, we calculate the challenge

$$\gamma = sha256(m, L_1, ..., L_n, R_1, ..., R_n)$$

```
// ComputeChallenge is used to compute c.
// c is used in computing the challenge for the real txn
// in the ring set
func ComputeChallenge(message []byte, L []*edwards25519.Point, R []*edwards25519.Point) (*ed
    var buffer bytes.Buffer

    // Write message (utxo in bytes representation) to buffer
    buffer.Write(message)

    // Write consecutive L_i to buffer
    for _, Li := range L {
        buffer.Write(Li.Bytes())
    }
    // Write consecutive R_i to buffer
    for _, Ri := range R {
        buffer.Write(Ri.Bytes())
    }

    // Calculate c = Hs(m, L_0, .., L_n, R_0, .., R_n)
    if hash, err := crypto.HashData(buffer.Bytes()); err != nil {
        return nil, err
    } else {
        return edwards25519.NewScalar().SetBytesWithClamping(hash)
    }
}
Copy
```

Finally we compute the response to our challenge:

$$\{\ c_i \mid c_i = w_i; i \neq s \mid\mid c_s = \gamma - \sum_{i=0}^{n} c_i \quad mod\ l; i = s\ \}$$

$$\{\ r_i \mid r_i = q_i; i \neq s \mid\mid r_i = q_s - c_s \cdot x \quad mod\ l; i = s\ \}$$

```
// Compute the components for our response
    l := make([]*edwards25519.Scalar, n)
    // Prepare component to update sum as we compute consecutive values
    sumCi := edwards25519.NewScalar()

    for i := 0; i < n; i++ {
        // Skip computing for true utxo until we have every other
        if i != truePos {
            l[i] = w[i]
            sumCi = edwards25519.NewScalar().Add(sumCi, w[i])
        }
    }
    // Compute value for true utxo. Since the addition is modular it's unrecoverable
    l[truePos] = edwards25519.NewScalar().Subtract(c, sumCi)

    r := make([]*edwards25519.Scalar, n)
    for i := 0; i < n; i++ {
        if i == truePos {
            r[i] = edwards25519.NewScalar().Subtract(
                q[i],
                edwards25519.NewScalar().Multiply(l[i], x),
            )
        } else {
            r[i] = q[i]
        }
    }
Copy
```

The signature is then:

$$\sigma = (I, c_1, ..., c_n, r_1, ..., r_n)$$

```
type RingSignature struct {
    // Utxos for which the ring sig is computed
    Utxos []Utxo
    // Key image in byte representation
    Image []byte
    // Challenges in byte representation
    C [][]byte
    // Responses in byte representation
    R [][]byte
}
Copy
```

Verification The verifier performs the inverse transform to get L'_i, R'_i values:

$$\{\ L'_i = r_i \cdot G + c_i \cdot P_i\ \}$$

$$\{\ R'_i = r_i \cdot H_p(P_i) + c_i \cdot I\}$$

A final check is performed:

$$\sum_{i=0}^{n} c_i = \text{sha256}(m, L'_0, ..., L'_n, R'_0, ..., R'_n) \quad mod\ l$$

```
func (ringSig RingSignature) CheckSignatureValidity(message []byte) bool {
    // Parse from byte values
    c, r, err := ringSig.CRToScalars()
    if err != nil {
        fmt.Println(err)
        return false
    }
    I, err := ringSig.ImageToPoint()
    if err != nil {
        fmt.Println(err)
        return false
    }
    // Prepare L, R arrays and scalar for sum of c values
    L, R := make([]*edwards25519.Point, len(c)), make([]*edwards25519.Point, len(c))
    sumCi := edwards25519.NewScalar()
    for i := 0; i < len(c); i++ {
        L[i] = edwards25519.NewIdentityPoint().Add(
            edwards25519.NewIdentityPoint().ScalarBaseMult(r[i]),
            edwards25519.NewIdentityPoint().ScalarMult(c[i], ringSig.Utxos[i].Keypair.P),
        )
        R[i] = edwards25519.NewIdentityPoint().Add(
            edwards25519.NewIdentityPoint().ScalarMult(r[i], HashPoint(ringSig.Utxos[i].Keyp
            edwards25519.NewIdentityPoint().ScalarMult(c[i], I),
        )
        sumCi = edwards25519.NewScalar().Add(sumCi, c[i])
    }

    challenge, err := ComputeChallenge(message, L, R)
    if err != nil {
        fmt.Println(err)
        return false
    }

    return challenge.Equal(sumCi) == 1
}
Copy
```

If this equality is correct that means the signer knows such a value x that at least one $P_i = xG$.

Link check The verifier checks in his set of used images that the key image I hasn't been used before to prevent double spends. If this check holds true he/she confirms the validity of the ring signature.

Components design

Chain

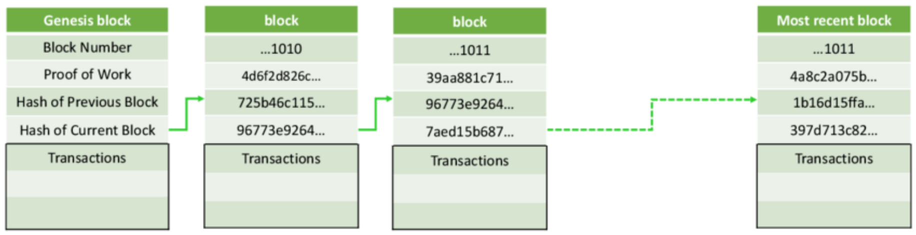
Header Header is a component of Block. It serves as a container for metadata (protocol version, time that block was created etc.), but also contains the root of the Merkle Tree(see below) holding the transactions. Based on the guarantees of this data structure the `blockHash` is in reality a hash of just the header and not the entire block.

```
// Header is the header of a block
type Header struct {
    nonce        [8]byte
    version      uint8
    blockHash    crypto.Hash
    previousHash crypto.Hash
    merkleRoot   crypto.Hash
    time         time.Time
}
Copy
```

Block The highest level container of cryptographic structures in this project is a `Block`. Blocks serve as data structures that bunch together bundles of transactions, which allows for setting "checkpoints" for the chain state. The structure definition is as follows:

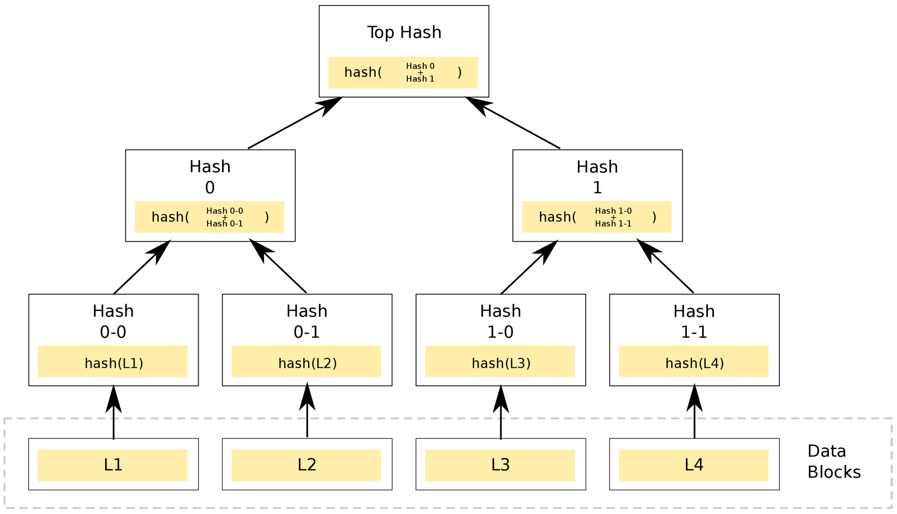
```
type Block struct {
    header       Header
    transactions crypto.MerkleTree
}
Copy
```

A block has two distinct components, a Header and transactions stored in a Merkle Tree(see below). The field allowing for data continuity is `previousHash` which is the `sha256` hash of the previous block header. As the process of finding a block hash is computationally very expensive the deeper our block is inside the chain, the harder it is to change its contents and switch the agreed upon consensus state.



ref: [@{{bitcoinblock}}]

Merkle Tree A Merkle Tree is a hash based tree-like data structure. It's a generalisation of the hash list. Each leaf is a piece of data, and non-leafs are hashes on their children. This guarantees strong integrity and



verifiability.
ref: https://en.wikipedia.org/wiki/Merkle_tree#/media/File:Hash_Tree.svg
Merkle Trees provide really computational complexity: $O(n)$ for tree traversal and storage, as well as $O(\log_2(n))$ for search. ref: https://brilliant.org/wiki/merkle-tree/.

Here is the definition of a Merkle Tree in code. As it's essentially a binary tree an array representation was chosen, given the fact it is more concise and less allocation intensive when compared to a typical linked list-like approach

```
type Node struct {
    leaf    bool
    Hash    Hash
    Content *Hashable
}

type MerkleTree struct {
    nodes []Node
    depth uint
}
// NewMerkleTree constructs a binary tree in which
// the leaves are transactions and parents are subsequent
// hashes. Since we know the depth at which the children will start
func NewMerkleTree(elements []Hashable) (*MerkleTree, error) {
    var pow float64
    elements, pow = fillElements(elements)
    // Tree size is sum(2^n for n from pow to 1)
    // so also 2^(n+1) - 1
    size := int(math.Pow(2, pow+1) - 1)
    tree := make([]Node, size)

    // Calculate the base index for level of leaves
    baseIndex := int(math.Pow(2, pow)) - 1
    // Fill all of the leaves with txs
    for i, element := range elements {
        hash, err := element.Hash()
        if err != nil {
            return nil, err
        }
        tree[baseIndex+i] = Node{leaf: true, Hash: hash, Content: &element}
    }
    // Fill all parent nodes until root
    for i := baseIndex - 1; i >= 0; i-- {
        childrenHash, err := HashData(append(tree[2*i+1].Hash, tree[2*i+2].Hash...))
        if err != nil {
            return nil, err
        }
        tree[i] = Node{leaf: false, Hash: childrenHash, Content: nil}
    }
    return &MerkleTree{tree, uint(baseIndex)}, nil
}
Copy
```

Since there is no guarantee that there will be exactly 2^n elements, a function to fill out the 2^n- `elements` leafs left is needed.

```
// fillElements rounds the number of leafs to the nearest
// power of 2 greater than the length
func fillElements(el []Hashable) ([]Hashable, float64) {
    l := len(el)
    // Calculate the nearest power of two greater than the len of our tx list
    pow := math.Ceil(math.Log2(float64(l)))
    for i := 0; i < int(math.Pow(2, pow))-l; i++ {
        el = append(el, EmptyLeaf(0))
    }
    return el, pow
}
Copy
```

When the `learncoin` node is ready to create a block, the top hash can be retrieved and embedded into the block header:

```
func (m MerkleTree) RootHash() Hash {
    return m.nodes[0].Hash
}
Copy
```

Transactions Transactions are the most important component of any cryptocurrency as they serve the purpose of transferring a value between entities on the chain. The cryptographybehind them has been extensively discussed in its own section, so only a short description of design reasoning will be provided.

```
// Transaction represents a transaction in the learncoin network. It has one ring input and
// or the output and change back to the origin address
type Transaction struct {
    UtxosIn  []Utxo
    UtxosOut []Utxo
    To       OneTimeAddress
    Sigature RingSignature
}
Copy
```

While performing a valid transaction two situations can occur:

- Input value is equal to output value
- Input value is bigger than output value

The latter situation constitutes the majority of cases. In such situation some change has to be returned to the sender. The logic for constructing a transaction has been shown below:

```
func (a Address) NewTransaction(utxosIn []Utxo, amount float32, to Address) Transaction {
    // Check if input amount == output amount.  If not we need
    // to generate change output to a new one time address generated
    // from our keys
    changeAmount := utxosIn[0].Amount - amount
    oneTimeTo, _ := to.NewDestinationAddress()

    // Create utxo out
    utxosOut := []Utxo{*NewUtxo(amount, oneTimeTo)}
    var changeUtxo *Utxo = nil
    if changeAmount != 0 {
        oneTimeChange, _ := a.NewDestinationAddress()
        changeUtxo = NewUtxo(changeAmount, oneTimeChange)
    }
    if changeUtxo != nil {
        utxosOut = append(utxosOut, *changeUtxo)
    }

    return Transaction{
        UtxosIn:  utxosIn,
        UtxosOut: utxosOut,
        To:       oneTimeTo,
        Sigature: RingSignature{},
    }
}
Copy
```

Peer-to-peer network

Node A node represents the basic building block of the peer-to-peer network. Data type definition:

```
type node struct {
    config config.NodeConfig
    logger log.Logger
    peers map[crypto.FixedHash]peer.Peer
}

type Address struct {
    Addr string
    Port string
}

type NodeConfig struct {
    // Connection vars
    addr     Address
    connType string
    // Protocol vars
    version string
    id      crypto.Hash
}
Copy
```

Each node in the network has a unique identity hash generated from the timestamp at which it was first started + a random seed:

```
func generateNewIdentity() (crypto.Hash, error) {
    var d []byte
    nonce := make([]byte, 8)
    if d, err := time.Now().MarshalText(); err != nil {
        return d, err
    }
    binary.BigEndian.PutUint64(nonce, uint64(rand.Int63()))
    return crypto.HashData(append(nonce, d...))
}
Copy
```

They hold a list of `Peers` which is a representation of a neighbour in the p2p network. Each node runs a separate inbound and outbound connection handler in concurrent fashion via go-routines.

Peer Peer represents a neighbour in the network.

```
type peer struct {
    // Connection to the peer. From doc:
    // Multiple goroutines may invoke methods on a Conn simultaneously.
    // So no mutex needed
    conn net.Conn

    // Logger dedicated to the peer
    logger log.Logger

    // Safe for concurrent access, set at creation and won't change
    id      crypto.FixedHash
    addr    config.Address
    inbound bool

    // Stats that arrive from node with the version flag
    alive bool
}
Copy
```

It launches an inHandler and an outHandler and allows for concurrent message processing.

Message Every message in the p2p network is hard-coded which allows for a machine-like state. The simplest are `Ping` and `Pong`, which allow for a quick connection health check.
Then there are `Version` and `Verack` which are sent while first connecting to a new node and allow for information exchange (version, unique node id, node address).
`Addr` and `GetAddr` are used to exchange information about `Peers` and in-turn allow the network to grow and create new connections (a gossip protocol).

Simulation output

A binary simulating chain activity has been provided to verify that the cryptographic component works properly. I will discuss its output below and explain what each output line means:

Program start

```
====== learncoin chain simulation ======
 This binary will generate a simulated
     chain state and perform random
      transactions every 2 seconds
========================================
====== Generating new chain simulation ======

Randomized 1000 addresses...
Randomized 150000 utxos for those addresses...
Copy
```

In this step the program first generates 1000 addresses (`{a, b}; {A,B} keypairs`). Then it creates a set of 150 000 UTxOs and assigns each to a one-time address generated from the previously generated addresses. It will use them to perform random transactions in the network.

Creating random transaction

First, the program chooses a random address from the 1000 generated addresses. Then it scans the entire UTxO set to check for One-time confidential addresses that were generated from its public keyset. It then picks a random UTXO from the ones it found, and looks for others with identical amounts that do not originate from his address. It then creates a ring signature that signs the transactions byte format (the output is printed in the terminal for the viewer). I then perform a check of signature with the right message first, and then with a fake message.

```
==== Simulating transaction ====
Found 165 linked utxos for addr: lrn12X4Uek2oLzxwyCCPJVvgHdSNwb2hcd3N3aAKBRbJYXNVWeZRTV3ywTj
Picked random destination address...
Created new transaction...
Computing ring signature for transaction with:
  Real utxo:    {"Amount":22.48,"Keypair":{"P":"J1jKzNdPYXmbzKSpoQkn5rshNfya0PShUGZRuMB/aFk=
  Decoy utxo 0: {"Amount":22.48,"Keypair":{"P":"jf7t+Z4JXx7kmyNjibzKl78W4+cpaPphx3VKBmFM2MU=
  Decoy utxo 1: {"Amount":22.48,"Keypair":{"P":"Re7LIAmxwD5yEtm9eIiiQrac1roqJbT07blsb56aB+k=
  Decoy utxo 2: {"Amount":22.48,"Keypair":{"P":"kMUc2q069+huvFHEf99JKcTDQe30mVnmKaITNqyWbX4=
  Decoy utxo 3: {"Amount":22.48,"Keypair":{"P":"n43+NOsnpJso4td1qKK0DHLxtc/g4rMPkpRTs0W/+T0=
  Decoy utxo 4: {"Amount":22.48,"Keypair":{"P":"FGLfl3tVL9zcFQ+q4l5+fbOKe8VLc2tI+buOcLAR1vE=
  Decoy utxo 5: {"Amount":22.48,"Keypair":{"P":"fzxVjg8jbtl2BZr71DfPr02LI722Tc13xQIEXOrSc7U=
  Decoy utxo 6: {"Amount":22.48,"Keypair":{"P":"suJ19E9QE/hv735RmbzTUnmZSX6MyMWx/D4NHFucpNQ=

Signed byte representation of transaction
Ring signature validation:
  Transaction:  ok
  Fake message: x
Copy
```

Here is the code responsible for simulating this behaviour for clarity:

```
// Message is the byte representation of our txn
message := txn.Bytes()
```

```
// Sign the message with trueUtxo+decoyUtxos
ringSig := addr.NewRingSignature(*trueUtxo, decoyUtxos, message)
fmt.Println("\nSigned byte representation of transaction")
fmt.Printf("%s:\n", color.BlueString("Ring signature validation"))
fmt.Printf("  Transaction:  %v\n", trueFalse[ringSig.CheckSignatureValidity(message)])
fmt.Printf("  Fake message: %v\n", trueFalse[ringSig.CheckSignatureValidity([]byte("Fake"))]

// Assign the ring signature to our txns
txn.Sigature = ringSig
// Append of txn to the used key images set to prevent double spending
sim.keyImages = append(sim.keyImages, ringSig.Image)
Copy
```

Blocks

Every saved transaction is added to the mempool (storage for valid transactions not yet added to blocks). When there are more than 2 transactions in it, there is a 50% change we'll create a new block. Seen here is the output of creating a new block and adding it to the chain. In a real world scenario with nodes that are connected by a network the chain would then be shared between all participants. The current chain length is printed, and then the mempool is cleared waiting for new transactions.

```
====== Constructing block from transactions ======
Added block to chain
header: {
  previous_hash: aa9a30f7c292dfd1edfc92edd8f2c28850616c4aff79fea3d15d923db1125ae6,
  merkle_root:   0d51ab7596a02a1420b7798327af2e7a9f4a3f45eaa08c6e7ff9a5e45f1f4753,
  hash:          042648a6a4f88b23ad000545af0fc07f61219c43f3aa18e593b196df02cb8e46,
  time:          2022-01-02 11:49:50.36373 +0100 CET m=+123.708738126
}
txns:
0: {false 0d51ab7596a02a1420b7798327af2e7a9f4a3f45eaa08c6e7ff9a5e45f1f4753 <nil>}
 1: {false d757a2904896aaafee46d3708ff3319dcf35fa8f17b29b5934483aaf67e7b32b <nil>}
  2: {false b703a2266eff5b1662fa539250874d854c2350c7aa38d83a1967bf4736bf41c7 <nil>}
   3: {true 67018ac0bc7b744927c4f76313ca08013663a9ecf4d7598b4bf1cf10f3ac7744 0x14000010660}
   4: {true 5f7ce2290783ac85cff8d9f615d2685a78e8db7f1a9344fbdbbf56c193bd79b6 0x14000010660}
   5: {true c5be673165e0bbf3111a8f49bf185baaa3e5624626ee273258f78c242a2fda66 0x14000010660}
   6: {true  0x14000010660}


Chain length: 4
Clearing mempool
Copy
```

This sequence of actions is performed in an infinite for loop with a 2 second interval.

Summary

During the implementation process it became clear that while the general idea of cryptocurrencies is straightforward, there is a significant amount of intricacy related to the specific implementation of components.

- Peer-to-peer networking turned out to be especially challenging because of the sheer number of edge cases related to synchronising known neighbours.
- From a cryptographic standpoint despite operations on `Curve25519` begin clearly defined in RFC 8032 (the RFC that created this Elliptic curve cryptography ECC standard), each implementation has its own specific encoding for public keys (points on the curve). This turned out to be a challenge and led to multiple code refactors in the crypto module while trying to make the standard library cooperate with the `edwards25519` library. While the cryptographic module began with the idea of only using the standard library, at the end the whole implementation ended up using the `edwards25519` library as the design provided with the `Go` standard library was so divergent from the RFC definition that they were unable to work together.

Further development

There is plenty of ways to further improve and expand this project. Propositions will be listed from most critical to least:

- Connect the networking layer with the cryptographic layer to achieve a fully working cryptocurrency. This would entail expanding the networking layer, and adding a proof-of-work component.
- Reduce the size of encoded signatures and transactions by using more efficient encoding and improving the math behind the signatures themselves.
- Add Pedersen commitments ref: https://research.nccgroup.com/2021/06/15/on-the-use-of-pedersen-commitments-for-confidential-payments/ to completely blind transaction amounts, which would greatly improve the un-traceability by increasing the possible sender ringsize.