.:Projekt MilaCoin:.

Projektarbeit zum Kurs: **Einführung in das Programmieren mit Python**

Sommersemester 2022

Markus Reichold

Lehrstuhl für medizinische Zellbiologie

Fakultät für Biologie und vorklinische Medizin

Universität Regensburg

26.08.2022

Inhalt

1. Einleitung

1.1 Bitcoin und Blockchain-Technologie

Die Idee von Bitcoin wurde im Jahr 2008 in Form eines White Papers veröffentlicht und bereits 2009 wurde der erste Block generiert. Bitcoin entstand in einer Zeit, in der die Weltwirtschaft eine tiefe Krise erlebte und das Vertrauen in die herkömmlichen „Fiat"-Währungen einen Tiefpunkt erreichte. Ziel von Bitcoin war es, ein digitales Bezahlsystem zu schaffen, dass ohne eine vertrauenswürdige dritte Partei auskommt. Werden Euro von einer zur nächsten Person überwiesen, dann ist diese vertrauenswürdige Partei in aller Regel eine Bank, die sicherstellt, dass das Geld beim Konto des Senders abgezogen und auf dem Konto des Empfängers gutgeschrieben wird. Da das Vertrauen in die Banken zu diesem Zeitpunkt sehr gering war, wollte man mit Bitcoin die Rolle der Banken im Geldwesen durch mathematische und kryptographische Algorithmen ersetzen. Bitcoin wurde unter dem Pseudonym „Satoshi Nakamoto" veröffentlicht. Bis heute weiß niemand, wer diese Person ist und ob es sich vielleicht sogar um eine ganze Gruppe von Entwicklern handelt. Man vermutet aber, dass Bitcoin aus der Cypherpunk-Bewegung hervorgegangen ist. Dabei handelte es sich um mehrere Personen, die schon sehr früh die Probleme der Digitalisierung erkannten und sich deshalb sehr engagiert für den digitalen Datenschutz einsetzten.

Eine große Herausforderung bei der Schaffung von digitalem Geld war und ist nach wie vor das „Double-Spending-Problem". Wie bei allen digitalen Gütern kann sehr einfach eine verlustfreie Kopie angefertigt werden. Wie wird also verhindert, dass digitales Geld nicht einfach kopiert und damit unkontrolliert vermehrt werden kann? Als Lösung dafür hatte Nakamoto die Blockchain-Technologie erfunden, die heute in vielen Bereichen Anwendung findet. Bei der Bitcoin-Blockchain handelt es sich im Prinzip um ein Kassenbuch (engl. Ledger), in dem alle Transaktionen zwischen Personen niedergeschrieben werden. Ein Bitcoin-Block ist vergleichbar mit einer Seite in diesem Buch: Wenn die Seite voll ist, dann wird sie umgeblättert und eine neue Seite begonnen. Das Besondere ist aber, dass die vollen Seiten über kryptographische Algorithmen derart geschützt werden, dass zwar das Lesen noch funktioniert, aber ein Verändern nicht mehr möglich ist. Bitcoins existieren demnach nur aufgrund von Einträgen in diesem Kassenbuch und nicht als Dateien, die man beispielsweise auf den Rechner herunterlädt. Will man den Kontostand einer Person ermitteln, dann muss immer die gesamte Transaktionshistorie in der Blockchain durchsucht werden.

Bitcoin läuft als dezentrales Peer-to-peer Netzwerk. Einzelne Rechner in diesem Netzwerk werden als „Nodes" bezeichnet. Alle diese Nodes sind gleichwertig, es gibt also keinen zentralen Rechner, der sie koordiniert. Jede „Full-Node" im Netzwerk speichert die gesamte Blockchain. Das sind momentan (August 2022) 751.000 Blöcke, die etwa 419 GB groß sind. Hier sieht man bereits ein Problem, das mit der Blockchain-Technologie einhergeht, nämlich die Datenredundanz. Versucht man bei Datenbankanwendungen genau diese zu minimieren, ist sie bei Bitcoin ein unvermeidbares Übel. Ferner muss für den Kontostand einer jeden Person immer die gesamte Bockchain durchsucht werden, was mit wachsender Größe immer langsamer wird.

Für dieses Projekt sollten die Grundprinzipien von Bitcoin und der Bitcoin-Blockchain in simplifizierter Form nachgeahmt werden. Der Fokus lag hierbei auf der Durchführung von Transaktionen und den kryptographischen Verfahren, die dafür nötig sind. Die Software wurde aus Zeitgründen nicht als Peer-to-peer Netzwerk umgesetzt. Stattdessen wurde ein rudimentäres Login-System implementiert, um einzelne Benutzer zu unterscheiden. Die Software ist nicht dafür gedacht, eine weitere Kryptowährung ins Leben zu rufen (momentan gibt es knapp über 20.000). Dafür ist Python als Programmiersprache auch nicht geeignet, da es einige relevante Sicherheitslücken aufweist bzw. gegenüber bestimmten Angriffen verwundbar ist. Es sollen vielmehr die Grundprinzipien von Bitcoin und Kryptowährungen im Allgemeinen vermittelt werden.

1.2 Kryptographische Grundlagen

Die nachfolgenden kryptographischen Grundlagen, die für das Verständnis dieses Projekts benötigt werden, sind **kopierte (!!!)** und modifizierte Versionen der jeweils unten angegebenen Quellen. Meist wurden die Einträge gekürzt und um projektrelevante Aspekte erweitert.

1.2.1 Base58 Kodierung

Verwendetes Modul: base58

Base58 beschreibt ein Verfahren aus dem Computerbereich zur Kodierung von positiven ganzen Zahlen. Wie das Hexadezimalsystem eine Basis von 16 und das binäre System eine Basis von 2 besitzt, besitzen base58 codierte Zahlen eine Basis von 58. Das Alphabet von Base58 ergibt sich aus dem der (eher üblichen) Base62-Kodierung durch Weglassen der vier Zeichen 0 (Null), O (großes o), I (großes i) und l (kleines L). Gegenüber dem Alphabet der Base64-Kodierung fehlen außerdem die Zeichen + (Plus) und / (Schrägstrich). Das resultierende Alphabet der Länge 58 lautet `123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz`, es besteht somit nur aus verwechslungsfreien alphanumerischen Zeichen. Verwendung findet die Base58-Kodierung typischerweise dort, wo lange Integer-Zahlen in kürzere Zeichenketten umgewandelt werden sollen und eine verwechslungsfreie Erkennung der Zeichenkette sichergestellt sein soll, zum Beispiel bei Bitcoin-Adressen.

(abgeändert aus: https://de.wikipedia.org/wiki/Base58)

1.2.2 Hashfunktion und Hashwert

Verwendetes Modul: hashlib

Eine kryptografische Hashfunktion (Streuwertfunktion) ist eine Funktion mit besonderen, für die Kryptographie erwünschten Eigenschaften. Eine Hashfunktion erzeugt aus einem Eingabewert (z. B. eine Nachricht oder eine Datei) einen meist ganzzahligen Ausgabewert (Hash oder Hashwert) in einem gegebenen Wertebereich auf eine regellos erscheinende Weise. Folgende Merkmale besitzt die Hashfunktion:

- Sie kann jede beliebige Bit- oder Bytefolge verarbeiten und erzeugt daraus einen Hash von fixer Länge (typisch 256 Bit, z.B. bei der verwendeten SHA256-Funktion).
- Sie ist stark kollisionsresistent: Es ist praktisch nicht möglich (d. h. erfordert einen unrealistisch hohen Rechenaufwand), zwei unterschiedliche Eingabewerte zu finden, die den gleichen Hash ergeben.
- Sie ist eine Einwegfunktion: Es ist praktisch nicht möglich, einen Eingabewert zu finden, der einen vorgegebenen Hash ergibt.
- Kleinste Änderungen am Eingabewert (z.B. einzelne Zeichen vertauschen oder austauschen) führen unabhängig von der Länge der Daten zu einem vollkommen anderen Hashwert.

Solche Hashfunktionen werden vor allem zur Integritätsprüfung von Dateien oder Nachrichten eingesetzt. Dafür wird die Funktion auf die zu prüfende Datei angewendet und mit einem bekannten früheren Hash verglichen. Weicht der neue Hash davon ab, wurde die Datei verändert. Weiter dienen kryptografische Hashfunktionen zur sicheren Speicherung von Passwörtern und digitalen Signaturen. Wenn ein System ein Passwort oder eine Signatur prüft, vergleicht es dessen bzw. deren Hash mit einem gespeicherten Hash. Stimmen beide Werte überein, ist das Passwort bzw. die Signatur richtig. So kann vermieden werden, das Passwort oder die Signatur jemals im Klartext abzuspeichern, was sie verwundbar machen würde.

(abgeändert aus: https://de.wikipedia.org/wiki/Kryptographische_Hashfunktion)

1.2.3 Elliptische-Kurven-Kryptografie (Elliptic Curve Cryptography, ECC)

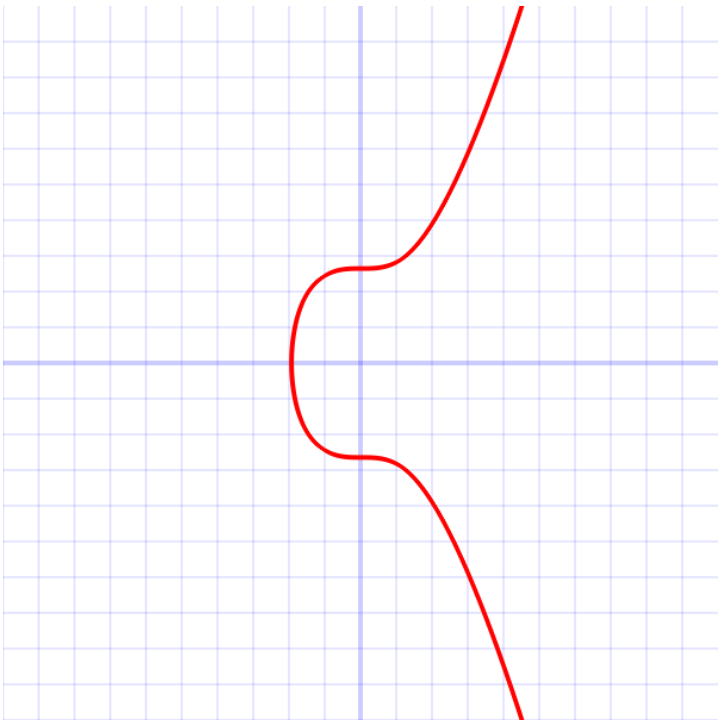
Verwendetes Modul: ecdsa (Elliptic Curve Digital Signature Algorithm)

Elliptische Kurven-Kryptografie ist ein Public-Key-Verfahren, das auf der Berechnung von elliptischen Kurven basiert. Es wird verwendet, um schneller kleine und effiziente Verschlüsselungs-Keys zu erstellen. ECC berechnet Keys mithilfe von Operationen auf Basis von elliptischen Kurven anstelle der sonst üblichen Zuhilfenahme von sehr großen Primzahlen. Nach Ansicht einiger Forscher bietet ECC bereits mit einem 164 Bit langen Schlüssel eine vergleichbare Sicherheit, für die ansonsten beim Einsatz anderer Verfahren (z.B. RSA) ein 1024 Bit langer Schlüssel benötigt wird.

Eine elliptische Kurve ist nicht gleichzusetzen mit einer ovalen Ellipse. Stattdessen wird sie in der Regel als kurvige Linie dargestellt, die zwei Achsen schneidet. ECC basiert auf den besonderen Eigenschaften von mathematischen Gruppen (Werte, die verwendet werden, um aus zwei Zahlen aus einer Gruppe eine dritte zu berechnen). Diese bestehen aus den Punkten, an denen die Linie die Achsen schneidet. Wenn ein Punkt auf der Kurve mit einer Zahl multipliziert wird, erhält man einen anderen Punkt auf der Kurve. Es ist praktisch unmöglich, herauszufinden, welche Zahl dabei genutzt wurde, selbst wenn der verwendete Punkt und das Ergebnis bekannt sind. Berechnungen auf der Basis von elliptischen Kurven haben deswegen einen wichtigen Vorteil beim Einsatz von Verschlüsselungstechnik: **Sie sind relativ leicht durchzuführen, aber nur sehr schwierig zurück zu berechnen**.

(abgeändert aus: https://www.computerweekly.com/de/definition/Elliptische-Kurven-Kryptografie-Elliptic-Curve-Cryptography-ECC)

Wie bei Bitcoin wird in dieser Projektarbeit die Kurve Secp256k1 verwendet. Die Formel für die Kurve lautet $y^2 = x^3 + 7$. Nachfolgend eine Abbildung der Kurve im Zahlenbereich der reellen Zahlen $\mathbb{R}$.



(Bildquelle: https://en.bitcoin.it/wiki/Secp256k1)

1.2.4 Public-Key-Verschlüsselungsverfahren

Das „asymmetrische Kryptosystem“ oder „Public-Key-Kryptosystem“ ist ein kryptographisches Verfahren, bei dem im Gegensatz zu einem symmetrischen Kryptosystem die kommunizierenden Parteien **keinen gemeinsamen geheimen Schlüssel** zu kennen brauchen. Jeder Benutzer erzeugt sein eigenes Schlüsselpaar, das aus einem geheimen Teil (**privater Schlüssel**) und einem nicht geheimen Teil (**öffentlicher Schlüssel**) besteht. Der öffentliche Schlüssel ermöglicht es jedem, Daten für den Besitzer des privaten Schlüssels zu verschlüsseln, dessen digitale Signaturen zu prüfen oder ihn zu authentifizieren. Der private Schlüssel ermöglicht es seinem Besitzer, mit dem öffentlichen Schlüssel verschlüsselte Daten zu entschlüsseln, digitale Signaturen zu erzeugen oder sich zu authentisieren. **Für dieses Projekt wird der private Schlüssel dafür benutzt, eine digitale Signatur zu erstellen und der öffentliche Schlüssel um diese Signatur zu authentisieren.**

Die theoretische Grundlage für asymmetrische Kryptosysteme sind **Falltürfunktionen**, also Funktionen, die leicht zu berechnen, aber ohne ein Geheimnis (die „Falltür“) praktisch unmöglich zu invertieren sind. Der öffentliche Schlüssel ist dann eine Beschreibung der Funktion, der private Schlüssel ist die Falltür. Eine Voraussetzung ist natürlich, dass der private Schlüssel aus dem öffentlichen nicht berechnet werden kann. Damit das Kryptosystem verwendet werden kann, muss der öffentliche Schlüssel dem Kommunikationspartner bekannt sein.

Der entscheidende Vorteil von asymmetrischen Verfahren ist, dass sie das Schlüssel-verteilungsproblem vermindern. Bei symmetrischen Verfahren muss vor der Verwendung ein Schlüssel über einen sicheren, d. h. abhörsicheren und manipulationsgeschützten Kanal ausgetauscht werden. Da der öffentliche Schlüssel nicht geheim ist, braucht bei asymmetrischen Verfahren der Kanal nicht abhörsicher zu sein. Wichtig ist nur, dass der öffentliche Schlüssel dem Inhaber des dazugehörigen privaten Schlüssels zweifelsfrei zugeordnet werden kann.

(abgeändert aus: https://de.wikipedia.org/wiki/Asymmetrisches_Kryptosystem)

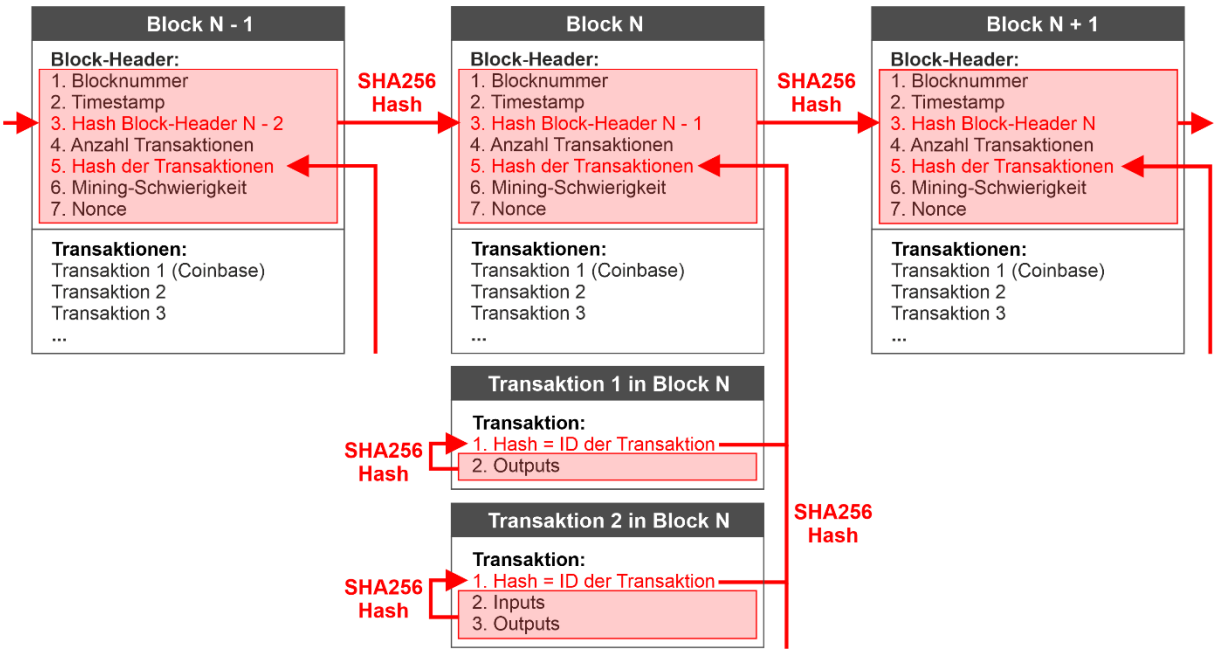
2. Das Projekt MilaCoin

2.1 Die Blockchain

Eine Blockchain ist eine Verkettung von individuellen Blöcken. In dieser Software werden Blöcke dauerhaft in Textdateien gespeichert. Zusätzlich werden sie mitsamt den enthaltenen Transaktionen zur Laufzeit des Programms in Form von Block-Objekten im Speicher gehalten. Ein Block besteht aus dem **Block-Header**, welcher Informationen über den Block enthält und **Transaktionen**. Eine Transaktion ist vergleichbar mit einer Überweisung, bei der Guthaben von einem Bankkonto zum anderen transferiert wird. In dieser Version der Software kann ein Block beliebig viele Transaktionen enthalten, bei Bitcoin ist die Blockgröße auf 1 Mb festgelegt.

Die **Verkettung der Blöcke** erfolgt über Hashwerte. Wie bei Bitcoin wird überwiegend der SHA256-Hash verwendet. Für die Verkettung wird der Hash des Block-Headers N - 1 generiert und in den Header des nächsten Blocks N geschrieben. Wird bei Block N - 1 in der Block-Datei etwas geändert/manipuliert, ändert sich der Hash des Blocks (eigentl. des Block-Headers) und stimmt nicht mehr mit dem ursprünglich generierten Hashwert im Header von Block N überein. Der Block bzw. die gesamte Blockchain ist damit nicht mehr valide. Der Hashwert wird aus einem String generiert, bei dem die Daten des Block-Headers ohne Leerzeichen aneinandergereiht werden.

In ähnlicher Weise werden auch **Transaktionen** gehandhabt. Aus den (meisten) Daten einer Transaktion wird der SHA256-Hash für die jeweilige Transaktion generiert. Dieser ist eindeutig und identifiziert die Transaktion (ID). Aus den Hashwerten aller Transaktionen in einem Block wird wiederum ein SHA256-Hash generiert. Dieser wird in den Block-Header geschrieben (zusammen mit der Anzahl der Transaktionen in diesem Block). Auf diese Weise sind Transaktionen gleich mehrfach geschützt. Wird eine Transaktion manipuliert, ändert das deren Hashwert. Dieser Hashwert geht in die Berechnung des Hashwertes aller Transaktionen im Block ein und ändert damit auch diesen. Der Hashwert aller Transaktionen eines Blocks geht schließlich in die Berechnung des Hashes des Block-Headers mit ein, wodurch die Blockchain unterbrochen wird.



Die Blockchain wird über die Klasse „Blockchain“ verwaltet (Modul bcm.py). Nachfolgend einige wichtige Methoden der Klasse mit deren Aufgaben:

- **load_mempool()** = staticmethod: Methode lädt alle aktuell vorhandenen Transaktionen im Mempool in eine Liste aus Transaktions-Objekten.
- **get_nonce()** = staticmethod: Methode ist für das Minen eines Blocks wichtig und dient dem Finden des richtigen Nonce für eine vorgegebene Mining-Schwierigkeit.
- **load_block_in_bc():** Methode um einen einzelnen Block in die Blockchain zu laden.
- **load_bc():** Die Methode lädt alle Blöcke in die Blockchain inklusive deren Transaktionen. Sie beinhaltet die Funktionen load_block() und load_tx(), mit denen überprüft wird, ob die Daten aus den Textdateien formal korrekt sind (ob z.B. die Anzahl der Transaktionen im Block-Header tatsächlich eine Integer Zahl ist). Die Methode gibt aus, ob und an welcher Stelle ein Fehler bzw. eine Manipulation der Blockchain gefunden wurde. Wird ein Fehler gefunden, bricht die Software ab.
- **validate_bc():** Über diese Methode wird überprüft, ob die Hashwerte in der Blockchain korrekt sind. Es wird aber nicht kontrolliert, ob die Signaturen der Transaktionen valide sind! Folgende Hashwerte werden überprüft:
    1. Der (berechnete) Hashwert des Blocks muss dem Hashwert im Header des nächsten Blocks entsprechen
    2. Der (berechnete) Hashwert einer Transaktion muss der Transaktions-ID entsprechen
    3. Der (berechnete) Hashwert aller Transaktionen in einem Block muss dem Hashwert der Transaktionen im Block-Header entsprechen
- **validate_tx():** Methode validiert die Signaturen von Transaktionen. Das wird zum einen gemacht, wenn Transaktionen in den Mempool geschrieben werden und zum anderen, wenn sie aus dem Mempool in Blöcke transferiert werden. Konkret werden zwei Prozeduren durchgeführt:
    1. Wird überprüft, ob der öffentliche Schlüssel eines Transaktions-Inputs in die Adresse des zugehörigen Outputs (UTXOs) der vorhergehenden Transaktion umgewandelt werden kann.
    2. Wird die Signatur der Transaktion überprüft. Dafür werden die Transaktions-ID, der öffentliche Schlüssel und die Signatur benötigt. Das beweist, dass der Sender die privaten Schlüssel der UTXOs für eine Transaktion besitzt und diese somit ausgeben darf.
- **mine_block():** Methode zum Erschaffen neuer Blöcke und damit neuen Coins. Wird später mehr im Detail besprochen.

2.2 Login-System und Benutzer

Ein einfaches (und unsicheres!) Login-System wurde implementiert, um einzelne Nutzer zu unterscheiden. Registrierte Nutzer können sich einloggen. Dabei werden die Daten des Nutzers in einem Objekt gespeichert. Nach dem Einloggen wird das Hauptmenü angezeigt, von dem aus sich der Benutzer auch wieder ausloggen kann.

Beim Anlegen (Registrieren) eines neuen Benutzers werden dessen Anmeldedaten in der Datei „credentials.txt“ gespeichert, welche sich im Ordner „user“ im Arbeitsverzeichnis befindet. Neben dem Benutzernamen (Überprüfung auf Eindeutigkeit) wird das Passwort als SHA256 verschlüsselter String gespeichert. Zeilenumbrüche („\n“) trennen die einzelnen Nutzer voneinander und als Delimiter für die Daten eines Nutzers wurde das Zeichen „|“ gewählt.

Ein Benutzer-Eintrag enthält folgende Informationen in dieser Reihenfolge:

1. **Name des Benutzers:** eindeutig (String)
2. **Passwort:** SHA256 verschlüsselter String des Passworts (String mit 64 Zeichen, SHA256 verschlüsselt, hexadezimal)

Login und Benutzer werden über die Klasse „User“ verwaltet (Modul usr.py). Das Login-System basiert auf folgender Quelle: https://stackoverflow.com/questions/59121573/python-login-and-register-system-using-text-files. Nachfolgend einige wichtige Methoden der Klasse mit deren Aufgaben:

- **user_exists():** Methode prüft, ob bei der Registrierung eines neuen Benutzers ein schon vorhandener Benutzername ausgewählt wurde. Ein Benutzername muss eindeutig sein! Groß- und Kleinschreibung wird nicht überprüft.
- **clear_user():** Methode leert das Benutzer-Objekt. Ist das Benutzer-Objekt leer, dann ist niemand eingeloggt.
- **signup():** Methode zum Registrieren eines neuen Benutzers.
- **user_authorized():** Methode überprüft, ob Name und Passwort beim Einloggen korrekt sind.
- **login():** Methode für das Einloggen.
- **logout():** Methode zum Ausloggen. Beinhaltet die Funktion clear_user() mit einer zusätzlichen Logout-Meldung.
- **is_loggedin():** Methode überprüft, ob ein Benutzer eingeloggt ist.

2.3 Blöcke

Die Daten eines Blocks werden in einer Textdatei gespeichert. Der Name dieser Datei ist die Blocknummer mit der Endung .bl (z.B. 42.bl). Der Ordner für die Block-Dateien liegt im Arbeitsverzeichnis und heißt „blocks“. Jeder Block besteht aus einem **Block-Header** (erste Zeile einer Block-Datei) und einer unterschiedlichen Anzahl an **Transaktionen**. Die Informationen werden in der Textdatei aus didaktischen Gründen nicht binär gespeichert, sondern uft-8 codiert und damit lesbar. Als Delimiter zum Trennen der Daten des Block-Headers wurde das Zeichen „|“ gewählt. Das Trennen von Transaktionen ist komplexer und wird weiter unten beschrieben. Zusätzlich werden die Blockinformationen zu Laufzeit des Programms auch als Block-Objekte im Speicher gehalten.

Ein **Block-Header** enthält folgende Informationen in dieser Reihenfolge:

1. **Blocknummer:** beginnt bei 0 (Integer)
2. **Timestamp:** Zeitpunkts als das Mining des Blocks begonnen wurde (Float)
3. **Hash des vorherigen Block-Headers:** aus dem vorhergehenden Block -> Verkettung der Blockchain! (String mit 64 Zeichen, SHA256 verschlüsselt, hexadezimal)
4. **Anzahl der Transaktionen:** welche im Block enthalten sind inklusive Coinbase Transaktion (Integer)
5. **Hash der Block-Transaktionen:** genauer Hash der Transaktion-Hashes (String mit 64 Zeichen, SHA256 verschlüsselt, hexadezimal)
6. **Mining-Schwierigkeit:** welche für das Minen dieses Blocks galt (Integer)
7. **Nonce**: „number used once“, Diese Zahl muss von den Minern beim Mining-Prozess gefunden werden (Integer), damit ein neuer Block entstehen kann

Blöcke werden über die Klasse „Block“ verwaltet (Modul blm.py). Wird das Programm gestartet, werden zunächst alle Blöcke aus den Textdateien in Block-Objekte geladen. Anschließend werden alle Hashwerte im Rahmen einer Validierung der Blockchain überprüft. Erst wenn diese Aktionen erfolgreich abgeschlossen sind, kann man sich als User einloggen und die Software benutzen. Nachfolgend einige wichtige Methoden der Klasse mit deren Aufgaben:

- **get_max_block()** = staticmethod: Liest anhand der Block-Dateien im Block-Ordner aus, was der momentan höchste Block ist. Diese Methode soll ersetzt werden durch eine Klassenvariable, welche bei der Erstellung eines neuen Blocks inkrementiert wird.
- **load_block():** Methode liest einen Block von einer Block-Datei aus und speichert die Daten in einem Block-Objekt. Beim Einlesen der Daten werden diese auf ein korrektes Format überprüft, jedoch nicht, ob die Angaben einen Sinn machen.
- **write_block_to_file():** Methode schreibt die Daten aus einem Block Objekt in eine Block-Datei. Der String dazu wird über die Methode get_block_string() generiert.

2.4 Wallet-Datei und -Software

2.4.1 Die Wallet-Datei

Wird ein neuer Benutzer angelegt, dann wird auch gleichzeitig eine Wallet-Datei für diesen Benutzer angelegt. Das ist eine Textdatei im Ordner „user“ im Arbeitsverzeichnis. Der Dateiname ist der Benutzername als SHA256 verschlüsselter String (User ID) mit der Dateiendung .wlt. In dieser Datei werden primär die privaten Schlüssel für Transaktionen gespeichert. Daneben werden der öffentliche Schlüssel und die Transaktions-Adressen gespeichert. Das ist redundant, da der öffentliche Schlüssel aus dem privaten Schlüssel und die Adresse wiederum aus dem öffentlichen Schlüssel generiert werden. Aus didaktischen Gründen und der einfacheren Handhabung halber wurde es jedoch beibehalten. Zeilenumbrüche („\n“) trennen Adressen voneinander und als Delimiter für die Adress-Daten wurde das Zeichen „|“ gewählt.

Ein **Wallet-Eintrag** enthält folgende Informationen in dieser Reihenfolge:

1. **Adresse:** wird sowohl für das Senden als auch für das Empfangen von Transaktionen benötigt, wird aus dem öffentlichen Schlüssel generiert (base58 codierter String)
2. **Privater Schlüssel:** wird zufällig generiert (String, hexadezimal)
3. **Öffentlicher Schlüssel:** wird aus dem privaten Schlüssel über ECC generiert (base58 codierter String)

2.4.2 Privater Schlüssel („private key“)

Der private Schlüssel ist eine Integer-Zahl, welche zufällig erzeugt wird und die feste Länge von 256 Bit (= 32 Byte) hat. Für die kryptographische Sicherheit ist es außerordentlich wichtig, dass die Zahl tatsächlich zufällig generiert wird und erfordert damit einen geeigneten Zufallsgenerator. Um den Schlüssel kompakter darzustellen und damit Speicherplatz zu sparen, wird er im Projekt immer als **hexadezimale Zahl** mit 64 Stellen kodiert (z.B.: 330e6129d5c89853fbe925d93f513c60a0c5eabb60 398a1c7b4a96e6c7ba0563).

Es ist sehr wichtig, die privaten Schlüssel für seine Coins an einem sicheren Ort aufzubewahren. Kommt eine andere Person in den Besitz dieser Schlüssel, hat sie uneingeschränkt Zugriff auf die eigenen Coins und kann sie nach Belieben ausgeben. Empfehlenswert sind Hardware-Wallets oder auch Cold-Wallets genannt (z.B. Ledger nano X oder Bitbox), in dem die Schlüssel offline aufbewahrt werden. Solche Geräte sehen zum Teil aus wie einfache USB-Sticks, tatsächlich handelt es sich dabei aber um kleine „Hochsicherheitsfestungen“ für genau solche Daten. Dadurch, dass diese Geräte von Netzwerk getrennt werden können, können sie nicht von Hackern angegriffen werden. Es empfiehlt sich auch nicht, die Coins auf den Börsen liegen zu lassen, bei denen man sie gekauft hat. Diese Börsen sind beständig Ziel von Hackern und die Geschichte hat mehrfach gezeigt, dass sie damit durchaus „erfolgreich“ waren. Generell gilt eine wichtige Grundregel in der Kryptowelt: **„Not your keys, not your coins!“**

2.4.3 Öffentliche Schlüssel („public key“)

Der öffentliche Schlüssel wird aus dem private Schlüssel mittels Elliptische-Kurven-Kryptografie erzeugt. Die **unkomprimierte Form** des Schlüssels hat eine feste Länge von 512 Bits (= 64 Bytes). Da der öffentliche Schlüssel eigentlich einen Punkt auf der elliptischen Kurve beschreibt, besteht er aus x- und y-Koordinaten mit jeweils 256 Bits. Bei der unkomrimierten Form werden diese beiden Koordinaten einfach zusammengehängt. Die unkomrimierten Form wird für dieses Projekt verwendet.

Um den Schlüssel kompakter darzustellen und damit Speicherplatz zu sparen, wird er im Projekt immer **base85** kodiert (z.B.: 4RLZccFQXJSgeha7ZskpdhaUXNXvRqw3AdN75ougzhw73CZhV8x4n6UyJhJfmcki TvbfPyd736xV3RafJaUXQoem).

Öffentlicher und privater Schlüssel sind mathematisch miteinander verbunden und damit ein Paar. Es ist einfach, den öffentlichen Schlüssel aus dem privaten Schlüssel zu berechnen, es ist jedoch mit enormen Aufwand verbunden, den privaten aus dem öffentlichen Schlüssel rückzurechnen. Eine weitere, sehr verbreitete Methode, um Schlüsselpaare zu erzeugen ist **RSA (Rivest–Shamir–Adleman)**. Die Schlüssel sind bei RSA aber im Vergleich zu ECC bei gleicher Sicherheit deutlich länger, was zu mehr Speicheraufwand führt. Außerdem verwendet Bitcoin das ECC-Verfahren.

2.4.4 Wallet-Adressen

Eine **Wallet-Adresse** wird aus dem privaten Schlüssel erzeugt. Die in diesem Projekt verwendete Methode zur Generierung von Adressen weicht von der bei Bitcoin ab und ist nur daran angelehnt. Dazu kommt noch, dass sich die Methode bei Bitcoin im Laufe der Versionen geändert hat bzw. erweitert wurde.

Wie bei Bitcoin wird aus dem privaten Schlüssel zunächst der **SHA256 Hashwert** gebildet. Aus diesem Hashwert wird bei Bitcoin wiederum ein RIPEMD160 Hash gebildet, welcher 160 Bit oder 20 Byte lang ist. Für dieses Projekt wurde aber als zweite **Hash-Funktion SHA1** gewählt, welche einen genauso langen Hashwert erzeugt und nicht das Installieren eines weiteren Moduls erfordert (in Modul hashlib). Der Sinn der zweiten Hashfunktion ist das Verkürzen des SHA256 Hashwerts. Zum weiteren Verkürzen und um nur verwechslungsfreie Zeichen zu verwenden wir der Hashwert schließlich **base58** kodiert (z.B.: 2Zkc8n3tbnYCbGnaVkT6KPww3syP).

Hier noch einmal in der Übersicht:
Öffentlicher Schlüssel -> SHA256(32bytes) -> SHA1(20bytes) -> Base58 Kodierung -> Adresse

Adressen wurden zu Beginn von Bitcoin noch in Form von Paper-Wallets verwaltet, d.h. einfach auf Papier geschrieben. Die base58 Kodierung war hierbei sehr hilfreich da 1. die Adressen dadurch kürzer wurden und 2. Zeichen, die leicht verwechselt werden können (O und 0, I und l), ausgeschlossen wurden. Die Adressen bei Bitcoin enthalten zudem noch Prüfsummen, mit welchen die korrekte Angabe einer Adresse bezüglich Format validiert wird (base58check()).

2.4.5 Die Wallet-Software

Die Wallet-Software ist ein zentraler Teil von Bitcoin, nicht nur um die Schlüssel einer Transaktion zu speichern. Die Software übernimmt vielfältige Aufgaben, von denen einige in dieser Software umgesetzt wurden. Die Wallet-Software wurde hier in Form der Klasse „Wallet“ (Modul wlm.py) implementiert. Nachfolgend einige wichtige Methoden der Klasse mit deren Aufgaben:

- **generate_keys():** Diese Funktion generiert die Schlüssel und Adressen für Transaktionen und schreibt diese in die Wallet Datei des Nutzers. Zudem werden die Informationen in temporären Variablen der Klasse gespeichert.

- **get_addr_list():** Die Methode liest alle Adressen eines Nutzers aus der Wallet-Datei aus und generiert daraus eine Liste.
- **get_keys_for_address():** Nimmt eine Adresse entgegen und sucht in der Wallet-Datei des Nutzers nach den zugehörigen Schlüsseln
- **load_user_utxos():** Eine sehr wichtige Methode! Sie liest aus der Blockchain (genauer aus dem Blockchain Objekt, nicht aus den Textdateien) alle UTXOs eines Benutzers aus und speichert diese in einer Liste. Wichtig ist, dass dabei auch die Transaktionen im Mempool durchsucht werden. Ansonsten könnten UTXOs beliebig oft ausgegeben werden, solange sie noch nicht in einen Block transferiert wurden.
- **get_user_balance():** Methode zum Berechnen des aktuellen Kontostands eines Benutzers. Dabei wird über die UTXO Liste iteriert und die Transaktionsvolumina addiert
- **receive_tx():** Diese Funktion generiert eine Adresse für den Empfang von MiCs. Die Adresse wird mit Hilfe des Moduls „pyperclip“ direkt in die Windows Zwischenablage kopiert.
- **send_tx():** Methode zum Versenden von MiC.

2.5 Transaktionen

2.5.1 Aufbau einer Transaktion

Transaktionen werden zunächst in einer Datei namens „mempool.mem“ im Ordner „mem“ im Arbeitsverzeichnis gespeichert. Beim minen eines Blocks werden die Transaktionen in die neue Block-Datei überführt und der Mempool geleert.

Die erste Transaktion eines Blocks wird automatisch beim minen des Blocks erzeugt und wird „Coinbase“ genannt. Diese Transaktion geht an den Miner des Blocks und umfasst bei Bitcoin sowohl die Belohnung für das minen des Blocks (Mining Reward) als auch die Transaktionsgebühren aller Transaktionen im Block. Transaktionsgebühren sind in dieser Software nicht umgesetzt. Coinbase-Transaktionen haben die Besonderheit, dass sie keine Inputs aufweisen (s.u.).

Aus den (meisten) Daten einer Transaktion wird ein doppelter SHA256 Hashwert generiert. Dieser dient in weiteren Versionen als eindeutige **ID der Transaktion**. Aus den Hashwerten aller Transaktionen in einem Block wird erneut ein Hashwert generiert. Dieser wird in den Block-Header in Position 5 geschrieben (Hash der Block-Transaktionen). Bei Bitcoin wird dieser Hashwert als „root-hash“ bezeichnet, weil er aus den Hashwerten der einzelnen Transaktionen in Form eines Merkle-Baums (Hash-Baum) generiert wird. Das ist in diesem Projekt aber einfacher gelöst. Die Hashwerte der Transaktionen in einem Block werden als String verkettet (ohne Leerzeichen) und davon der doppelte SHA256 Hashwert generiert.

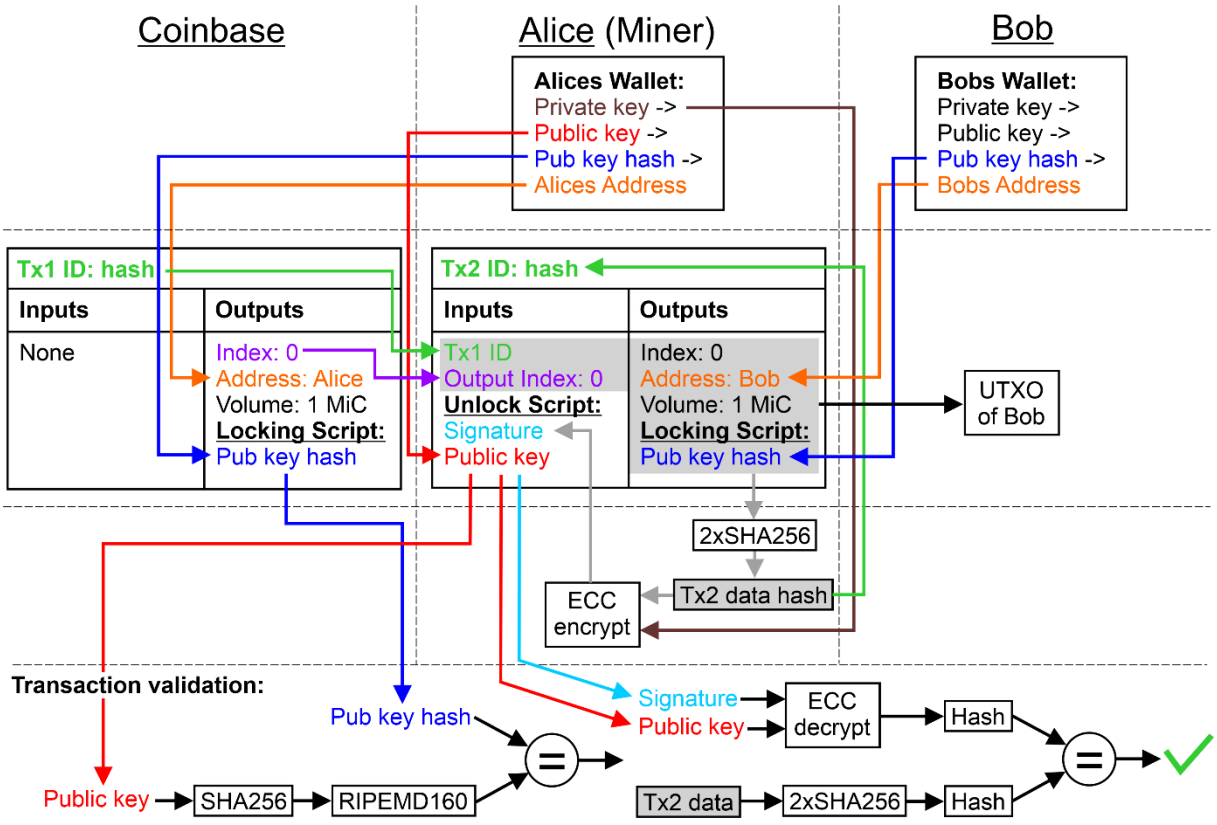
Eine **Transaktion** enthält folgende Informationen in dieser Reihenfolge:

1. **Hash der Transaktion:** ist auch gleichzeitig die eindeutige ID der Transaktion (String mit 64 Zeichen, SHA256 verschlüsselt, hexadezimal)
2. **Timestamp:** Transaktionszeitpunkt, nicht bei Bitcoin-Transaktionen vorhanden (Float)
3. **Anzahl der Inputs:** Gibt an, wie viele Inputs die Transaktion besitzt (Integer)
4. **Liste an Inputs mit:**
    - **ID der vorangegangenen Transaktion**, wo der Output als neuer Input für die aktuelle Transaktion verwendet werden soll (String mit 64 Zeichen, SHA256 verschlüsselt, hexadezimal)
    - **Output-Index der vorangegangenen Transaktion** = UTXOs (Integer)
    - **Digitale Signatur**: wird aus den Transaktionsdaten und dem privaten Schlüssel generiert, Teil des Unlocking-Mechanismus (base58 codierter String)
    - **Öffentlicher Schlüssel**: „public key" zum Validieren der Signatur, wird aus dem privaten Schlüssel („private key") über ECC (ellyptic curve cryptography, Modul ecdsa) erzeugt, Teil des Unlocking-Mechanismus (base58 codierter String)
5. **Anzahl der Outputs:** Gibt an, wie viele Outputs die Transaktion besitzt (Integer)
6. **Liste an Outputs mit:**
    - **Output-Index:** Identifiziert den Output einer Transaktion eindeutig, beginnt bei 0
    - **Adresse:** an die der Output transferiert werden soll, aus dem öffentlichen Schlüssel des Empfängers generiert (base58 codierter String)
    - **Transaktionsvolumen:** in MilaCoin (MiC) mit maximal 3 Nachkommastellen (float)
    - **Hash des öffentlichen Schlüssels:** „public key hash", Teil des Locking-Mechanismus, wird aus dem öffentlichen Schlüssel über SHA256 und SHA1 generiert

Zusätzlich zum Speichern der Transaktionsdaten im Mempool oder in Block-Dateien werden diese zu Laufzeit des Programms auch als Transaktions-Objekte im Speicher gehalten. Entweder als Liste bei Mempool-Transaktionen, oder als Teil eines Block-Objekts bei Blöcken. Transaktionen werden über die Klasse „Transaction" verwaltet (Modul trm.py). Nachfolgend einige wichtige Methoden der Klasse mit deren Aufgaben:

- **get_tx_id():** Methode berechnet den doppelten SHA256 Hashwert einer Transaktion basierend auf einem Transaktionsobjekt.
- **get_tx_string():** Methode erzeugt aus einem Transaktions-Objekt einen String, der in eine Datei (Mempool oder Block) geschrieben werden kann.
- **get_coinbase_data():** Methode generiert die notwendigen Transaktionsdaten für eine Coinbase-Transaktion. Diese wird automatisch vom System erzeugt.
- **tx_to_mempool():** Methode nimmt einen Transaktionsstring entgegen und schreibt ihn in den Mempool
- **load_tx_from_mempool():** Methode lädt eine Transaktion aus dem Mempool in ein Transaktionsobjekt. Dabei muss durch Angabe eines Index spezifiziert werden, welche der Transaktionen geladen werden soll.
- **load_tx_from_block():** Methode lädt eine Transaktion aus einer Block-Datei in ein Transaktionsobjekt. Dabei muss durch Angabe der Blocknummer und eines Index spezifiziert werden, welche Transaktion des Blocks geladen werden soll.
- **load_tx():** Gemeinsame Endstrecke der Methoden load_tx_from_mempool() und load_tx_from_block(). Hier werden die Daten einer Transaktion auf ein korrektes Format überprüft (aber NICHT die Transaktion validiert!).

2.5.2 Ablauf und Validierung von Transaktionen



(Bild nach: https://www.youtube.com/watch?v=ir4dDCJhdB4&list=PL6TbWIxWsLY0VPlese2_z5xDZZ33ZuvV6&index=13)

Die obenstehende Abbildung zeigt eine einfache **Transaktionskette**, wie sie für Bitcoin üblich ist. Diese Prinzipien wurden für das Projekt MilaCoin übernommen, allerdings zum Teil stark vereinfacht. Gerade die Validierung von Transaktionen wird nicht, wie bei Bitcoin, über eine eigens dafür geschaffene Skriptsprache ausgeführt, sondern über eine Funktion bzw. Methode in der Software.

Die Minerin Alice bekommt als Belohnung für das „finden“ eines Blocks 1 MiC. Diesen Betrag überweist sie dann an ihren Freud Bob. Bei diesem Vorgang finden also zwei Transaktionen (Tx1 und Tx2) statt. Die erste Transaktion in dieser Kette (Tx1) wird vom System automatisch generiert und wird **Coinbase** genannt. Es ist immer die erste Transaktion in einem Block ist dadurch gekennzeichnet, dass sie keine Inputs aufweist. Coinbase-Transaktionen sind immer der Beginn von Transaktionsketten, da hier neue Coins generiert werden. Innerhalb der gesamten Blockchain kann damit jede Transaktion lückenlos auf eine oder mehrere Coinbase-Transaktionen zurückgeführt werden.

Transaktionen besitzt eine eindeutige ID. Dabei handelt es sich um einen Hashwert (2xSHA256), der aus den (meisten) Transaktionsdaten (Tx data) generiert wird (s.u.). Ferner besitzt (bis auf Coinbase-Transaktionen) jede Transaktion einen bis mehrere **Inputs** und einen bis mehrere **Outputs**. Diese sind durch Indizes, beginnend bei 0, innerhalb einer Transaktion eindeutig gekennzeichnet. Wie in einer Kette sind die Outputs von Transaktionen die Inputs für nächste Transaktionen. Als Input für eine Transaktion dienen die ID der vorhergehenden Output-Transaktion (Tx1 ID) und der Index des entsprechenden Outputs (Output Index). Der Wert an MilaCoins (**Volume**) aller Inputs muss immer größer oder gleich dem Wert aller Outputs sein, ansonsten stehen dem Sender nicht genügend Coins für die Transaktion zur Verfügung. Inputs können für Transaktion nicht geteilt werden und werden immer komplett „verbraucht“. Ist die Summe aller Input-Coins also größer als die der Output-Coins, wird **„Wechselgeld“** an den Sender zurück transferiert.

Die Coinbase-Transaktion an Alice besitzt keine Inputs und nur einen Output (Index: 0). Bis Alice sich dazu entschließt, den erworbenen MilaCoin auszugeben, ist der Output 0 von Tx1 ein sogenannter **„Unspent Transaction Output“**, kurz **UTXO**. Alice will ihren guten Freund Bob in die Welt der Kryptowährung einführen und will ihm dafür 1 MiC überweisen. Der UTXO von Alice beinhaltet genau 1 MiC, weswegen dieser Coin ohne Wechselgeld an Bob überwiesen werden kann (Transaktionsgebühren nicht berücksichtigt, s.u.). Die Transaktion Tx2, die Alice für Bob erstellt, beinhaltet demnach einen Input über 1 MiC und einen Output über 1 MiC. Würde Wechselgeld entstehen, würde die entsprechende Menge an Coins in Form eines weiteren Outputs zurück an Alice überwiesen werden. Dies ist aber in dem betrachteten Beispiel der Einfachheit halber nicht zu berücksichtigen, da Input- und Output-Coins gleich sind. Wird der UTXO von Alice bei dieser Transaktion „verbraucht“, d.h. ist er der Input in einer weiteren Transaktion, dann wird er zu einem **„Spent Transaction Output“ (STXO)** und kann damit nicht ein weiteres Mal ausgegeben werden (Lösung des **„Double-Spending-Problems“** bei digitalen Währungen). Der Wert aller UTXOs von Alice beschreibt ihren **Kontostand** und der Wert aller existierenden UTXOs (**„UTXO Set“**) die Menge aller im Umlauf befindlichen MilaCoins.

Damit Alice die 1 MiC an Bob überweisen kann, benötigt dieser aber zunächst eine **Wallet-Software**, in der die Coins, vergleichbar mit einer Briefbörse, aufbewahrt werden. Das ist aber nur eine Analogie, denn es existieren keine MilaCoins, die z.B. in Form einer Datei auf die Wallet-Software geladen werden. Die Funktionsweise einer Wallet Software ist viel komplexer und soll im Folgenden erläutert werden. Damit Bob die Transaktion empfangen kann, braucht er eine **Adresse**, vergleichbar mit einer IBAN, an die Alice die MiC überweisen kann. Solche Adressen werden von der Wallet-Software generiert und gespeichert. Zunächst wird der **private Schlüssel** (Private key) zufällig erstellt. Aus dem privaten Schlüssel wird dann über ECC der **öffentliche Schlüssel** (Public key) generiert. Durch zweimaliges Hashen des öffentlichen Schlüssels (SHA256 und SHA1 bzw. RIPEMD160 bei Bitcoin) wird der **„Public key hash“** (Pub key hash) erzeugt, der wiederum base58 codiert die Adresse liefert. Da alle diese Daten aus dem privaten Schlüssel generiert werden, muss prinzipiell nur dieser in der Wallet-Software gespeichert werden.

Die Adresse, die Bobs Wallet generiert hat, muss er nun an Alice weitergeben. Dann kann Alice ihre Transaktion (Tx2) ausführen. Die Transaktion Tx2 enthält nur einen **Output** mit dem Index 0. Als Empfänger-Adresse gibt sie Bobs Adresse an (Address). Zudem erzeugt sie durch base58-Decodierung den „Public key hash“ (Pub key hash), der Teil des Locking-Mechanismus (s.u.) ist und ebenfalls in den Output geschrieben wird. Als letztes muss sie die Menge an MiC spezifizieren, die an Bob transferiert werden soll (Volume). Der **Input** (Index 0) der Transaktion wiederum referenziert auf die Coinbase-Transaktion (Tx1), bei der Alice 1 MilaCoin als Mining-Belohnung bekommen hat. Ferner wird der ehemalige Output dieser Transaktion spezifiziert (Index 0), was Alices UTXO über 1 MiC entspricht. Bei diesem Vorgang wird also Alices UTXO verbraucht und ein neuer UTXO auf Bobs Adresse generiert.

In den Outputs einer Transaktion wird also über die Adresse festgelegt, welche Person Zugriff auf die MilaCoins dieses UTXO hat. Das bedeutet, dass nur diejenige Person die Coins ausgeben bzw. als Input für eine neue Transaktion verwenden darf, die den **privaten Schlüssel zur angegebenen Adresse** (bzw. zum öffentlichen Schlüssel) im Output besitzt. Um dies sicherzustellen, sind bei Bitcoin ein **Locking- und ein Unlocking-Skript** in den Inputs bzw. in den Outputs implementiert. Über eine sehr rudimentäre und eigens dafür entwickelte Stak-basierte Skriptsprache wird überprüft, ob eine Person die Rechte besitzt, die Bitcoins einer UTXO auszugeben. So eine Skriptsprache schafft eine zusätzliche Sicherheitsbarriere, wurde aber für dieses Projekt nicht umgesetzt. Hier erfolgt die Validierung einer Transaktion über eine einfache Funktion, die die Funktionsweise des Bitcoin-Skripts nachahmt.

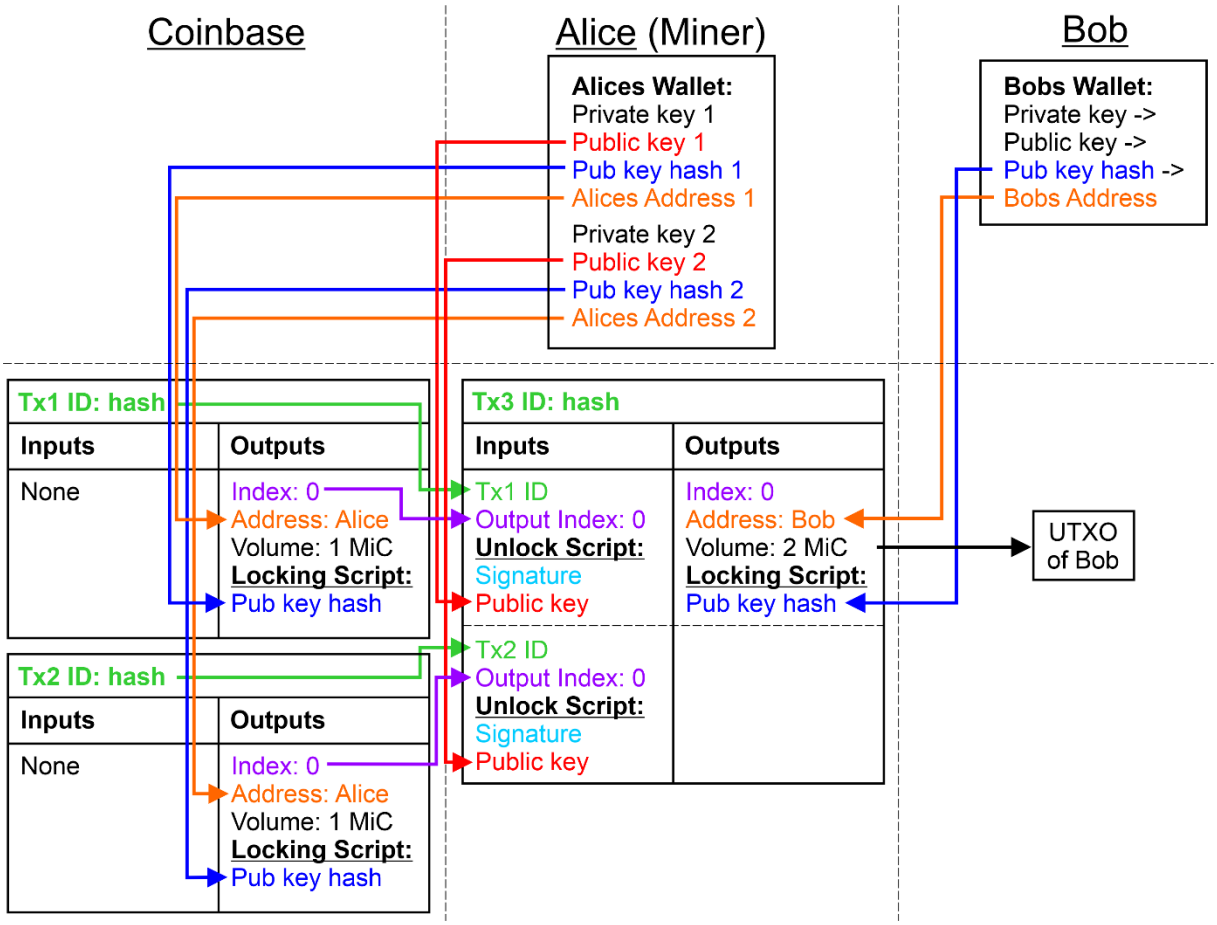
Betrachtet man **Outputs von Transaktionen**, dann wird hier neben der Empfänger-Adresse auch der „Public key hash“ aufgeführt. Das ist Teil des Locking-Skripts, denn damit wird der UTXO an eine Adresse bzw. einen öffentlichen Schlüssel gebunden. **Nur wer den zugehörigen privaten Schlüssel besitzt, kann die MiC dieses Outputs wieder ausgeben!** Dafür gibt es in jedem Input einer Transaktion ein Unlocking-Skript, das aus dem vollen **öffentlichen Schlüssel** (Public key) und einer **Signatur** (Signature) der Transaktion besteht. Um die Signatur einer Transaktion zu erhalten, wird der **private**

Schlüssel benötigt, den in diesem Falle nur Alice kennt. Der private Schlüssel verbleibt immer in Alices Wallet und wird niemals preisgegeben bzw. in die Daten einer Transaktion aufgenommen. Um trotzdem zu beweisen, dass Alice die MiC dieses UTXOs ausgeben kann, generiert sie aus dem privaten Schlüssel und aus den Daten der Transaktion mit Hilfe von ECC eine **digitale Signatur**. Die Daten einer Transaktion (exklusive der Daten des Unlocking-Skripts, also öffentlicher Schlüssel und Signatur) werden dafür 2x mit SHA256 gehashed (Tx2 data hash) und dann zusammen mit dem privaten Schlüssel zur Signatur verrechnet. Der „Tx2 data hash“ wird übrigens auch als ID der Transaktion verwendet.

Für die **Validierung einer Transaktion** werden der Output der vorangegangenen Transaktion und der zugehörige Input der aktuellen Transaktion benötigt. Zunächst wird der öffentliche Schlüssel im Unlocking-Skript 2x gehashed (SHA256 und SHA1 bzw. RIPEMD160 bei Bitcoin) und mit dem „Public key hash“ im Locking-Skript des Outputs verglichen. **Sind diese gleich, dann heißt das, dass die Person, die diese MiC empfangen hat auch versucht, diese auszugeben.** Ist dieser Vergleich erfolgreich, dann wird in einem nächsten Schritt die Signatur überprüft, wofür die Daten der Transaktion, die Signatur und der öffentliche Schlüssel benötigt werden. Mit Hilfe des ecdsa-Moduls kann dann verifiziert werden, dass die Person, welche die digitale Signatur erstellt hat, auch tatsächlich **im Besitz des privaten Schlüssels ist**, ohne den Schlüssel selbst preisgeben zu müssen. Ist auch dieser Test erfolgreich, gilt die Transaktion als valide. Wird bei Bitcoin eine Transaktion erstellt, wird diese zunächst an alle Nodes im Netzwerk geschickt. Erst wenn alle Nodes die Transaktion validiert haben, wird diese in den Mempool geschrieben und kann dann in einen Block aufgenommen werden.

Nach erfolgreicher Transaktion ist aus dem ehemaligen **UTXO** von Alice eine **STXO** geworden. Dafür ist Output 0 in Tx2 ein neuer UTXO, welcher an Bobs Adresse gebunden ist. Die Rechte an einem MilaCoin sind also von Alice auf Bob übertragen worden. Bei Bitcoin werden alle UTXOs im Speicher gehalten, was momentan etwa einer Zahl von 84,475 Millionen entspricht. Für das Projekt sollen alle UTXOs entweder in einer Textdatei gespeichert werden oder alternativ können auch die privaten Schlüssel aus den Wallets der Benutzer gelöscht werden, wenn aus einem UTXO ein STXO wird.

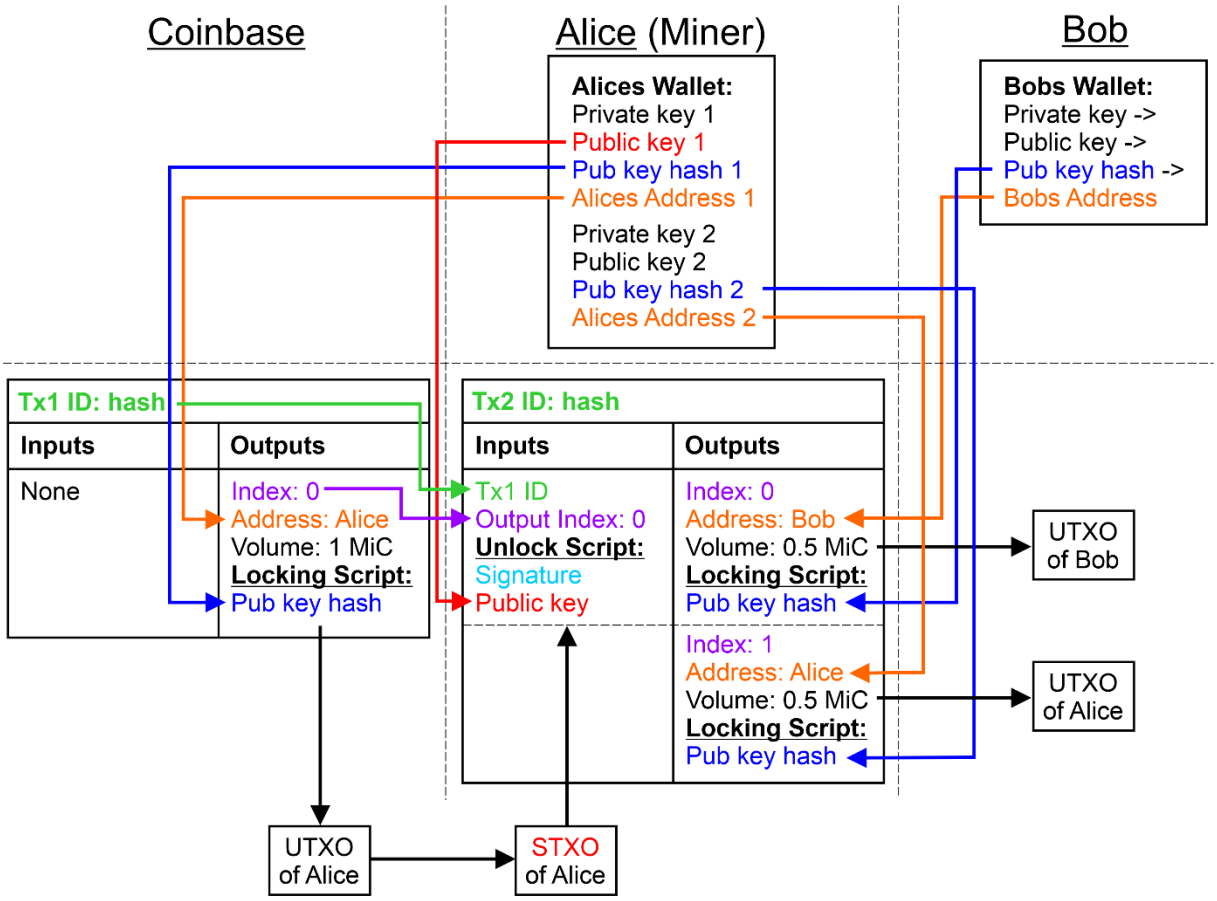
2.5.3 Transaktionen mit mehreren Inputs



Im oben abgebildeten Beispiel überweist Alice 2 MiC an Bob. Da die 1 MiC aus Tx1 nicht ausreichen, muss Alice einen weiteren Input über 1 MiC in ihre Transaktion einfügen. Den entsprechenden **UTXO** aus Tx2 hat Alice ebenfalls beim Mining erworben. Die Transaktion von Alice beinhaltet somit **zwei Inputs (2 x 1 MiC) und einen Output (2 MiC)**. Wie bereits erwähnt, muss der Wert aller Inputs größer oder gleich dem Wert aller Outputs sein, damit eine Transaktion durchgeführt werden darf. Die Wallet-Software scannt dabei alle UTXOs einer Person und wählt diejenigen aus, deren Summe größer oder gleich dem Output Wert ist. Kann die Summe nicht erreicht werden, stehen dem Sender nicht genügend MiC zur Verfügung und die Transaktion kann nicht durchgeführt werden.

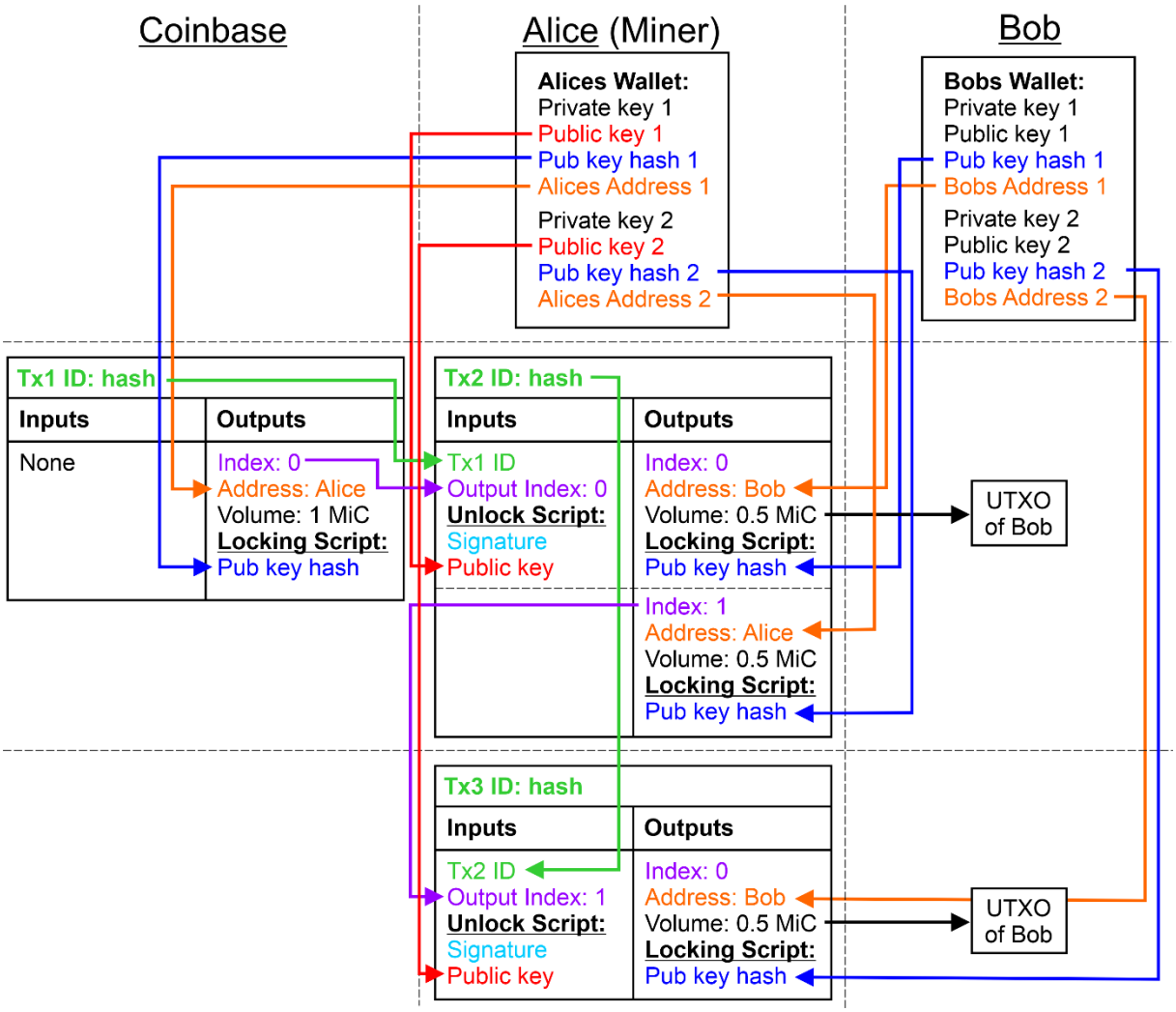
Bei dieser Transaktion Tx3 werden zwei UTXOs über jeweils 1 MiC von Alice „verbraucht“ und die Rechte an diesen Coins in Form eines UTXOs über 2 MiC an Bob übertragen. **Ein UTXO gilt dann als „verbraucht“, wenn er als Input in irgendeiner Transaktion verwendet wurde.** Dann wird er als **STXO** bezeichnet und kann nicht mehr als Input für weitere Transaktionen dienen, die Coins sind sozusagen ausgegeben. Um die UTXOs und STXOs einer Person herauszufinden, muss die gesamte Blockchain gescannt werden. Diese Aufgabe wird von der Wallet-Software durchgeführt. Zunächst müssen alle Outputs aller Transaktionen aller Blöcke gescannt werden, welche an eine Adresse der Person gesendet wurden. Danach müssen alle Inputs gescannt werden. Dort wird überprüft, ob auf die Outputs bereits referenziert wurde (Tx ID und Output Index), dann sind sie STXOs, oder nicht, dann sind es UTXOs.

2.5.4 Transaktionen mit mehreren Outputs (Wechselgeld)



Im nächsten Beispiel überweist Alice nur 0,5 MiC an Bob. Die Wallet-Software findet keine passenden UTXOs, weswegen ein UTXO über 1 MiC verwendet werden muss. Da UTXOs bei Transaktionen immer vollständig "verbraucht" werden, beinhaltet Alices Transaktion neben dem Output 0 über 0,5 Mic an Bob einen weiteren Output 1, der an sie selbst gerichtet ist (**Wechselgeld**, „Change"). Damit entstehen aus dem ursprünglichen UTXO über 1 MiC, das Alice beim Minen gewonnen hat, zwei neue UTXOs, einer an Bob und einer an Alice.

Als **Analogie** kann man sich vorstellen, dass UTXOs Goldmünzen verschiedener Größe und damit Wert darstellen. Bei Transaktionen werden Münzen eingeschmolzen und entweder zu einer neuen Münze zusammengeführt, oder in mehrere kleine Münzen aufgeteilt. Münzen müssen dabei immer komplett eingeschmolzen werden und können nicht geteilt werden.



Alice mag Bob <3. Sie sieht, dass Bob sich gut in die Krypto-Welt eingearbeitet hat und entschließt sich ein paar Tage später, weitere 0,5 MiC an ihn zu überweisen (Tx3). Die Abbildung oben beschreibt den Vorgang und ist einer Erweiterung der vorhergehenden Abbildung. Wichtig ist, dass die Outputs von Transaktionen an bestimmte Adressen gebunden sind („**Locking Script**“). Nur wer über die privaten Schlüssel verfügt, aus dem die entsprechenden öffentlichen Schlüssel und damit auch die Adressen generiert wurden, kann diese Outputs auch ausgeben („**Unlocking Script**“). Betrachtet man die Transaktionskette, dann findet die Validierung immer zwischen den Outputs der vorhergehenden mit den Inputs der nachfolgenden Transaktion statt. Jeder kann Coins an beliebige Adressen versenden. Dieses System stellt primär sicher, dass nur diejenige Person Coins ausgeben kann, die auch die privaten Schlüssel dafür besitzt.

2.5.5 Transaktionen mit mehreren Inputs und Outputs

Oft besteht eine Transaktion aus einer Kombination der oben abgebildeten Varianten. Ein UTXO reicht nicht aus, um die zu überweisende Summe zu decken, weswegen die Transaktion mehrere Inputs besitzt. Darüber hinaus ist die Summe der Coins aller verwendeten UTXOs nicht exakt passend, weswegen Coins in Form von Wechselgeld an den Sender zurück überwiesen werden müssen. Damit besitzt die Transaktion auch mehrere Outputs.

2.5.6 Transaktionen in der Textdatei

Einzelne Transaktionen:

Delimiter: {tx:}

Transaktions-Header: Delimiter: [x]

Einzeilig, Abschluss: \n
Header Elemente: Delimiter |

- Tx ID
- Timestamp
- Anzahl Inputs
- Anzahl Outputs

Inputs: Delimiter: [x]

Einzelne Inputs: Delimiter \n
Input Elemente: Delimiter |

- Tx ID Output
- Index Output (UTXO)
- Öffentlicher Schlüssel
- Signatur

Outputs: Delimiter: [x]

Einzelne Outputs: Delimiter \n
Output Elemente: Delimiter |

- Index Output
- Adresse
- Volumen
- [Hash öffentlicher Schlüssel]

Da hier nicht, wie bei Bitcoin, ein Skript für die Validierung von Transaktionen verwendet wird, kann auf die Angabe des Hashwertes des öffentlichen Schlüssels verzichtet werden. Dieser kann auch direkt aus der Adresse über base58 Decodierung generiert werden, welche sowieso im Output enthalten ist.

Beispiel:

{tx:}
5bd12b1005ee893aecb8047a3efc06a487df21058b4dcae658c4c6d91f532f33|1661099078.101517|0
|1
[x]
[x]
0|4CCDRPtRzknVprT2KBmqs7xTBgz2|10.000\n
{tx:}
d42414c6be60b98f0514993f80dc271894797d38d463c396cf7acde6f6af6dde|1661097175.406116|3
|2
[x]
ce7f2d2751f0f4b851952b3eaf096f22c1ead6e97f80e33a5833edd67da8748f|0|4vExyFMRQ5PxuP6h1
qMbKv8TdwKDgMMk1W545A1HtAi1Nik5mqvjnf7z12usqnVnzrCU9ppZjMaf6z1pqc8vds9P|2dzsbohT
V2zmShpf4hyFfwnNbm1HnjNCei6giXnWe4FuqaXHFwUaXi3wC8HmWgYQeqABYmEpmqgTeV7ATQBG
xGsi\n
b383bae8fe273f764fc6c91d7fa36c8645368789fc6427cabd26eefda515764d|1|7gkm3iYaZzxNzRHcVK
qiD8BVxxKmKywBdbon6nCN6wn2VW4Xwyq1MtkAdZxkrkLgzmipBMHgTCmDTykqwf4FNHZ|67mLK6

GHitA67tYYpdkRczMQmnSuPxKiG8MeUNDGvk22dan3fHkJLNQ68q4HQdNSi477rV3oioz9mLKtGV1q2cQ7\n
da9665703c547539f585e31c1c9f89fbfce0ac7ca2d09ade6cd904da1b04a2f0|0|L6vsa7Yf7xPDTc1RXov1DmntFsSpTZEHJ6PLPxkwJtDqUmHiBSAxXn6ttmgG4Tu1Vyhh5MrbkKfcNzK55GkoD5r|2iVbemJvf4jWQyabWzKfMPA9A9fnXwX63X3Lm4a8dV8bmVc5NUXbm4EA4DjQpq7ncvcT4BFeF1kB5MKMtEQuC9oG\n
[x]
0|3a5FBwxhJUvzCSLcCw5bfu3GMfkW|5.555\n
1|4NgGiW2d7pbGzch5DfLCiBPVMdWT|5.945\n

Transaktion Delimiter: {tx:}
Transaktions-Elemente Delimiter: [x]
Delimiter zum Trennen von Inputs/Outputs: \n
Element Delimiter: |

2.6 Das Minen von Blöcken

In dieser Software kann unter „Settings“ eingestellt werden, wie viele Transaktionen im **Mempool** sein müssen, bevor ein Block „gemined“ werden kann. Da zu Beginn einer Blockchain keine Coins vorhanden sind, können folglich auch keine Transaktionen durchgeführt werden. Deswegen muss ein Block zunächst direkt gemined werden können, um überhaupt Geld zu erschaffen. Für spätere Versionen der Software kann es so gehandhabt werden, dass beispielsweise alle 10 Blöcke eine weitere Transaktion in Mempool hinzukommen muss, bevor der Block gemined werden kann. Das wurde allerdings noch nicht umgesetzt. Bei Bitcoin muss die Menge an Transaktionsdaten größer als 1 Mb sein, damit ein neuer Block generiert werden kann, die Blockgröße ist dort also begrenzt.

Beim minen werden folgende Prozeduren in der angegebenen Reihenfolge ausgeführt:

- Alle Transaktionen aus dem Mempool werden in Transaktions-Objekte geladen
- Die Signatur der Transaktionen wird auf Gültigkeit überprüft
- Der Mempool wird geleert, damit neue Transaktionen aufgenommen werden können, solange der Mining-Prozess andauert
- Eine neues Block-Objekt wird erstellt
- Der Header des Blocks wird geschrieben
- Die erste Transaktion eines Blocks, die Coinbase Transaktion, wird in den Block geschrieben
- Alle gültigen(!) Transaktionen aus dem Mempool werden in den Block geschrieben
- Der Nonce wird vom Miner gesucht („proof of work“, siehe unten)
- Eine neue Block-Datei wird erstellt (Dateiname = Nummer des Blocks mit Endung .bl) und die Daten des Block-Objekts als String transferiert
- Das neue Block-Objekt wird dem Blockchain-Objekt angefügt

Der **Nonce** ist eine Integer-Zahl, die an die letzte Stelle des Block-Headers geschrieben wird. Sie sorgt dafür, dass der Hash des Block-Headers mit einer definierten Anzahl an Nullen beginnt. Beim Minen des Blocks werden von 0 beginnend aufsteigend Zahlen ausprobiert, bis die richtige gefunden wurde. Bei jeder neuen Zahl wird mit den weiteren Daten des Block-Headers der Hashwert gebildet und dann überprüft, ob der Hash mit einer vorgegebenen Anzahl an Nullen beginnt. Die Anzahl an Nullen wird als Mining-Schwierigkeit bezeichnet und wird momentan als globale Konstante definiert. Eine Mining-Schwierigkeit von 5 geht relativ schnell, 6 ist auch noch zumutbar, 7 dauert bereits mehrere Minuten, je nach Computer.

Ein Beispiel: Wird die Mining-Schwierigkeit 5 gewählt, so muss der Block-Header Hashwert mit 5 Nullen beginnen (z.B.: **00000**98ddc85310dfbb91cb695593c6471e98d8cd7ad70b03fb1be1ae26a9cc3). Da es unmöglich ist, einen solchen Hashwert zu berechnen, müssen zufällig Zahlen als Nonce ausprobiert werden. Je nach eingestellter Mining-Schwierigkeit kann das unter Umständen mehrere Stunden bis Tage dauern. Limitierend ist bei diesen Berechnungen die Generierung des Hashwertes. Diese Hashwerte können von Grafikkarten viel schneller berechnet werden, als von CPUs. Die Summe der Hashwerte aller Miner im Bitcoin Netzwerk wird als Hashrate (hash rate) bezeichnet. Diese liegt für Bitcoin zum Zeitpunkt des Schreibens dieser Arbeit bei 184.456 Ehash/s (E = Exa = 10^{18}). Als Vergleich: Das Minen mit einer aktuellen Grafikkarte (z.B. NVIDIA rtx 3060 LHR) bringt etwa 35 Mhash/s. Die Hashrate ist bei Bitcoin inzwischen derart groß, dass sich das Minen mit Grafikkarten nicht mehr lohnt. In aller Regel werden eigens dafür gebaute Geräte verwendet (ASIC Miner). Während die Mining-Schwierigkeit in diesem Programm händisch vorgegeben werden muss, wird diese bei Bitcoin für jeden Block neu festgelegt. Die Schwierigkeit wird derart gewählt, dass durchschnittlich alle 10 min ein neuer Block gefunden wird. Je mehr Miner also im Netzwerk nach Bitcoin schürfen, desto größer wird die Schwierigkeit.

Warum das ganze Mining? Das Mining sichert das Netzwerk! Wird eine Manipulation in den Daten der Blockchain vorgenommen, wird sich zwangsläufig der Header(-Hash) im betroffenen Block ändern. Damit passt dieser Block nicht mehr zum darauffolgenden Block, weil diese ja über den Header-Hash verbunden sind. Um also eine solche Manipulation durchführen zu können, muss der Hash ALLER nachfolgenden Blöcke neu berechnet werden. Dafür bräuchte man allerdings mehr Hashrate, als das gesamte restliche Bitcoin Netzwerk aktuell hat. Jede „Full-Node“ im Bitcoin Netzwerk speichert die gesamte Blockchain. Diese ist damit weltweit in Form vieler Kopien vorhanden. Will man die Blockchain manipulieren, dann müssten die nachfolgenden Blöcke sehr schnell berechnet werden. Ist man zu langsam, wird dieser „Seitenast“ der Blockchain einfach ignoriert, weil er kürzer ist als die restlichen Kopien im Netzwerk. Kommt es zum Auftreten von Seitenketten in der Bitcoin-Blockchain, dann wird immer derjenige Ast als der Richtige definiert, der am längsten ist (oder genauer diejenige, in dem die meiste Arbeit geflossen ist). Vor diesem Hintergrund lässt sich auch leicht verstehen, warum Transaktionen als umso sicherer gelten, je mehr Blöcke darüber liegen. Man sagt bei Bitcoin, dass eine Transaktion „in Stein gemeißelt“ ist, wenn mindesten sechs Blöcke über dem Block mit der Transaktion liegen.

Das ober beschriebene Prinzip wird **„proof-of-work“** genannt und wurde nicht für Bitcoin erfunden, sondern schon viel früher diskutiert. Zu Beginn des Internets gab es Vorschläge, Emails nach dem PoW-Prinzip zu verschicken, um Spam Mails zu unterbinden. Die Idee war dabei, dass der Computer beim Versenden einer Email Arbeit leisten muss. Wird nur eine einzelne Mail versandt, dann würde das nicht groß auffallen. Würden jedoch tausende Spam-Emails versandt, dann würde das enorme Energiekosten verursachen. Leider ist das nie umgesetzt worden.

3. Limitierungen der Umsetzung und Fazit

Diese Projektarbeit erhebt keinen Anspruch auf Vollständigkeit, d.h. es handelt sich nicht um eine lauffähige Kryptowährung. Vielmehr sollen die Grundzüge der Funktionsweise von Bitcoin und damit auch vielen anderen Kryptowährungen demonstriert werden. Es gibt mehrere Gründe, warum die Software in dieser Form nicht einsatzbereit ist:

1. Ist die Software nicht als **Peer-to-peer (P2P) Netzwerk** umgesetzt, was wahrscheinlich die wichtigste Limitierung darstellt. Momentan kann das Programm nur auf einem Rechner laufen, was überhaupt keinen Sinn macht. Benutzer, die eine Transaktion machen wollen, müssten sich um diesen einen Rechner streiten. Da es jedoch Peer-to-peer Module für Python gibt (z.B. PyP2P), sollte die Implementierung von P2P auch ohne weitreichende Netzwerkkenntnisse möglich sein.
2. Sind alle Textdateien, in denen die Daten dauerhaft gespeichert sind, UTF-8 codiert und damit lesbar. Das wurde aus didaktischen Gründen gemacht, damit nachvollzogen werden kann, wo Daten in welcher Form gespeichert werden. Standard wäre es, die **Daten binär zu speichern**, was einen weiteren (wenn auch geringen) Manipulationsschutz bietet.
3. Sind die **Algorithmen nicht auf Geschwindigkeit optimiert**. Bei einer Blockchain mit wenig Blöcken fällt das nicht auf. Würde die Blockchain allerdings wie bei Bitcoin 420 BG groß sein, wären die momentanen Algorithmen viel zu langsam und eine sinnvolle Bedienung nicht möglich. Dies betrifft vor allem Methoden, die Daten in der Blockchain auslesen, allen voran die Methode „load_user_utxos()“ der Klasse Blockchain. Um das Problem partiell zu umgehen, werden solche Daten, wenn möglich, im Blockchain-Objekt (und damit im Speicher) gesucht und nicht in den Block-Dateien, was ein permanentes Öffnen und Schließen von Textdateien erfordern würde.
4. Sollte die **Mining-Schwierigkeit** nicht per Hand vorgegeben werden, sondern über Algorithmen. Sinnvoll wäre es, sie wie bei Bitcoin von der aktuellen Hashrate abhängig zu machen. Das erfordert allerdings wiederum ein P2P-Netzwerk.
5. Sollte die **minimale Anzahl an Transaktionen im Mempool** zum minen eines neuen Blocks nicht per Hand vorgegeben werden. Bei Bitcoin würde das der Limitierung der Blockgröße auf 1 MB entsprechen. Dieser Parameter hat starken Einfluss darauf, wie schnell die Geldmenge im System wächst. Wächst sie zu schnell an, kommt es zu einer Inflation. Vorstellbar wäre, die minimale Anzahl an Transaktionen alle 10 Blöcke um eins zu erhöhen. Der Maximale Wert sollte aber gedeckelt sein.
6. Sind keine **Transaktionsgebühren** umgesetzt worden. Begrenzt man wie bei Bitcoin die Anzahl an Coins, die jemals generiert werden können, dann gäbe es für die Miner nach dem Schürfen der letzten Coins keinen Grund mehr, neue Blöcke zu validieren. Keine Miner, kein Bitcoin!

Als **Fazit** muss ich sagen, dass die Umsetzung dieses Projekts sehr viel Spaß gemacht hat. Für mich war es eine große Herausforderung, die Prozesse, welche beim Bezahlen mit Kryptowährung ablaufen, zu verstehen. Darüber hinaus musste ich mich in kryptographische Prinzipien und Algorithmen einarbeiten, mit denen ich bisher praktisch keine Berührung hatte. Dieses Projekt hat meinen Horizont stark erweitert und mit gezeigt, dass Programmieren mehr sein kann als Sensordaten zu prüfen oder Datenkolonnen auszuwerten. Mit Python kann ein solches Projekt schnell und unkompliziert umgesetzt werden. Aus Zeitgründen wurde aber vermutlich ein Teil des Quelltextes nicht „pythonisch“ genug programmiert, sondern stattdessen eher an das mir vertraute C angelehnt. Hier bleibt noch viel Raum für Verbesserungen und Optimierungen.

4. Quellenangaben

Bitcoin- und Blockchain-Tutorials:

- https://www.youtube.com/watch?v=ir4dDCJhdB4&list=PL6TbWIxWsLY0VPlese2_z5xDZZ33ZuvV6&index=13
- https://www.youtube.com/playlist?list=PL6TbWIxWsLY0VPlese2_z5xDZZ33ZuvV6
- https://www.oreilly.com/library/view/mastering-bitcoin/9781491902639/
- https://developer.bitcoin.org/devguide/transactions.html
- https://medium.com/@blairlmarshall/how-does-a-bitcoin-transaction-actually-work-1c44818c3996
- https://en.bitcoin.it/wiki/Transaction
- https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch06.asciidoc
- https://learnmeabitcoin.com/beginners/
- https://www.deltecbank.com/2021/10/05/bitcoin-transaction-validation-what-exactly-goes-on-under-the-hood/?locale=en

Kryptographie und ECC:

- https://de.wikipedia.org/wiki/Base58
- https://de.wikipedia.org/wiki/Kryptographische_Hashfunktion
- https://www.computerweekly.com/de/definition/Elliptische-Kurven-Kryptografie-Elliptic-Curve-Cryptography-ECC
- https://en.bitcoin.it/wiki/Secp256k1
- https://de.wikipedia.org/wiki/Asymmetrisches_Kryptosystem
- https://davidederosa.com/basic-blockchain-programming/elliptic-curve-keys/
- https://learnmeabitcoin.com/beginners/digital_signatures
- https://cryptobook.nakov.com/digital-signatures/ecdsa-sign-verify-messages

Python und Programmierung:

- https://stackoverflow.com/questions/59121573/python-login-and-register-system-using-text-files
- https://pypi.org/project/ecdsa/
- https://www.programcreek.com/python/example/94248/ecdsa.ecdsa
- https://www.codingem.com/copy-text-to-clipboard-in-python/
- https://www.devdungeon.com/content/working-binary-data-python
- https://www.adamsmith.haus/python/examples/4038/base64-check-if-a-string-can-be-decoded-from-
- https://stackoverflow.com/questions/58437353/how-can-we-efficiently-check-if-a-string-is-hexadecimal-in-python
- https://www.pythontutorial.net/python-basics/python-check-if-file-exists/
- https://pynative.com/python-timestamp/
- https://datagy.io/python-sha256/
- https://www.quickprogrammingtips.com/python/how-to-calculate-sha256-hash-of-a-file-in-python.html
- https://pynative.com/python-count-number-of-files-in-a-directory/
- https://stackoverflow.com/questions/28749177/how-to-get-number-of-decimal-places
- https://www.delftstack.com/de/howto/python/sort-list-of-lists-in-python/