Dennis | Alkan | Samer

1 Einleitung

Die Erstellung einer sicheren Bitcoin-Wallet erfordert einige wichtige Schritte aus der Perspektive der Informationssicherheit. Dazu gehören die Auswahl der Wallet-Software, die sichere Generierung des privaten Schlüssels, die sichere Aufbewahrung des privaten Schlüssels, die Verbindung mit dem Bitcoin-Netzwerk und die regelmäßige Überprüfung der Wallet-Sicherheit. Indem man diese Schritte befolgt, kann man eine sichere Bitcoin-Wallet erstellen und vor potenziellen Angriffen schützen.

2 Laufzeitumgebung anpassen

Nach dem Starten des Jupyter Notebooks muss der folgende Code immer zu erst ausgeführt werden, um die Laufzeitumgebung auf Java anzupassen.

```
%%bash
#!/usr/bin/env bash

echo "Update environment..."
apt update -q  &> /dev/null

echo "Install Java..."
apt-get install -q openjdk-11-jdk-headless &> /dev/null

echo "Install Jupyter java kernel..."
curl -L https://github.com/SpencerPark/IJava/releases/download/v1.3.0/ijava-1.3.0.zip \
 -o ijava-kernel.zip &> /dev/null

unzip -q ijava-kernel.zip -d ijava-kernel \
 && cd ijava-kernel \
 && python3 install.py --sys-prefix &> /dev/null

echo "Install proxy for the java kernel"
# NOTE: required after changes to Google Colab defaults in Dec. 2022
# See https://stackoverflow.com/questions/74674688/google-colab-notebook-using-ijava-stuck-at-connecting-after-installation-ref/748217

wget -qO- https://gist.github.com/SpencerPark/e2732061ad19c1afa4a33a58cb8f18a9/archive/b6cff2bf09b6832344e576ea1e4731f0fb3df10c.tar.ga
python install_ipc_proxy_kernel.py --kernel=java --implementation=ipc_proxy_kernel.py
```

```
Update environment...
Install Java...
Install Jupyter java kernel...
Install proxy for the java kernel
e2732061ad19c1afa4a33a58cb8f18a9-b6cff2bf09b6832344e576ea1e4731f0fb3df10c/install_ipc_proxy_kernel.py
e2732061ad19c1afa4a33a58cb8f18a9-b6cff2bf09b6832344e576ea1e4731f0fb3df10c/ipc_proxy_kernel.py
Moving java kernel from /usr/share/jupyter/kernels/java...
Wrote modified kernel.json for java_tcp in /usr/share/jupyter/kernels/java_tcp/kernel.json
Installing the proxy kernel in place of java in /usr/share/jupyter/kernels/java
Installed proxy kernelspec: {"argv": ["/usr/bin/python3", "/usr/share/jupyter/kernels/java/ipc_proxy_kernel.py", "{connection_file}
Proxy kernel installed. Go to 'Runtime > Change runtime type' and select 'java'
```

3 Externe Bibliotheken

Es werden die Bibliotheken bitcoinj & bouncycastle benutzt, welche durch folgenden Befehl in die Laufzeitumgebung importiert werden.

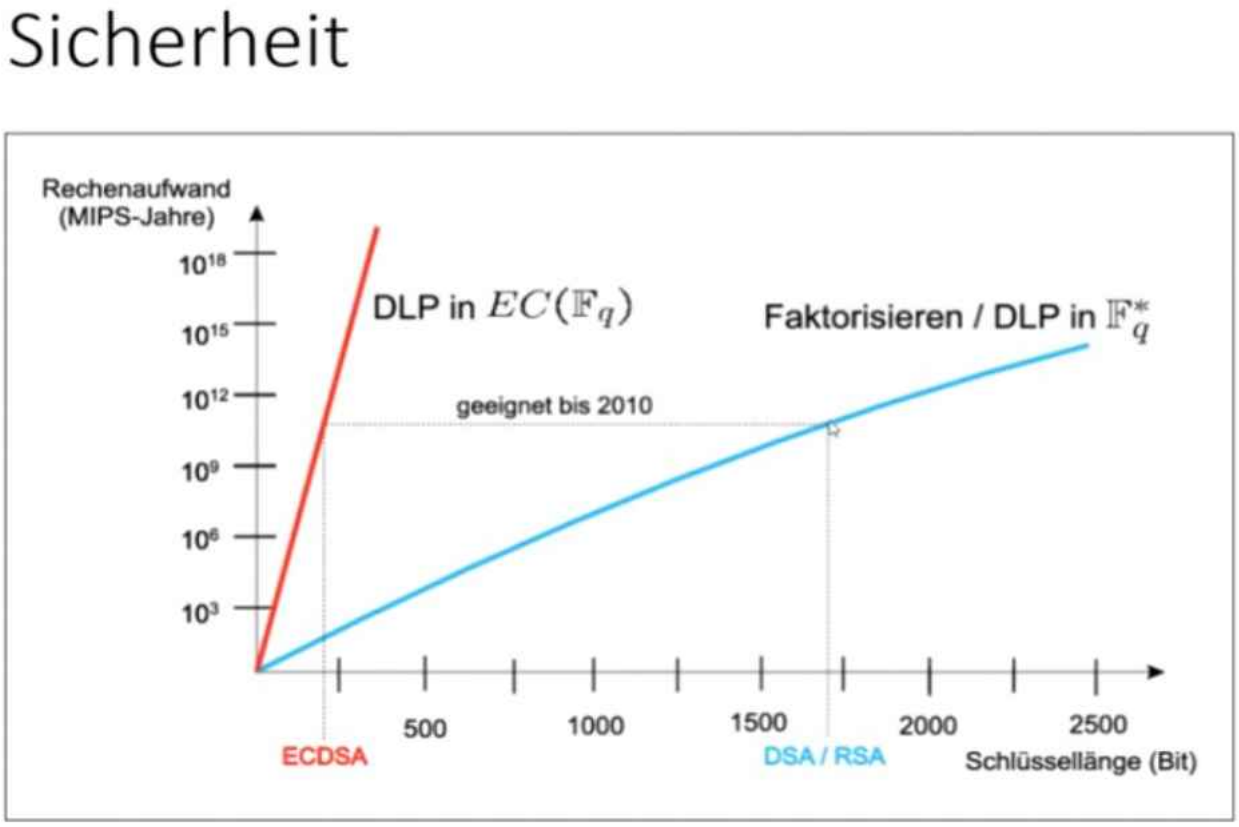
```
%%loadFromPOM
<dependencies>
    <dependency>
      <groupId>org.bitcoinj</groupId>
      <artifactId>bitcoinj-core</artifactId>
      <version>0.16.2</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>org.bouncycastle</groupId>
      <artifactId>bcpkix-jdk15on</artifactId>
      <version>1.62</version>
    </dependency>
</dependencies>
```

4 Erläuterung von Hashfunktionen & Algorithmen

4.1 ECDSA

ECDSA steht für Elliptic Curve Digital Signature Algorithm & ist eine Variante des Digital Signature Algorithm (DSA), erweitert um elliptische Kurvenkryptographie. Der Vorteil der sich durch die Erweiterung ergibt ist, dass bei selber Schlüssellänge ein weitaus höherer Rechenaufwand nötig ist, um den privaten Schlüssel zu finden.

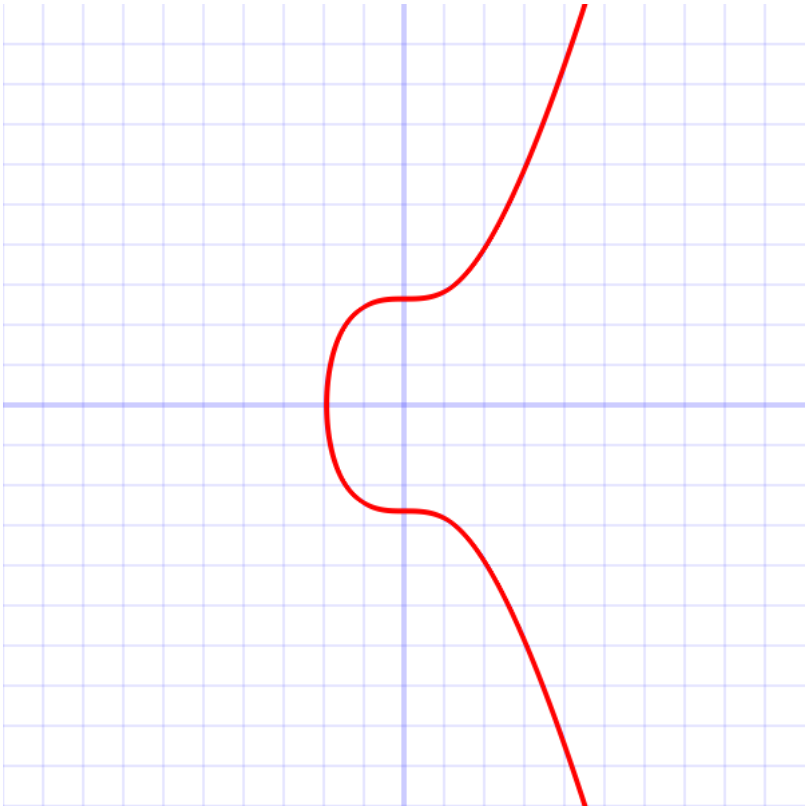
Ein Sicherheitsniveau von t = 80 Bit bedeutet, dass ein Angreifer 2^80 elementare Operationen durchfüren muss, um den privaten Schlüssel zu finden. Ein DSA-Schlüssel hätte bei diesem Niveau eine Länge von etwa 1024 Bit, ein ECDSA-Schlüssel hingegen bloß eine Länge von 160 Bit, also doppelt so groß wie das Sicherheitsniveau.



Wir verwenden ECDSA einerseits zur Schlüsselerzeugung und andererseits zum erzeugen einer Signatur. ECDSA ist abhängig von der verwendeten Kurvenordnung und Hashfunktion. Bei Bitcoin sind dies secp256k1 und SHA256.

4.2 secp256k1

secp256k1 ist ein definierter Standard, der bei Bitcoin beim ECDSA als Kurvenordnung Anwendung findet und bedeutet "Standard for Efficient Cryptography Prime 256-bit Koblitz Curve". "Efficient Cryptography" bedeutet, dass die Kurve für kryptographische Anwendungen wie digitale Signaturen und Schlüsselaustausch optimiert ist. "256-bit" bezieht sich auf die Größe des Schlüsselraums, der in diesem Fall 256 Bits beträgt. "Koblitz curve" bezieht sich auf die spezielle Form der elliptischen Kurve, die in SECP256K1 verwendet wird und von Neal Koblitz im Jahr 1987 eingeführt wurde.



secp256k1 ist wie folgt definiert:

Die Parameter der elliptischen Kurvendomäne über Fp, die mit einer Koblitz-Kurve secp256k1 verbunden sind, werden durch das Sextupel T = (p,a,b,G,n,h) spezifiziert, wobei das endliche Feld Fp definiert ist durch:

> p = 2^256 - 2^32 - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1

oder in Hexadezimaldarstellung:

> p = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F

Die Kurve E: $y^2 = x^3 + ax + b$ über Fp ist definiert durch:

> a = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

> b = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000007

Generatorpunkt G in unkomprimierter Form:

> G = 04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8

Ordnung n von G:

> n = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141

Kofaktor:

> h = 01

4.2 Schlüsselerzeugung

Der private Bitcoinschlüssel ist ein zufällig erzeugter, 256-bit großer Wert. Um einen öffentlichen Bitcoinschlüssel zu erzeugen, muss der private Schlüssel mittels Punkt-Multiplikation mit dem dem Generatorpunkt G der secp256k1-Kurve multipliziert werden. Das Ergebnis ist der EC-Point, der ein Punkt auf der secp256k1-Kurve ist. Der unkomprimierte öffentliche Schlüssel wird zusammengesetzt aus dem Prefix "04", welcher für unkomprimiert steht, sowie der X- und Y-Koordinate des EC-Points.

> EC-Point = privater Schlüssel * G

> Unkomprimierter öffentlicher Bitcoinschlüssel: "04" & X-Achse des EC-Points & Y-Achse des EC-Points

Die Größe des unkomprimierten öffentlichen Schlüssels beträgt 513-bit. Die Koordinaten sind jeweils 256-bit groß, der Prefix 1-bit.

Durch die Eigenschaften der Punt-Multiplikation ist es mathematisch unmöglich, den private Schlüssel aus dem öffentlichen zu berechnen.

4.3 Signatur

Die Signatur findet bei Bitcoin vorallem bei Transaktionen statt und basiert genau wie die Erzeugung des öffentlichen Schlüssels auf ECDSA(secp256k1). Es wird ein Wertepaar r und s aus dem Privaten Schlüssel und dem Transaktionsdaten-Hash erzeugt.

Ablauf im Detail:

> Die Transaktionsdaten werden doppelt mit SHA256 gehasht. --> hash = SHA256(SHA256((Transaktionsdaten))

> Es wird eine zufällige Zahl k im Bereich [1...n-1] erzeugt, wobei n der Ordnung von G der secp256k1-Kurve entspricht.

> Erzeugung des Random-Points R, welches aus der Multiplikation von k mit dem Generatorpunkt G entsteht. --> R = k * G

> Die X-Koordinate des Random-Points modulo n ergibt den r-Wert der Signatur. (Ist r = 0, muss eine neue zufällige Zahl k erzeugt werden.) --> r = x mod n

> Der s-Wert der Signatur wird wie folgt berechnet: s = k^-1 * (hash + r * privater Schlüssel) (mod n) (Ist s = 0, muss eine neue zufällige Zahl k erzeugt werden.)

Die berechnete Signatur {r, s} ist ein Integer-Wertepaar, wobei r und s beide im Bereich [1...n-1] liegen. Der Wert s bestätigt, das der signierende sowohl die Transaktionsdaten kennt wie auch den privaten Schlüssel besitzt. Durch diesen Wert kann mittels des öffentlichen Schlüssels die Signatur verifiziert werden.

Um die Signatur zu verifizieren, passiert folgendes:

> Die Transaktionsdaten werden erneut doppelt mit SHA256 gehasht.

> Es wird das modulare Inverse des Wertes s berechnet: s1 = s^-1 (mod n)

> Wiederherstellung des Randompoints durch die Funktion: R' = (h * s1) * G + (r * s1) * öffentlicher Schlüssel

> Nun berechnen wir r' in dem wir die X-Koordinate von R' modulo n rechnen. r' = x mod n

> Wenn r' und r übereinstimmen, ist die Signatur gültig.

Außerdem ist es wichtig, dass der s-Wert maximal der Hälfte von n entsptricht. Wenn nicht, kommt es beim veröffentlichen der Transaktion dazu, dass einer Fehlermeldung erscheint, welche besagt, dass die Transaktion nicht canonicalised ist. Durch Anwenden von -s mod n behalten wir weiter eine gültige Signatur, die der Vorgabe entspricht, also dass s maximal der Hälfte von n entspricht.

4.4 Allgemeines zu Hashfunktionen

Was ist eine Hashfunktion?

> Eine Hashfunktion nimmt eine Eingabe beliebiger Größe und erzeugt eine Ausgabe fester Größe, die als Hashwert bezeichnet wird.

Was ist ein Hashwert?

> Der Hashwert ist eine eindeutige Darstellung der Eingabe und wird häufig als eine Art Fingerabdruck oder digitale Signatur der Eingabe verwendet.

4.5 SHA256

SHA256 ist eine kryptographishe Hashfunktion, die für die Erzeugung von Hashwerten für Datenblöcke verwendet wird und ist Teil des SHA-2 Standards.

Wie funktioniert eine SHA-256-Hashfunktion?

> Die SHA-256-Hashfunktion erzeugt einen 256-Bit-Hashwert, der oftmals in Form von Hexadezimal-Zahlen dargestellt wird. Die Hashfunktion ist kryptographisch sicher, was bedeutet, dass es praktisch unmöglich ist, zwei verschiedene Eingaben zu finden, die denselben Hashwert erzeugen (eine sogenannte Kollision).

Warum wird SHA-256 verwendet?

> SHA-256 wird in vielen kryptographischen Anwendungen verwendet, wie z.B. in der digitalen Signatur, bei der der Hashwert eines Dokuments oder einer Nachricht berechnet wird und dann mit dem privaten Schlüssel des Unterzeichners verschlüsselt wird, um eine digitale Signatur zu erzeugen. Die digitale Signatur kann dann vom Empfänger mit dem öffentlichen Schlüssel des

> Unterzeichners verifiziert werden, indem der Hashwert des Dokuments erneut berechnet und mit der Entschlüsselung der digitalen Signatur verglichen wird.

SHA-256 wird auch in vielen Blockchain-Implementierungen verwendet, wie z.B. in Bitcoin und anderen Kryptowährungen, um Transaktionen und Blöcke zu hashen und somit die Integrität der Blockchain zu gewährleisten.

4.6 RipeMD160

Was ist RIPEMD160?

> RIPEMD160 ist eine kryptographische Hashfunktion, die verwendet wird, um eine feste Größe Hashwerte (160-Bit) von Datenblöcken zu erzeugen. Der Name RIPEMD steht für "RACE Integrity Primitives Evaluation Message Digest". Die Hashfunktion wurde von Hans Dobbertin, Antoon Bosselaers und Bart Preneel im Jahr 1996 entwickelt.

Wie funktioniert RIPEMD160?

> RIPEMD160 wurde als Alternative zu der älteren Hashfunktion SHA-1 entwickelt, die im Laufe der Zeit als anfällig für Kollisionen erwiesen wurde. RIPEMD160 ist kryptographisch sicher und es ist praktisch unmöglich, zwei verschiedene Eingaben zu finden, die denselben Hashwert erzeugen (eine sogenannte Kollision).

Warum wird RIPEMD160 verwendet?

> RIPEMD160 wird häufig in der Blockchain-Technologie und in Kryptowährungen wie Bitcoin verwendet, um die Adressen von Empfängern von Transaktionen zu hashen. Es wird auch in anderen Anwendungen wie digitalen Signaturen, Authentifizierung und Schlüsselableitung eingesetzt.

Was ist der Unterschied zwischen RIPEMD160 und SHA-256?

> RIPEMD160 ist schneller als SHA-256, aber da der Hashwert kleiner ist, gibt es theoretisch eine höhere Chance auf Kollisionen. Aus diesem Grund wird RIPEMD160 oft in Kombination mit anderen Hashfunktionen verwendet, um die Sicherheit zu erhöhen. Zum Beispiel wird in Bitcoin-Transaktionen oft SHA-256 und RIPEMD160 nacheinander verwendet, um eine "Hash160" zu erzeugen, die dann als Grundlage für die Erstellung der öffentlichen Adresse des Empfängers verwendet wird.

4.7 Hash160

Die Hash160-Funktion besteht aus doppeltem Hashing mit SHA256 & RipeMD160, wobei zuerst SHA256 und dann RipeMD160 angewandt wird.

4.8 Base58

Base58 ist ein Verfahren zum codieren positiver ganzer Zahlen. Im wesentlichen werden folgende vier Zeichen weggelassen:

> 0 (Null)

> O (großes o)

> I (großes i)

> l (kleines L)

Base58 findet Anwendung bei Bitcoin-Adressen, um eine bessere Lesbarkeit bereitszustellen, da man die genannten Zeichen leicht verwechselt. Somit ist sichergestellt, dass es nicht zu Falscheingaben von Adressen kommt und Transaktionen somit an die richtige Adressen gehen.

> Kodierungstabelle:

Dezimal	Binär	Base58	Dezimal	Binär	Base58	Dezimal	Binär	Base58	Dezimal	Binär	Base58
0	000000	1	16	010000	H	32	100000	Z	48	110000	q
1	000001	2	17	010001	J	33	100001	a	49	110001	r
2	000010	3	18	010010	K	34	100010	b	50	110010	s
3	000011	4	19	010011	L	35	100011	c	51	110011	t
4	000100	5	20	010100	M	36	100100	d	52	110100	u
5	000101	6	21	010101	N	37	100101	e	53	110101	v
6	000110	7	22	010110	P	38	100110	f	54	110110	w
7	000111	8	23	010111	Q	39	100111	g	55	110111	x
8	001000	9	24	011000	R	40	101000	h	56	111000	y
9	001001	A	25	011001	S	41	101001	i	57	111001	z
10	001010	B	26	011010	T	42	101010	j			
11	001011	C	27	011011	U	43	101011	k			
12	001100	D	28	011100	V	44	101100	m			
13	001101	E	29	011101	W	45	101101	n			
14	001110	F	30	011110	X	46	101110	o			
15	001111	G	31	011111	Y	47	101111	p			

4.9 DER (Distinguished Encoding Rules)

DER sind eine Codierung von ASN.1-Datenbeschreibungen, die auf Bitebene völlig eindeutig ist, für jeden ASN.1-Wert(Abstract Syntax Notation One = Beschreibungssprache zur Definition von Datenstrukturen) gibt es also nur eine mögliche Codierung.

DER benutzen wir um eine eindeutige Codierung des r- und s-Wertes der Signatur zu gewährleisten.

Zum Bearbeiten doppelklicken (oder Eingabe)

5 Schlüssel- und Adresserzeugung

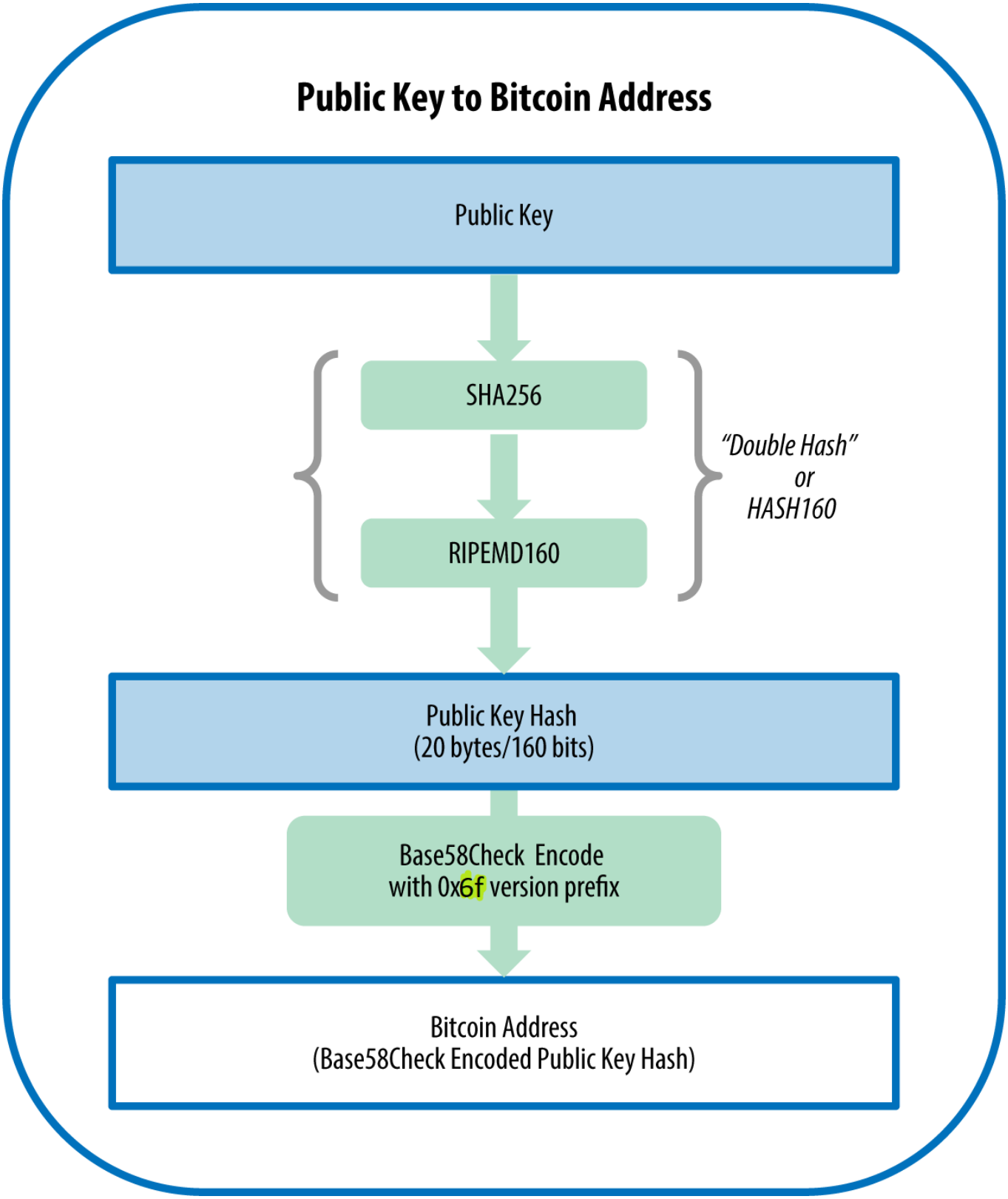
Der folgende Code erzeugt Public Key, Private Key & Adresse. Wir benutzen für die zufällige Erzeugung eines Keys den KeyPairGenerator von Java, welchen wir mit dem Elliptic Curve Algorithmus, bereitgestellt von Bouncycastle, instanzieren. Elliptic Curve Digital Signature Algorithm oder kurz ECDSA nutzen viele verschiedene elliptische Kurven. Bei Bitcoin findet die secp256k1-Kurve Anwendung. Wir nutzen Bouncycyastle als Security Provider nicht nur, weil der Zufallsgenerator sicherer sein soll, da die erzeugten Zahlen nicht auf einen Hashwert der Zeit beruhen, sondern auch, weil die secp256k1-Kurve in eigenen Java-Bibliotheken nicht verfügbar ist.

Unser Code erzeugt zu aller erst einen zufälligen privaten EC Schlüssel und leitet durch die secp256k1 Kurve den dazugehörigen öffentlichen EC Schlüssel ab. Die beiden Schlüssel sind zunächst als KeyPair-Objekt abgespeichert, doch werden für die weitere Bearbeitung aufgetrennt.

Den privaten EC Schlüssel wollen wir nun als erstes in ein für Bitcoin brauchbarer Format umwandeln. Die Methode getS() liefert einen BigInteger-Wert zurück, weshalb wir die Methode toString(16) anhängen, um den BigInteger-Wert in einen Hexadezimal-String umzuwandeln. Gelegentlich kommt es vor, dass der so erzeugte Schlüssel nicht 32 byte groß ist, weshalb er mit bis zu zwei führenden Nullen aufgefüllt wird. Ist der erzeugte Schlüssel kleiner als 31 byte, ist er nicht zu gebrauchen, und das Programm bricht ab. Nach dem Auffüllen ist unser private Bitcoin Schlüssel fertig.

Auch den öffentlichen EC Schlüssel müssen wir nun bearbeiten, so dass er für Bitcoin brauchbar wird. Durch die Anwendung des ECDSA mit der secp256k1-Kurve auf den privaten EC Schlüssel ergibt sich ein Koordinatenpaar, welches durch die Methode getW() als ECPoint-Objekt gespeichert wird. Durch die Methoden getAffineX()/getAffineY() kriegen wir die x- und y-Koordinaten als BigInteger-Werte zurückgeliefert und formen diese mit toString(16) in Hexadezimal-Strings um. Wir nutzen der Einfachheit halber unkomprimierte Schlüssel, wobei man das eigentlich nicht mehr macht, da man mit komprimierten Schlüsseln/Adressen Transaktionsgebühren spart. Der unkomprimierte, öffentliche Bitcoinschlüssel ergibt sich aus dem Prefix "04"(steht für unkomprimiert) sowie der x- und y-Koordinate. Auch hier werden die Koordinaten jeweils darauf geprüft, ob sie jeweils 32 byte groß sind, wie oben schon beim privaten Schlüssel erklärt. Das Endresultat ist ein 64 byte großer Schlüssel, mit 1 byte großem Prefix.

Um aus den öffentlichen Schlüssel nun eine Adresse zu erzeugen, müssen wir die Hash160-Funktion anwenden, ein Versionbyte-Prefix anfügen, eine Checksum berechen und diese als Suffix anfügen. Das Ergebnis wird zum Schluss noch mit Base58 codiert.



Die Hash160-Funktion besteht aus zwei verschiedenen Hashfunktionen, SHA256 & RipeMD160. Die Reihenfolge ist wichtig, zuerst SHA256, dann RipeMD160. Das doppelte Hashen mit den beiden Funktionen erzeugt einen 20 byte großen Hash, welcher auch Public Key Hash genannt wird. In unserem Fall, also für Testzwecke, fügen wir den Versionbyte "6f" als Prefix hinzu, der eine Testnet-Adresse signalisiert. Public Key Hash & Prefix werden nun gemeinsamen doppelt mit SHA256 gehashed. Das Ergebnis ist eine 32 byte großer Wert, von dem wir nur die ersten 4 bytes benötigen. Diese 4 bytes sind die Checksum unserer Adresse, welche hinter den Public Key Hash als Suffix angefügt wird. Zum Schluss müssen wir das ganze noch mit Base58 codieren, was für eine bessere Lesbarkeit sorgt, da leicht zu verwechselnde Zeichen wie 0 (Null), O (großes o), I (großes i) und l (kleines L) weggelassen werden.

Die fertige Adresse setzt sich dann wie folgt zusammen: Base58(6f + Public Key Hash + Checksum)

Sie ist mit Prefix & Suffix zusammen 25 byte groß.

```
import java.nio.ByteBuffer;
import java.security.InvalidAlgorithmParameterException;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.NoSuchProviderException;
import java.security.Security;
import java.security.interfaces.ECPrivateKey;
import java.security.interfaces.ECPublicKey;
import java.security.spec.ECGenParameterSpec;
import java.security.spec.ECPoint;
import org.bitcoinj.core.Base58;
import org.bouncycastle.crypto.digests.RIPEMD160Digest;
import org.bouncycastle.crypto.digests.SHA256Digest;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.util.encoders.Hex;


        ECPublicKey ecPubKey;
        ECPrivateKey ecPrivKey;

        Security.addProvider(new BouncyCastleProvider());
        KeyPairGenerator ecKeyGen = KeyPairGenerator.getInstance("EC", "BC");
        ECGenParameterSpec ecSpec = new ECGenParameterSpec("secp256k1");
        ecKeyGen.initialize(ecSpec);
        KeyPair ecKeyPair = ecKeyGen.generateKeyPair();
        ecPubKey = (ECPublicKey) ecKeyPair.getPublic();
        ecPrivKey = (ECPrivateKey) ecKeyPair.getPrivate();

        String privateKeyHex = ecPrivKey.getS().toString(16);

        if (privateKeyHex.length() != 64) {
            if (privateKeyHex.length() == 63)
                privateKeyHex = "0" + privateKeyHex;
            else if (privateKeyHex.length() == 62)
                privateKeyHex = "00" + privateKeyHex;
            else
                System.out.println("not a valid private key");
                System.exit(0);
        }

        ECPoint ecPoint = ecPubKey.getW();
        String pointXhex = ecPoint.getAffineX().toString(16);
        String pointYhex = ecPoint.getAffineY().toString(16);

        if (pointXhex.length() != 64) {
            if (pointXhex.length() == 63)
                pointXhex = "0" + pointXhex;
            else if (pointXhex.length() == 62)
                pointXhex = "00" + pointXhex;
            else
                System.out.println("not a valid public key");
                System.exit(0);
        }

        if (pointYhex.length() != 64) {
            if (pointYhex.length() == 63)
                pointYhex = "0" + pointYhex;
            else if (pointYhex.length() == 62)
                pointYhex = "00" + pointYhex;
            else
                System.out.println("not a valid public key");
                System.exit(0);
        }

        String publicKeyHex = "04" + pointXhex + pointYhex;
        // 04 für unknomprimierter Public Key
        byte[] pubKeyBytes = Hex.decode(publicKeyHex);

        SHA256Digest shaDigest = new SHA256Digest();
        RIPEMD160Digest rmdDigest = new RIPEMD160Digest();

        shaDigest.update(pubKeyBytes, 0, pubKeyBytes.length);
        byte[] pubKeyShaHashed = new byte[shaDigest.getDigestSize()];
        shaDigest.doFinal(pubKeyShaHashed, 0);
        rmdDigest.update(pubKeyShaHashed, 0, pubKeyShaHashed.length);
        byte[] pubKeyHash = new byte[rmdDigest.getDigestSize()];
        rmdDigest.doFinal(pubKeyHash, 0);
        // 6f für Testnet als Prefix einfügen
        byte[] pubKeyHashWithPrefix = ByteBuffer.allocate(pubKeyHash.length + 1).put(Hex.decode("6f")).put(pubKeyHash)
                .array();

```

```
        shaDigest.update(pubKeyHashWithPrefix, 0, pubKeyHashWithPrefix.length);

        byte[] doubleSha = new byte[shaDigest.getDigestSize()];
        shaDigest.doFinal(doubleSha, 0);
        shaDigest.update(doubleSha, 0, doubleSha.length);
        shaDigest.doFinal(doubleSha, 0);
        byte[] checksum = new byte[4];
        for (int i = 0; i < checksum.length; i++)
            checksum[i] = doubleSha[i];

        byte[] address = ByteBuffer.allocate(pubKeyHashWithPrefix.length + checksum.length).put(pubKeyHashWithPrefix)
                .put(checksum).array();

        String addressBase58 = Base58.encode(address);

        System.out.println("Private Key: " + privateKeyHex + "\nUncompressed Public Key: " + publicKeyHex
                + "\nUncompressed Address: " + addressBase58);


```

```
Private Key: 5eea8628771ae5def10f8932c2e411de724978cc1a87e9cb8a169411b53906aa
Uncompressed Public Key: 049e74476727846f031c41ce5d06cd7a751dfe568e74bb8351e6d58c9228066bce878ded316dc0bdabb4a3247c0731ab02737cd96c
Uncompressed Address: mxfz39WY5XX3rtZET72VjnzThhLPX1KM2z
```

Das Ergebnis der Ausgabe halten wir fest.

> Privater Schlüssel: 5eea8628771ae5def10f8932c2e411de724978cc1a87e9cb8a169411b53906aa

> Unkomprimierter öffentlicher Schlüssel:
> 049e74476727846f031c41ce5d06cd7a751dfe568e74bb8351e6d58c9228066bce878ded316dc0bdabb4a3247c0731ab02737cd
> 96cf7740268c75733d53b94c160

> Adresse: mxfz39WY5XX3rtZET72VjnzThhLPX1KM2z

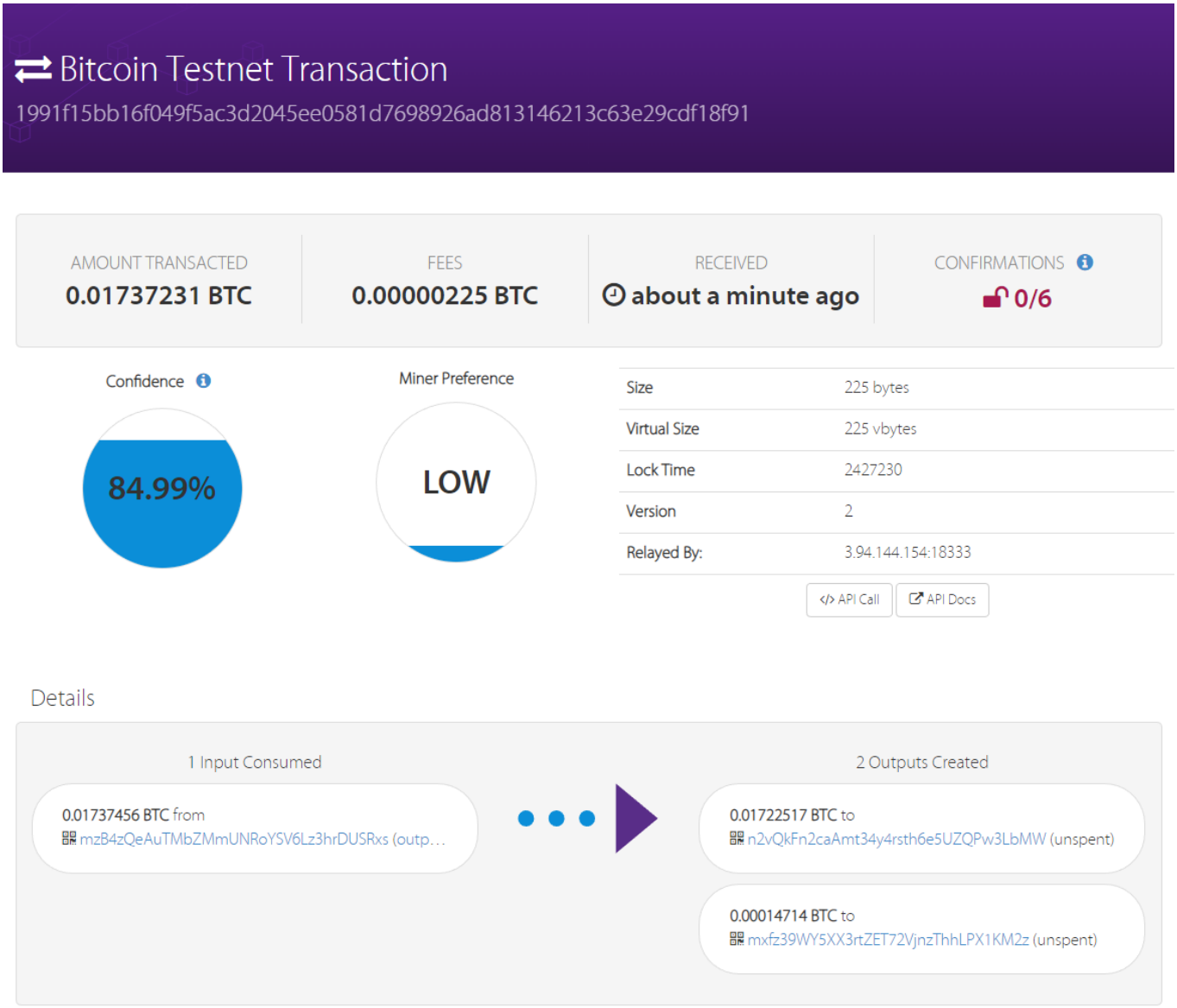
6 Testcoins anfordern

Über die Seite https://testnet-faucet.com/btc-testnet/ schicken wir uns nun Testcoins an unsere eben erzeugte Adresse.

Transaktions-ID: 1991f15bb16f049f5ac3d2045ee0581d7698926ad813146213c63e29cdf18f91

Die Transaktion können wir nun mithilfe des Blockchain Browsers Blockcypher tracken.

https://live.blockcypher.com/btc-testnet/tx/1991f15bb16f049f5ac3d2045ee0581d7698926ad813146213c63e29cdf18f91/



7 Begriffserklärungen zu Transaktionen

7.1 Pay-to-Public-Key-Hash (P2PKH)

In diesem Projekt finden P2PKH-Adressen & -Transaktionen Anwendung.

Pay-to-Public-Key-Hash (P2PKH) ist ein Transaktionsformat in Bitcoin und anderen Kryptowährungen, bei dem eine Zahlung an eine Empfängeradresse erfolgt, die aus einem PublicKey-Hash abgeleitet wurde. Der Public-Key-Hash wird aus dem öffentlichen Schlüssel des Empfängers berechnet und stellt eine verkürzte und verschleierte Form des öffentlichen Schlüssels dar.

P2PKH-Adressen beginnen normalerweise mit der Ziffer "1" (für das Hauptnetzwerk) oder "m" (für das Testnetzwerk) und sind etwa 34 Zeichen lang. Wenn eine Transaktion an eine P2PKH-Adresse gesendet wird, muss der Empfänger den zugehörigen privaten Schlüssel kennen, um die ausgehende Transaktion signieren und die darin enthaltenen Mittel ausgeben zu können.

P2PKH ist eine der gängigsten Transaktionsarten in Bitcoin und anderen Kryptowährungen, da sie eine gute Balance zwischen Sicherheit und Benutzerfreundlichkeit bietet.

7.2 Inputs & Outputs

7.2.1 Inputs (UTXO's)

Inputs stellen die Grundlage von Transaktionen in Kryptowährungen dar. Sie sind unverzichtbar, da sie den Nachweis darüber erbringen, dass eine bestimmte Menge an Kryptowährungen an den Sender der Transaktion geschickt wurde. Im Kontext von Bitcoin und anderen ähnlichen Kryptowährungen bezieht sich der Begriff "inputs" auf Unspent Transaction Outputs (UTXOs). Das sind Transaktionen, die bereits an eine Bitcoin-Adresse gesendet wurden, aber noch nicht ausgegeben wurden.

Sobald eine Transaktion durchgeführt wird, indem man Kryptowährungen an eine Adresse schickt, wird diese Transaktion als Ausgabe (output) bezeichnet. Eine Ausgabe wird zum Eingang (input) einer neuen Transaktion, wenn sie als unspent transaction output in einer Blockchain

gespeichert wird. Sobald diese unspent transaction output als Eingang in einer neuen Transaktion verwendet wird, wird er als ausgegeben (spent) betrachtet und kann nicht mehr in einer weiteren Transaktion verwendet werden.

Der Prozess der Verwendung von inputs in einer Transaktion kann komplex sein, da mehrere UTXOs zusammengeführt werden können, um eine größere Menge an Kryptowährungen zu senden. Wenn der Wert des Outputs einer Transaktion größer als der Wert des Inputs ist, wird die Differenz als Transaktionsgebühr genommen, die an den Miner gezahlt wird, der die Transaktion bestätigt.

Insgesamt sind inputs ein entscheidender Bestandteil des Kryptowährungsökosystems. Ohne sie könnte keine Transaktion durchgeführt werden und die Sicherheit und Integrität der Blockchain wären gefährdet.

7.2.2 Outputs

Outputs sind ein wesentlicher Bestandteil von Bitcoin-Transaktionen und enthalten Informationen darüber, an wen die ausgegebene Menge an Bitcoins gehen soll. Jeder Output hat eine bestimmte Menge an Bitcoins, die ausgegeben wird, sowie ein öffentliches Schlüsselskript (publickeyscript), das die Empfängeradresse enthält. Der öffentliche Schlüsselskript wird verwendet, um den Empfänger der Transaktion zu identifizieren und zu authentifizieren. Um eine Transaktion erfolgreich abzuschließen, müssen alle Outputs einer Transaktion zusammen mit einer gültigen Signatur der zugehörigen Private Keys der Inputs verarbeitet werden. Outputs können auch dazu verwendet werden, um eine sogenannte Multi-Signatur-Transaktion zu erstellen, bei der mehrere Parteien gemeinsam über eine Transaktion entscheiden. Insgesamt sind Outputs ein wichtiger Bestandteil der Bitcoin-Blockchain-Technologie und ermöglichen es den Benutzern, auf sichere und zuverlässige Weise Transaktionen auszuführen.

7.3 Raw-Transaction

Eine Raw-Transaction ist eine Transaktion, die in Form von Hexadezimalcode dargestellt wird und besteht grob gesagt aus Inputs & Outputs. Diese werden zusammen mit einer Mengenangabe, der Version des Transaktionsprotokoll und wenn gewünscht bestimmten Locktimes zusammen gesetzt.

7.4 Signierte Transaktion

Zum Signieren benötigen wir zunächst eine Raw-Transaction.

Vor dem Signieren beinhalten die Inputs den eigenen ScriptPubKey, welcher nach dem Signieren durch die scriptSig ersetzt wird.

Die Raw-Transaction wird doppelt mit SHA256 gehashed und mit dem privaten Schlüssel signiert. Die Signatur wird dann im DER-Format decodiert und mit dem öffentlichen Schlüssel zu einem String zusammengesetzt, wodurch wir die scriptSig erhalten.

Setzt man die scriptSig in die Raw-Transaction ein erhält man ein signiertes Transaktionsskript, welches im Anschluss veröffentlicht werden kann.

8 Transaktionsinformationen abfragen

Dem vorherigen Bildausschnitt kann man viele Informationen entnehmen, welche wir später für das versenden einer Transaktion benötigen, jedoch nicht alle, weshalb wir die Funktion "API Call" nutzen, wodurch wir folgendes sehen: (Bildausschnitt von https://api.blockcypher.com/v1/btc/test3/txs/1991f15bb16f049f5ac3d2045ee0581d7698926ad813146213c63e29cdf18f91?limit=50&includeHex=true)

```
{
  "block_hash": "0000000000000007f57d64a8285217aaf32479a59f3a31bd87b0e2a797668c71",
  "block_height": 2427261,
  "block_index": 25,
  "hash": "1991f15bb16f049f5ac3d2045ee0581d7698926ad813146213c63e29cdf18f91",
  "hex":
"020000000164ef32adba755a7a5ba6cb50b18ae833a98101f4d9a6092b1559eb1b21bcc679000000006a473044022039604356f279de2c8d6a4f88ac7a390000000000001976a914bc2f5642c576d9675eb67c8dcee169f4c5b098ab88ac5e092500",
  "addresses": [
    "mxfz39WY5XX3rtZET72VjnzThhLPX1KM2z",
    "mzB4zQeAuTMbZMmUNRoYSV6Lz3hrDUSRxs",
    "n2vQkFn2caAmt34y4rsth6e5UZQPw3LbMW"
  ],
  "total": 1737231,
  "fees": 225,
  "size": 225,
  "vsize": 225,
  "preference": "low",
  "relayed_by": "3.94.144.154:18333",
  "confirmed": "2023-04-03T19:51:29Z",
  "received": "2023-04-03T19:48:51.226Z",
  "ver": 2,
  "lock_time": 2427230,
  "double_spend": false,
  "vin_sz": 1,
  "vout_sz": 2,
  "confirmations": 2,
  "confidence": 1,
  "inputs": [
    {
      "prev_hash": "79c6bc211beb59152b09a6d9f40181a933e88ab150cba65b7a5a75baad32ef64",
      "output_index": 0,
      "script": "473044022039604365f6032828812abf04589402a5059d19554bea51331a2010b8344176302201 22ac3
      "output_value": 1737456,
      "sequence": 4294967294,
      "addresses": [
        "mzB4zQeAuTMbZMmUNRoYSV6Lz3hrDUSRxs"
      ],
      "script_type": "pay-to-pubkey-hash",
      "age": 2427260
    }
  ],
  "outputs": [
    {
      "value": 1722517,
      "script": "76a914eaca4a157b04c28c29bbe53456f279de2c8d6a4f88ac",
      "addresses": [
        "n2vQkFn2caAmt34y4rsth6e5UZQPw3LbMW"
      ],
      "script_type": "pay-to-pubkey-hash"
    },
    {
      "value": 14714,
      "script": "76a914bc2f5642c576d9675eb67c8dcee169f4c5b098ab88ac",
      "addresses": [
        "mxfz39WY5XX3rtZET72VjnzThhLPX1KM2z"
      ],
      "script_type": "pay-to-pubkey-hash"
    }
  ]
}
```


Die API-Abfrage enthält alle wichtigen Informationen für das Erstellen einer Transaktion. Wir sehen die Transaktions-ID, welche hier einfach "hash" genannt wird, die Menge Testcoins (in Satoshi) die an uns verschickt wurde, "value" genannt, gelistet unter Outputs, den scriptPubKey, genannt "script" und den Outputindex, also die Stelle, an der unsere Adresse unter Outputs in der Transaktion steht. Der Index startet bei 0 und unsere Adresse steht an zweiter Stelle, also Output-Index = 1.

Transaktions-ID: 1991f15bb16f049f5ac3d2045ee0581d7698926ad813146213c63e29cdf18f91

Menge: 14714

scriptPubKey: 76a914bc2f5642c576d9675eb67c8dcee169f4c5b098ab88ac

Output-Index: 1

Diese Parameter berechtigen uns zusammen mit unserem Private Key, die empfangene Transaktion auszugeben. Es fehlt nur noch eine Adresse, wohin wir die Coins verschicken können.

9 Transaktion erstellen und signieren

Da der Betreiber von https://testnet-faucet.com/btc-testnet/ sich über die Rückgabe der Testcoins freut, schicken wir Sie ihm abzüglich einer Gebühr für die Miner zurück.

Please return your unused testnet coins to this address so other people can use them!

mohjSavDdQYHRYXcS3uS6ttaHP8amyvX78

Wir übergeben die Transaktions-ID, den scriptPubKey, den Output-Index, die Menge und die Adresse dem untenstehenden Programmcode, um eine signierte Transaktion zu erstellen, bzw. genauer gesagt ein Skript, welches die Transaktion ausführt.

```
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import org.bitcoinj.core.Base58;
import org.bitcoinj.core.ECKey;
import org.bitcoinj.core.ECKey.ECDSASignature;
import org.bitcoinj.core.Sha256Hash;
import org.bouncycastle.util.encoders.Hex;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;


    String txVersion = "02000000";
    String inputs = "01";
        String txId = "1991f15bb16f049f5ac3d2045ee0581d7698926ad813146213c63e29cdf18f91"; // Hier tragen wir die Transaktions-ID der e
        int vout_n = 1; // Hier tragen wir den Output-Index der Transaktion ein.
        String scriptPubKey = "76a914bc2f5642c576d9675eb67c8dcee169f4c5b098ab88ac"; // Hier tragen wir das Script der empfangenen Trar
        String sighash = "01";
        String sequence = "ffffffff";
        String outputs = "01";
        int outputAmount = 14000; // Wir schicken 14000 Satoshi zurück an den Absender, den Rest überlassen wir den Minern als Gebühr.
        String receiveAddress = "mohjSavDdQYHRYXcS3uS6ttaHP8amyvX78"; // Testnet Adresse von testnet-faucet.com
        String privateKey = "5eea8628771ae5def10f8932c2e411de724978cc1a87e9cb8a169411b53906aa"; // Der passende Private Key zu unserer
        String locktime = "00000000";
        String sighashall = "01000000";
        String scriptCodes1 = "76a914"; // Script Code für duplizieren, hash160, und Länge des scriptPubKey
        String scriptCodes2 = "88ac"; // Scriptcodes für ...

        // LE = LittleEndian
        byte[] txIdBytes = Hex.decode(txId);
        byte[] txIdBytes_LE = new byte[txIdBytes.length];
        for (int i = 0; i < txIdBytes.length; i++) {
            txIdBytes_LE[i] = txIdBytes[txIdBytes.length - i - 1];
        }
        String txId_LE = Hex.toHexString(txIdBytes_LE);

        String vout = Integer.toHexString(vout_n);
        int diff = 8 - vout.length();

        for (int i = 0; i < diff; i++) {
            vout = "0" + vout;
        }

        byte[] voutBytes = Hex.decode(vout);
        byte[] voutBytes_LE = new byte[voutBytes.length];
        for (int i = 0; i < voutBytes.length; i++) {
            voutBytes_LE[i] = voutBytes[voutBytes.length - i - 1];
        }
        String vout_LE = Hex.toHexString(voutBytes_LE);
        String scriptSize = Integer.toHexString((scriptPubKey.length() / 2));

        String outputAmountHex = Integer.toHexString(outputAmount);

        diff = 16 - outputAmountHex.length();

        for (int i = 0; i < diff; i++) {
            outputAmountHex = "0" + outputAmountHex;
        }

        byte[] outputAmountBytes = Hex.decode(outputAmountHex);
        byte[] outputAmountBytes_LE = new byte[outputAmountBytes.length];
        for (int i = 0; i < outputAmountBytes.length; i++) {
            outputAmountBytes_LE[i] = outputAmountBytes[outputAmountBytes.length - i - 1];
        }
        String outputAmount_LE = Hex.toHexString(outputAmountBytes_LE);

        byte[] receivePubKeyHashWithPrefix = Base58.decode(receiveAddress);


        byte[] receivePubKeyHashBytes = new byte[receivePubKeyHashWithPrefix.length - 5];
        for (int i = 0; i < receivePubKeyHashBytes.length; i++) {
            receivePubKeyHashBytes[i] = receivePubKeyHashWithPrefix[i + 1];
        }
        String receivePubKeyHash = Hex.toHexString(receivePubKeyHashBytes);
        String outputScript = scriptCodes1 + receivePubKeyHash + scriptCodes2;
        String outputScriptLength = Integer.toHexString(outputScript.length() / 2);
        String rawTX = txVersion + inputs + txId_LE + vout_LE + scriptSize + scriptPubKey + sequence + outputs
                + outputAmount_LE + outputScriptLength + outputScript + locktime + sighashall;
```

```
78
79
80
81    byte[] data = Hex.decode(rawTX);
82        MessageDigest sha256 = MessageDigest.getInstance("SHA-256");
83        byte[] firstHash = sha256.digest(data);
84        byte[] secondHash = sha256.digest(firstHash);
85
86        ECKey ecKey = ECKey.fromPrivate(Hex.decode(privateKey), false);
87        ECDSASignature sig = ecKey.sign(Sha256Hash.wrap(secondHash));
88
89        sig.toCanonicalised();
90
91        byte[] derSigned = ByteBuffer.allocate(sig.encodeToDER().length + 1).put(sig.encodeToDER())
92                .put(Hex.decode(sighash)).array();
93
94        String pubKeyHex = ecKey.getPublicKeyAsHex();
95        int sigLength = derSigned.length;
96        int pubKeyLength = pubKeyHex.length() / 2;
97
98        String scriptSig = Integer.toHexString(sigLength) + Hex.toHexString(derSigned)
99                + Integer.toHexString(pubKeyLength) + pubKeyHex;
100
101       String scriptSigSize = Integer.toHexString(scriptSig.length() / 2);
102
103       String signedTX = txVersion + inputs + txId_LE + vout_LE + scriptSigSize + scriptSig + sequence + outputs
104               + outputAmount_LE + outputScriptLength + outputScript + locktime;
105
106       System.out.println("Transaction Hex Code: " + signedTX);
107
108
```

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Transaction Hex Code: 0200000001918ff1cd293ec613621413d86a9298761d58e05e04d2c35a9f046fb15bf19119010000008b483045022100fca92b9c7295b
```

```
0200000001918ff1cd293ec613621413d86a9298761d58e05e04d2c35a9f046fb15bf19119010000008b483045022100fca92b9c7295b10b4450cfd4069d3b7
9e4ef8f127bce1a57a2b6c831c4d63f7f02200fa2989012a7b698e87fc2d70b9eff1ebb02f66dd7012cc5b754a546030a32260141049e74476727846f031c41c
e5d06cd7a751dfe568e74bb8351e6d58c9228066bce878ded316dc0bdabb4a3247c0731ab02737cd96cf7740268c75733d53b94c160ffffffff01b0360000000
000001976a91459cada50314c829e19f5a7786f8ee0d4987f429d88ac00000000
```

Liefert uns die Ausgabe des Programms. Dieser lange Hexcode stellt ein Skript zum Ausführen einer Bitcoin-Transaktion dar.

10 Transaktionsskript überprüfen und veröffentlichen

Zunächst decodieren wir das Transaktionsskript von Hand um die einzelnen Bestandteile zu erläutern.

02 00 00 00 // Version des Transaktionsprotokelles

01 // Anzahl von Inputs, bzw UTXO's die für die Transaktion verbraucht werden

91 8f f1 cd 29 3e c6 13 62 14 13 d8 6a 92 98 76 1d 58 e0 5e 04 d2 c3 5a 9f 04 6f b1 5b f1 91 19 // Die Transaktions-ID des UTXO im Little-Endian Format

01 00 00 00 // Output-Index im Little-Endian Format

8b // Größe der scriptSig

48 // Größe der gesamten Signatur

30 45 02 21 00 fc a9 2b 9c 72 95 b1 0b 44 50 cf d4 06 9d 3b 79 e4 ef 8f 12 7b ce 1a 57 a2 b6 c8 31 c4 d6 3f 7f 02 20 0f a2 98 90 12 a7 b6 98 e8 7f c2 d7 0b 9e ff 1e bb 02 f6 6d d7 01 2c c5 b7 54 a5 46 03 0a 32 26 // Signatur DER-codiert

01 //SIGHASH_ALL Optcode

41 // Größe des Öffentlichen Schlüssels

04 // Art des Öffentlichen Schlüssels --> 04 = Unkomprimiert

9e 74 47 67 27 84 6f 03 1c 41 ce 5d 06 cd 7a 75 1d fe 56 8e 74 bb 83 51 e6 d5 8c 92 28 06 6b ce // X-Koordinate des Öffentlichen Schlüssel

87 8d ed 31 6d c0 bd ab b4 a3 24 7c 07 31 ab 02 73 7c d9 6c f7 74 02 68 c7 57 33 d5 3b 94 c1 60 // Y-Koordinate des Öffentlichen Schlüssel

ff ff ff ff // Wir nutzen keine spezielle Locktime, deswegen belassen wir den Punkt beim Standard

01 // Anzahl der Outputs, also Adressen die wir begünstigen

b0 36 00 00 00 00 00 00 // Menge der Satoshi im Little-Endian Format

19 // Größe des ScriptPubKey

76 // Opt-Code für Duplizieren

a9 // Opt-Code für Hash160

14 // Größe des Public Key Hash

59 ca da 50 31 4c 82 9e 19 f5 a7 78 6f 8e e0 d4 98 7f 42 9d // Public Key Hash

88 // Opt-Code für equalverify

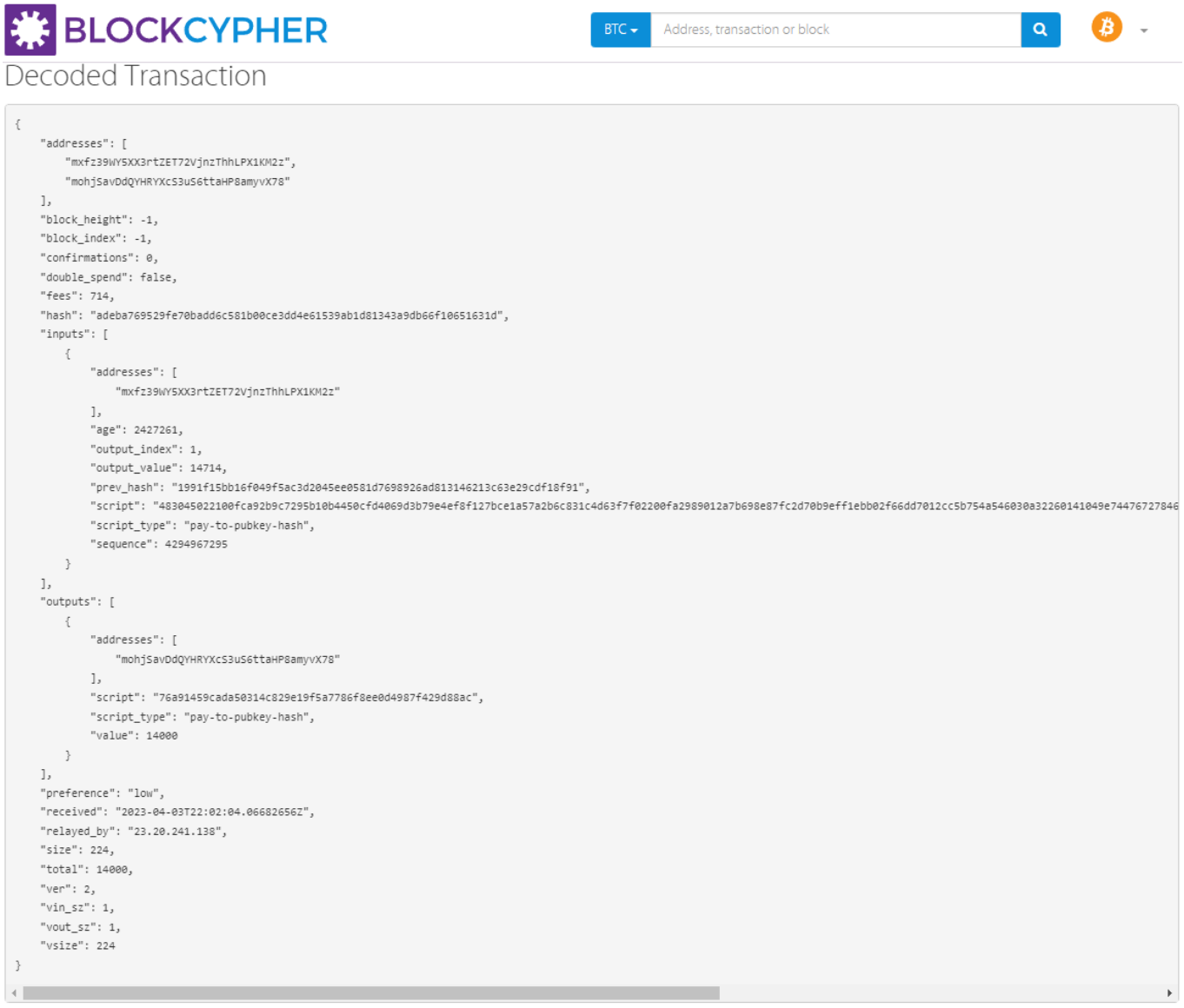
ac // Opt-Code für checksig

00 00 00 00 // Keine Nutzung von Locktime, Standardwert

> Aber es gibt auch Tools mit denen man die Transaktion decodieren und untersuchen kann.
> https://live.blockcypher.com/btc/decodetx/

> Vor dem finalen Versenden der Transaktion schauen wir uns unter unter der genannten Website an, ob alle Parameter stimmen.

> Wir sehen keine Nullwerte und alle Parameter sind vernünftig initialisiert, weshalb wir die Transaktion nun veröffentlichen, dafür gehen wir auf https://live.blockcypher.com/btc/pushtx/ und wählen das Bitcoin Testnet aus.



Input New Transaction Hex

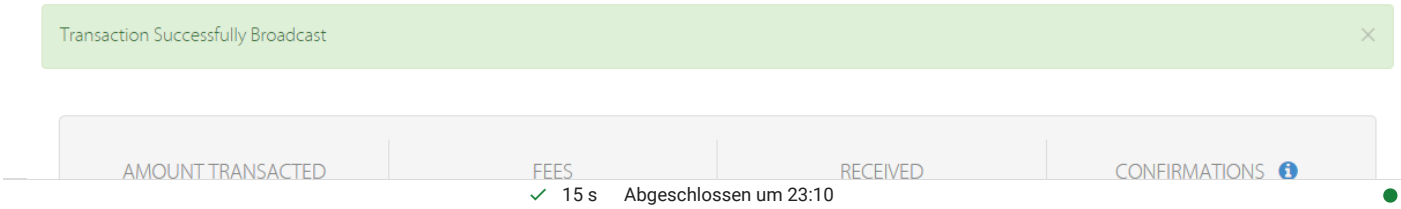
Transaction Hex*

```
0200000001918ff1cd293ec613621413d86a9298761d58e05e04d2c35a9f046fb15bf19119010000008b483045022100fca92b9c7295b10b4450cfd4069d3b79
e4ef8f127bce1a57a2b6c831c4d63f7f02200fa2989012a7b698e87fc2d70b9eff1ebb02f66dd7012cc5b754a546030a32260141049e74476727846f031c41ce5d
06cd7a751dfe568e74bb8351e6d58c9228066bce878ded316dc0bdabb4a3247c0731ab02737cd96cf7740268c75733d53b94c160ffffffff01b03600000000000
01976a91459cada50314c829e19f5a7786f8ee0d4987f429d88ac00000000
```

> Die Transaktion wurde erfolgreich veröffentlicht.

> Transaktions-ID: adeba769529fe70badd6c581b00ce3dd4e61539ab1d81343a9db66f10651631d

> https://live.blockcypher.com/btc-testnet/tx/adeba769529fe70badd6c581b00ce3dd4e61539ab1d81343a9db66f10651631d/



✓ 15 s Abgeschlossen um 23:10