Blockchain Engineering

Public Key Kryptografie und Digitale Signaturen

Dr. Lars Brünjes

MODULARES INNOVATIVES
NETZWERK FÜR DURCHLÄSSIGKEIT

26. September 2019

- In traditionellen symmetrischen Verschlüsselungsmethoden gibt es nur einen Schlüssel, der sowohl zum Verschlüsseln (encryption) als auch zum Entschlüsseln (decryption) verwendet wird. Dies Methoden haben das Problem, dass sie nur sicher sind, solange der Schlüssel geheim bleibt. Insbesondere muss der Schlüssel zunächst zwischen Sender und Empfänger *ausgetauscht* werden, bevor verschlüsselt kommuniziert werden kann.
- Public Key Kryptografie ist asymmetrisch, und kein Schlüsseltausch ist nötig: Benutzer generieren zunächst Schlüssel-Paare (key pairs), bestehend aus einem öffentlichen Schlüssel (public key) und einem privaten/geheimen Schlüssel (private/secret key). Der öffentliche Schlüssel wird veröffentlicht, der private Schlüssel muss geheim bleiben.
- Verschlüsseln und Entschlüsseln verwenden denselben Algorithmus, allerdings wird zum Verschlüsseln der *öffentliche Schlüssel des Empfängers* verwendet, während der Empfänger seinen *privaten* Schlüssel zum Entschlüsseln benutzt.

- Zuerst 1973 von dem britischen Mathematiker *Cliffort Cocks* beschrieben, der allerdings für den britischen Geheimdienst GCHQ arbeitete, so dass seine Erkenntnisse erst 1997 veröffentlicht wurden.
- Die Idee wird *Whitfield Diffie* und *Martin Hellman* zugeschrieben, die das Konzept 1976 veröffentlichten.
- *Ron Rivest*, *Adi Shamir* und *Leonard Adleman* vom Massachusetts Institute of Technology (MIT) veröffentlichten den ersten praktischen Algorithmus 1977.

- Einer der ältesten und bekanntesten Public Key Algorithmen.
- Benannt nach Rivest, Shamir und Adleman.
- Die Sicherheit von RSA beruht auf der Tatsache, dass es zwar leicht ist, sehr große Primzahlen (mit mehr als hundert Ziffern) zu *multiplizieren*, dass es aber sehr schwer ist, eine sehr große Zahl in ihre Primfaktoren zu zerlegen. Diese Asymmetrie liegt RSA zugrunde.

```python
def gen_key_pair(p, q):

    if not is_prime(p):
        raise Exception("%d␣ist␣nicht␣prim." % p)
    if not is_prime(q):
        raise Exception("%d␣ist␣nicht␣prim." % q)
    if p == q:
        raise Exception("zwei␣verschiedene␣Primzahlen␣erwartet")

    n = p * q
    ln = lcm(p - 1, q - 1)
    e = next(e for e in range(2, ln) if gcd(e, ln) == 1)
    d = invert(e, ln)

    return {'n': n, 'e': e}, {'n': n, 'e': d}

def encrypt(m, pk):
    return pow(m, pk['e'], pk['n'])

def decrypt(c, sk):
    return encrypt(c, sk)
```

```
pa = next_prime(1000)
qa = next_prime(pa)
pka, ska = gen_key_pair(pa, qa)

pb = next_prime(2000)
qb = next_prime(pb)
pkb, skb = gen_key_pair(pb, qb)

m1 = 777777
c1 = encrypt(m1, pkb)
print("Alices␣Nachricht␣an␣Bob:␣%d,␣verschlüsselt:␣%d" % (m1, c1))

d1 = decrypt(c1, skb)
print("Bob␣entschlüsselt␣%d␣als␣%d" % (c1, d1))

m2 = 333333
c2 = encrypt(m2, pka)
print("Bobs␣Nachricht␣an␣Alice:␣%d,␣verschlüsselt:␣%d" % (m2, c2))

d2 = decrypt(c2, ska)
print("Alice␣entschlüsselt␣%d␣als␣%d" % (c2, d2))
```

RSA Beispiel (Ergebnis)

```
Alices Nachricht an Bob: 777777, verschlüsselt: 970666
Bob entschlüsselt 970666 als 777777
Bobs Nachricht an Alice: 333333, verschlüsselt: 989845
Alice entschlüsselt 989845 als 333333
```

```
from rsa import *

(pk_alice, sk_alice) = newkeys(2048) # 2048 ist Schlüssellänge in Bits
(pk_bob, sk_bob) = newkeys(2048)
c1 = encrypt(b'Hallo,␣Bob!', pk_bob)
print (c1.hex())

# 62607984...0d4487f6 (abhängig von den zufällig generierten Schlüsseln)

print(decrypt(c1, sk_bob))

# b'Hallo, Bob!'
```

- RSA kann nur Nachrichten beschränkter Länge ver- und entschlüsseln (abhängig von Parameter n).
- Längere Nachrichten müssten in Blöcke aufgeteilt werden.
- Außerdem ist RSA (oder allgemein asymmetrische Kryptografie) relativ langsam.
- In der Praxis wird RSA daher meist anders benutzt:
  - Eine Partei erzeugt einen zufälligen Schlüssel für ein *symmetrisches* Verschlüsselungsverfahren wie AES (Advanced Encryption Standard).
  - Dieser Schlüssel wird mittels RSA sicher an die andere Partei übertragen.
  - Der Rest der Kommunikation benutzt AES mit dem Schlüssel, den nun beide Parteien kennen.

- In vielen Blockchains werden anstelle von RSA andere asymmetrische Verschlüsselungsverfahren benutzt.
- Verbreitet (z.B. bei Bitcoin und Cardano) sind Verfahren, die auf elliptischen Kurven basieren.
- Während die Sicherheit von RSA darauf beruht, dass es schwer ist, eine große Zahl in ihre Primfaktoren zu zerlegen, beruht elliptic curve cryptography darauf, dass das "diskrete-Logarithmus-Problem" für elliptische Kurven schwer ist.
- Der Vorteil von Verfahren basierend auf elliptischen Kurven ist die kleinere Schlüssellänge bei gleicher Sicherheit: 256 Bit Schlüssellänge sind so sicher wie 3072 Bit Schlüssellänge bei RSA.

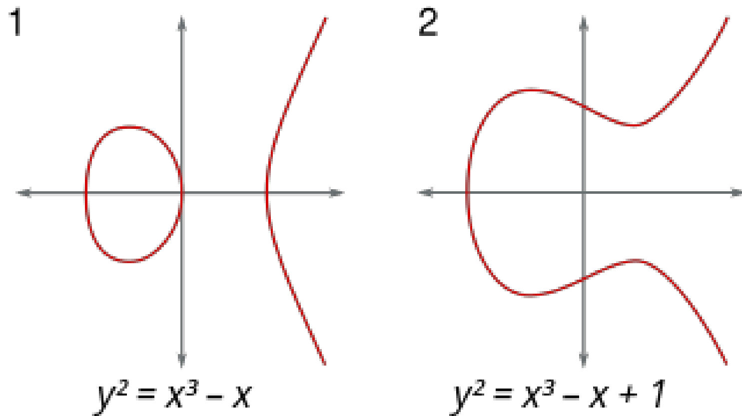

Abb.: Elliptische Kurven

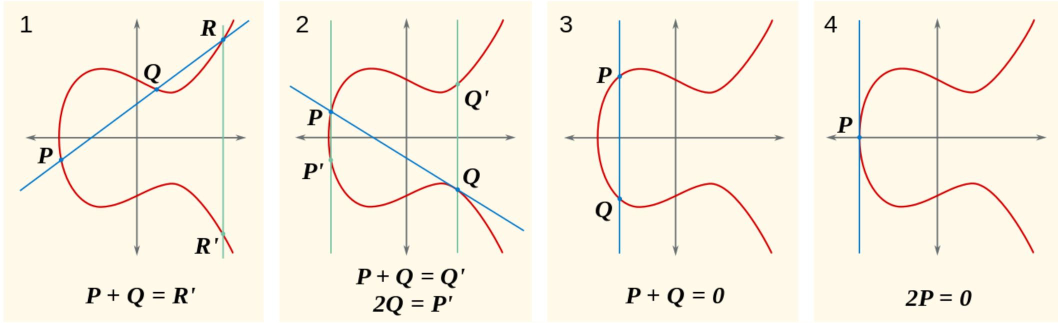

Abb.: Addition auf Elliptische Kurven

Das diskrete-Logarithmus-Problem besagt, dass es leicht ist, nP zu gegebenen P und n zu berechnen, aber schwer, zu gegebenen P und Q ein n mit $nP = Q$ zu finden.

- Asymmetrische Kryptografie an sich wird in Blockchain-Systemen normalerweise nicht benutzt.
- Sie kann aber benutzt werden, um digitale Unterschriften (digital signatures) zu implementieren.
- Eine digitale Unterschrift soll dasselbe für digitale Dokumente leisten, was traditionelle handschriftliche Unterschriften für Dokumente auf Papier leisten, nämlich bestätigen, dass ein Dokument von dem Unterschreibenden als echt bestätigt wird.

Bemerkung

Während eine Unterschrift auf Papier immer (fast) gleich aussieht, kann eine digitale Unterschrift offenbar nicht immer gleich sein, denn dann könnte ein Fälscher sie einfach kopieren und an beliebige Dokumente anhängen. Die digitale Unterschrift muss daher *vom Dokument abhängen*!

Die grundlegende Idee, wie ein asymmetrisches Verschlüsselungsverfahren wie RSA benutzt werden kann, um digitale Signaturen zu erstellen, ist die folgende:

- Das zu unterzeichnende Dokument wird *gehasht* (z.B. mittels SHA-256).
- Der Unterzeichnende verschlüsselt den Hash (z.B. mit RSA) mit seinem *geheimen* Schlüssel.
- Um die digitale Unterschrift zu verifizieren, entschlüsselt der Empfänger sie mit dem *öffentlichen* Schlüssel des Absenders.
- Zuletzt vergleicht der Empfänger den Hash des Dokuments mit der entschlüsselten Unterschrift. Stimmen beide überein, so kann er sicher sein, dass die Unterschrift vom Absender (oder jemandem, der den geheimen Schlüssel des Absenders kennt) stammt.

In der Praxis ist es etwas komplizierter (der Hash wird zunächst mit zufälligen Bytes verlängert. . .).

```
from rsa import *

(pk_alice, sk_alice) = newkeys(2048)
msg = b'Ich,␣Alice,␣bestaetige␣hiermit,␣dass␣ich␣Bob␣1000␣Euro␣ueberweise.'
h = compute_hash(msg, 'SHA-256')
sig = sign_hash(h, sk_alice, 'SHA-256')
print(sig.hex())
# a42fad...844f74

verify(msg, sig, pk_alice)

fake = b'Ich,␣Alice,␣bestaetige␣hiermit,␣dass␣ich␣Bob␣10000␣Euro␣ueberweise.'
verify(fake, sig, pk_alice)
# Traceback (most recent call last):
#       File "sigs.py", line 14, in <module>
#           verify(fake, sig, pk_alice)
#             File "/usr/lib/python3.7/site-packages/rsa/pkcs1.py", line 336, in verify
#                 raise VerificationError('Verification failed')
#             rsa.pkcs1.VerificationError: Verification failed
```

- Digitale Unterschriften sind neben Hashing der zweite fundamentale kryptografische Baustein in der Blockchain Technologie.
- Alle “Artifakte” (Blöcke, Transaktionen,...), die in der Blockchain gespeichert werden, sind digital signiert.

Hinweis

Diese Publikation wurde im Rahmen des vom Bundesministerium für Bildung und Forschung (BMBF) geförderten Bund- Länder- Wettbewerbs "Aufstieg durch Bildung: offene Hochschulen" erstellt. Die in dieser Publikation dargelegten Ergebnisse und Interpretationen liegen in der alleinigen Verantwortung der Autor/innen.