RSA Dokumentation

Was ist RSA?

RSA ist eine asymmetrische Verschlüsselungstechnik. Es wird also für das Verschlüsseln ein anderer Schlüssel verwendet, als für das entschlüsseln. Der gesamte Schlüssel setzt sich also aus einem privaten Schlüssel (Entschlüsselung) und einem öffentlichen Schlüssel (Verschlüsselung) zusammen.

Wie funktioniert RSA?

Das kryptographische Verfahren basiert auf der Grundlage, dass einige mathematische Verfahren so gut wie gar nicht umkehrbar sind. Es gibt also Einwegfunktionen, die nur mit einem bestimmten Parameter gelöst werden können – dem privaten Schlüssel.
Mathematisch funktioniert dies wie folgt:
Ein Empfänger, im Folgenden Alice (nach Rivest, Shamir und Adleman – den Erfindern des Verfahrens), generiert zwei Primzahlen, p und q. Multipliziert ergeben diese n. Auch muss die Zahl m berechnet werden. Diese folgt der Formel: $m=(q-1)(p-1)$. Ebenso muss eine teilerfremde Zahl zu m generiert werden, a. Dies ist beispielsweise dadurch möglich, eine kleiner Primzahl als m zu generieren.
Der Teil des Schlüssels den Alice veröffentlicht ist n und a.
Nun kann ein Sender, im Folgenden Bob, der diesen Schlüssel erlangt eine sicher-verschlüsselte Nachricht an Alice senden.
Diese Nachricht, eine Zahl x, welche kleiner als n sein muss, kann mit Hilfe der Formel: $y = x\hat{}a \ mod \ n$. Verschlüsselt werden.
Alice kann y entschlüsseln, indem sie zunächst das multiplikative Inverse b von $a \ mod \ m$ berechnet. Dies funktioniert mit der Formel: $b = a\hat{}(-1) \ mod \ m$, (Beweis hierfür ist der Satz von Euler-Fermat) oder mit dem erweiterten euklidischen Algorithmus.
Bobs Nachricht kann dann mit der Formel: $x = y\hat{}b \ mod \ n$, wieder entschlüsselt werden.
Möglich ist dies wie gesagt eben nur mit dem Wissen über den privaten Schlüssel und genau hier liegt auch der große Vorteil in der asymmetrischen Verschlüsselung. Um Nachrichten auszutauschen bedarf es keinem Austausch eines Schlüssels, wie bei der symmetrischen Verschlüsselung, also beispielsweise der Caesar- oder Vigenere-Verschlüsselung.

Nun zur Programmierung:
Wie erwähnt muss Alice zunächst einen Schlüssel generieren. Dafür ist folgende Funktion zuständig:

```
    def generate_key(self):
        dict = self.generate_p_q_n_m(200, 210)
        dict["a"] = self.generate_coprime(dict["m"])
        with open("backend_client_side/RSA/key/private_keys.json", "r") as f:
            key = f.read()
        key = json.loads(key)
        key.append(dict)

        with open("backend_client_side/RSA/key/private_keys.json", "w") as f:
```

```
            json.dump(key, f)
```

Es wird ein öffentlicher Schlüssel generiert, mit der Länge zwischen den angegebenen Parametern. In unserem Fall generieren wir Primzahlen mit bis zu 210 Stellen.
Nach der Generierung wird der Schlüssel im JSON-Format in einer Datei gespeichert.

Die Funktion zur Generierung des Schlüssels lautet:

```
    def generate_p_q_n_m(self, lowerValue, upperValue):
        p = self.generate_prime_number(random.randint(lowerValue, upperValue))
        q = self.generate_prime_number(random.randint(lowerValue, upperValue))
        n = p * q
        m = (p - 1) * (q - 1)

        return {"p": p,"q": q,"n": n,"m": m}
```

Diese Funktion generiert die Primzahlen p und q und führt die anschließenden Berechnungen durch. Es wird in dieser Funktion also der Schlüssel generiert außer die teilerfremde Zahl a.

Die Generierung der Primzahlen sieht folgendermaßen aus:

```
    def generate_prime_number(self, length):
        a = "1"
        z = "9"
        for i in range(length):
            a += "0"
            z += "9"
        while True:
            number = random.randint(int(a), int(z))
            if self.is_prime_MRT(number, 8):
                return number
```

Nun wird eine zufällige Zahl *number* so oft generiert, bis sie den Miller-Rabin-Test 8 Mal fehlerfrei durchläuft. Der Miller-Rabin-Test kurz MRT, testet, ob es sich bei der gegebene Zahl um eine Primzahl handelt. Tut dies jedoch nur mit einer gewissen Wahrscheinlichkeit. Daher muss der Test einige Male durchlaufen werden, um mit genügender Gewissheit von einer Primzahl sprechen zu können. Auf den Algorithmus werde ich hier jedoch nicht weiter eingehen.

Wir haben also jetzt p,q,n und m generiert, um einen vollständigen Schlüssel zu erhalten müssen wir lediglich noch die teilerfremde Zahl a generieren.

```
dict["a"] = self.generate_coprime(dict["m"])
```

```
    def generate_coprime(self, n):
        while True:
            a = random.randint(2, n - 1)
            if self.euclidean_algorithm_is_one(a, n) == True:
                return a

    def euclidean_algorithm_is_one(self, a, n):
        if a == 1 and n == 0:
            return True
        elif n == 0:
            return False
        return self.euclidean_algorithm_is_one(int(n), int(int(a) % int(n)))
```

Um die teilerfremde Zahl a zu m zu generieren. Wird geprüft, ob der euklidische Algorithmus 1 ergibt bzw. ob der größte gemeinsame Teiler 0 ist. Ist dies so, dann weiß man, dass die gegebenen Zahlen, teilerfremd sind. In der Programmierung habe ich dies mit einer rekursiven Funktion gemacht. Ein Problem, welches bei der rekursiven Programmierung in Python auftreten kann, ist, die Erreichung der maximalen Rekursionstiefe. Diese kann jedoch mit folgender Programmzeile erhöht werden:

```
sys.setrecursionlimit(n)
```

Mathematisch gesehen, kann man den euklidischen Algorithmus wie folgt darstellen:
$ggT(a,b)$
$a : b = x\ Rest\ c \rightarrow ggT(a,b) = ggT(b,c)$
Wird dies mit Zahlen weitergeführt, so erhält man den größten gemeinsamen Teiler der beiden Zahlen.

Nun bleiben nur noch zwei Programmteile übrig, die Ver- und Entschlüsselung.
Die Verschlüsselung wird durch folgende Funktion geregelt:

```
    def encrypt(self, __message_in_number_format_list, __public_key_n, __key_a):
        message = __message_in_number_format_list
        key_n = __public_key_n
        key_a = __key_a
        message_encrypted = []
        for i in message:
           message_encrypted.append(pow(int(i), key_a, key_n))
        return message_encrypted
```

Die Variable *message* ist eine Liste von Nummern, die einzeln verschlüsselt wird und als Liste zurückgegeben wird. Hier wird die Formel: $y = x\hat{}a\ mod\ n$, verwendet.

Die Entschlüsselung erledigt diese Funktion:

```
    def decrypt(self, __message_in_number_format_list, __private_key_n):
        message = __message_in_number_format_list
        key_n = __private_key_n
        key = self.load_private_key(key_n)
        message_decrypted = []
        b = self.extended_euclidean_algorithm([key["m"], 1, 0], [key["a"] , 0,
1])
        for i in message:
            if i == 0 or i == 1:
                message_decrypted.append(i)
            else:
                message_decrypted.append(pow(int(i), b, key["n"]))

        return message_decrypted
```

Hier wird ebenfalls wieder eine Liste zum entschlüsseln verwendet. Der korrekte private Schlüssel wird dadurch gefunden, dass der verschlüsselnde Teilschlüssel n, mitgegeben wird. Die JSON-Datei enthält eine Liste von Dictionaries, welche die Schlüssel enthalten. Mit der Zeit füllt sich diese Datei also mit mehr und mehr Schlüsseln. So kann jede Nachricht entschlüsselt werden, egal welcher der privaten Schlüssel verwendet wurde.
Um nun aber b herauszufinden, was Teil des privaten Schlüssels ist, kann einerseits die Formel: $b = a\hat{}(-1)\ mod\ m$ verwendet werden, was auch deutlich schneller wäre, andererseits kann dies manuell durch den erweiterten euklidischen Algorithmus erreicht

werden. Diesen habe ich ebenfalls, wie den euklidischen Algorithmus, mit einer rekursiven Funktion programmiert.

```
    def extended_euclidean_algorithm(self, array1, array2):
        if array2[0] == 1:
            return int(array2[2])
        values1 = np.array(array1)
        values2 = np.array(array2)
        factor = values1[0] // values2[0]
        values2 = np.subtract(values1, np.multiply(factor, values2))
        return self.extended_euclidean_algorithm(array2, values2)
```

Nun wird die b in die Formel: $x = y\hat{}b \ mod \ n$ unter der Bedingung eingesetzt, dass y nicht 1 oder 0 ist, da das Verfahren dann nicht funktioniert.
Schlussendlich wird dann wieder eine Liste zurückgegeben, die die entschlüsselten Werte enthält.

Wie schon erwähnt ist der Vorteil der asymmetrischen Verschlüsselung, dass kein Schlüssel ausgetauscht werden muss. Jedoch ist es dadurch auch so, dass die Verschlüsselung deutlich länger braucht. Hier kommen hybride Verschlüsselungsverfahren ins Spiel.

Was sind hybride Verschlüsselungsverfahren?

Anhand eines vereinfachten Beispiels lässt sich ein hybrides Verschlüsselungsverfahren folgendermaßen erklären:
Es wird Beispielsweise eine Nachricht mit der Caesarverschlüsselung verschlüsselt. Es wird also beispielsweise jeder Buchstabe einer Nachricht um 8 Stellen verschoben.
Nun hat man sowohl eine verschlüsselte Zeichenkette, als auch einen Schlüssel, der weitergegeben werden muss. Dieser Schlüssel wird dann mit Hilfe des RSA Verfahrens verschlüsselt.
Hat man beispielsweise den öffentlichen Schlüssel:
$n = 55 \ und$
$a = 7$
so wird dann die 8 verschlüsselt, in
$y = x\hat{}a \ mod \ n = 8^7 \ mod \ 55 = 2.$ Jetzt wird an den Empfänger sowohl die verschlüsselte Zeichenkette gesendet als auch, der verschlüsselte Schlüssel 2. Mithilfe des privaten Schlüssel kann nun der Schlüssel entschlüsselt werden und auf die Zeichenkette angewendet werden.
In der Praxis ist die Caesarverschlüesselung natürlich deutlich zu unsicher. Daher habe ich in meinem Projekt die AES Verschlüsselung verwendet. Diese Verschlüsselungstechnik ist ebenfalls eine symmetrische Verschlüsselung, die theoretisch ohne Schlüssel gelöst werden kann, aber praktisch weltweit keine Mittel zur Verfügung stehen dies zu tun.

Erfahrungen im Projekt:

Da ich das Projekt mit der Zeit immer größer gestaltet habe, also zunächst nur RSA, dann noch eine GUI und als letztes ein gesamtes Messenger-System, wurde es für mich mit der Zeit immer schwieriger den Überblick zu behalten. Ebenso ist RSA mittlerweile nur noch ein kleiner Teil des gesamten Projektes.

Neben einer SQL-Datenbank und einer GUI mit Flask (, was ich in Zukunft deutlich anders machen würde), habe ich natürlich auch noch die gesamte Kommunikation zwischen Frontend und Backend regeln müssen.
Ebenso ist klar mir klar geworden, dass eine selbst-programmierte Verschlüsselung nur in äußersten Ausnahmefällen sinnvoll ist, denn die eigene Implementierung von RSA ist deutlich langsamer und enthält möglicherweise Sicherheitslücken, da nicht immer die gleiche Rechenleistung gebraucht wird und in verschiedenen Teilen verschieden Schnell ist. Dadurch könnten Informationen über den Schlüssel herausgefunden werden.
Außerdem ist es möglich, dass die Schlüssel zu klein ist und dadurch geknackt werden könnte. Möglich wäre es die Schlüssellänge zu verändern, dann würde der Messenger jedoch deutlich längere Wartezeiten haben.
Hierfür wäre es interessant sich asynchrone Programmierung anzuschauen. Also, dass die Wartezeiten nur indirekt spürbar sind und das GUI geladen wird, während im Hintergrund die Verschlüsselung vollzogen wird.