Städtisches Gymnasium Rheinbach

Entwicklung eines RSA-Programms in Java und Erklärung der Funktionsweise von RSA

FACHARBEIT

Leistungskurs Informatik

Herr Faßbender

Jahrgangsstufe 11

Marius Zavelberg

Schuljahr 2017/2018

Inhaltsverzeichnis

1 Einleitung

Die Verschlüsselung von Daten nimmt heutzutage eine entscheidende Rolle ein, da die Welt immer digitaler wird und die Frage der Datensicherheit für die Menschen sehr wichtig ist. Unternehmen müssen die Daten ihrer Kunden und ihre eigenen vor Wirtschaftsspionage schützen. Wenn zum Beispiel Online-Dienste wie Facebook, Google und Instagram die Daten ihrer Nutzer nicht hinreichend schützen, riskieren diese das Vertrauen der Nutzer und damit ihren Unternehmenserfolg. Nicht nur in der Wirtschaft spielt die Verschlüsselung wichtiger Daten eine entscheidende Rolle, sondern auch in der Politik, wo in den vergangenen Jahren immer wieder Datenlecks aufgetaucht sind. Aber auch Kryptowährungen wie Bitcoin und Ethereum machen Gebrauch von der Idee der Public Key-Kryptografie, zu der auch das RSA-Verfahren gehört. Kryptowährungen gelten, dank dem Verfahren der Public Key-Kryptografie, als eine der sichersten und transparentesten Zukunftswährungen. In meiner Facharbeit behandele ich die Entwicklung eines RSA-Programms in Java und die Funktionsweise des RSA-Verfahrens. Aus aktuellem Anlass und der Möglichkeit, mein Wissen auf diesem Feld weiter aufzubauen, habe ich mich für dieses Thema entschieden. Beim RSA-Verfahren handelt es sich um ein auf mathematischen Grundlagen basierendes Verfahren. Ich habe mich für die Programmiersprache Java entschieden, da wir diese im Unterricht anwenden und ich somit mein vorhandenes Vorwissen beim Programmieren einbringen kann. Im Folgenden werde ich auf die mathematischen Grundlagen und die Implementation in Java eingehen. Zusätzlich werde ich eine GUI vorstellen, die das Verfahren bedienfreundlich und vereinfacht für den Nutzer darstellt. Dabei werde ich auch auf die Anwendung des RSA-Verfahrens, sowie die Geschichte und den Vergleich zu anderen gleichwertigen Verschlüsselungsalternativen eingehen.

2 Die Funktionsweise des RSA-Verfahrens

Das RSA-Verfahren ist ein Public Key-Kryptosystem und gehört damit zu den asymmetrischen Verschlüsselungsverfahren, auf die ich in Kapitel 5 eingehen werde. Beim RSA-Verfahren liegen zwei Schlüssel vor. Bei einem Public Key-Kryptosystem kann die Kommunikation auch ohne die Kenntnis über einen gemeinsamen, geheimen Schlüssel stattfinden.[1] Die Kommunikation findet mit einem Public Key (öffentlicher Schlüssel) und einem Private Key (privater Schlüssel) statt. Informationen werden mit dem Public Key verschlüsselt. Da der Public Key bekannt ist, kann jeder die

[1] http://deacademic.com/dic.nsf/dewiki/107671

Informationen, die er an den Inhaber des Schlüssels senden will, damit verschlüsseln. Diese Informationen können nur vom Inhaber des zugehörigen Private Keys entschlüsselt werden. Da nur Zahlen und keine Klartexte verschlüsselt werden können, wird der Klartext mithilfe des ASCII-Codes[2] in eine Zahlenfolge konvertiert. Der ASCII-Code kann mithilfe von Hexadezimalzahlen oder Dezimalzahlen angegeben werden. Diese Zahlenfolgen werden dann in Blöcke einer bestimmten Länge unterteilt (siehe Kapitel 2.2). Für den PC wird dieser in Binärcode umgewandelt und blockweise verschlüsselt. Da der Kryptotext auch in ASCII vorliegt, wird dieser wieder in Blöcke unterteilt, die wiederum blockweise entschlüsselt werden. Die Ver- und Entschlüsselung sind schwer umkehrbare, auf Algorithmen basierende Verfahren.[3] Sie beruhen darauf, dass es nur schwer möglich ist, große Zahlen zu faktorisieren und aus dem Produkt zweier Primzahlen diese Primzahlen zu ermitteln.

2.1 Die Generierung des Public Keys und Private Keys

Die Generierung des Public Keys beruht auf den Zahlen e und n.[4] Die Zahl e kann vorher festgelegt werden. In den Anfängen des RSA-Verfahrens wurde aus Effizienzgründen empfohlen, die Primzahl 3 zu wählen, die nach heutigen Standards jedoch als unsicher gilt. Deshalb wird heutzutage die Zahl $2^{16} + 1$ verwendet, da diese eine gute Alternative darstellt.[5] Die Zahl n ist das Produkt zweier Primzahlen und dementsprechend schwer faktorisierbar, wenn diese Primzahlen entsprechend groß gewählt werden.[6] Die Generierung des Private Keys beruht auf den Zahlen p, q, $\varphi(n)$, d. Die Zahlen p und q sind zwei groß gewählte Primzahlen, aus denen sich die Eulersche Phi-Funktion (die Eulersche Zahl $\varphi(n)$) berechnet. Mit ihrer Hilfe und dem multiplikativ Inversen der Zahl e lässt sich die Zahl d bestimmen. Sollten Dritte aus der Zahl n die beiden Primzahlen p und q bestimmen, können diese damit den Private Key generieren, welche den Kern des Verfahrens bilden. Deshalb müssen die beiden Primzahlen p und q möglichst groß gewählt werden, damit das Produkt n entsprechend groß ausfällt, nur schwer faktorisiert werden kann und sich wieder in die Primfaktoren p und q zerlegen lässt.[7] Auf die Bestimmung der Komponenten vom Public Key und Private Key wird in Kapitel 3 näher eingegangen.

---

[2] https://www.xovi.de/wiki/ASCII-Code
[3] https://www.staff.uni-mainz.de/pommeren/DSVorlesung/KryptoBasis/RSA.html
[4] Vgl. Freiermuth u.a., Einführung in die Kryptologie, Zürich 2010, S. 282, S. 326-328
[5] http://www.inf.fh-flensburg.de/lang/krypto/protokolle/rsa.htm
[6] Ebd.
[7] Vgl. Freiermuth u.a., a.a.O., S. 282, S. 326-328

2.2 Die Wahl der Blocklänge

Die Wahl der Blocklänge bei modernen Kryptosystemen beruht darauf, dass diese das Alphabet, bestehend aus 0 und 1, verwenden. Es ist deshalb erforderlich, einen Klartext oder Kryptotext zu konvertieren, wenn diese als Buchstaben, Zahlen oder Zeichen vorliegen. Der Klartext wird in Blöcke verschiedener Länge eingeteilt, die nötigenfalls aufgefüllt werden, wenn der angegebene Text nicht der Blocklänge entspricht. Die Bits, mit denen dieser Rest angegeben wird, entsprechen keinem Zeichen, Buchstaben und keiner Zahl, sondern sind nur „Lückenfüller“, die den Wert *null* haben. Diese Blöcke werden nacheinander und unabhängig voneinander verschlüsselt. Die Blocklänge wird durch Bits angegeben. Buchstaben, Zeichen und Zahlen werden durch Bitfolgen ersetzt. Dieser Vorgang ist die Vorbereitung für die Verschlüsselung.[8] Die Länge der Blöcke des RSA-Verfahrens wird durch $[\log_2(n\text{-}1)] + 1$ bestimmt, wobei *n* in Bits angegeben ist. Die maximale Anzahl der Symbole, die mithilfe eines Blocks verschlüsselt werden können, hängt also von der Wahl der Primzahlen *p* und *q* ab, welche die zwei Primfaktoren zur Berechnung des Produkts *n* sind. Ein Block ist die binäre Codierung einer Zahl $w \in \{0, 1, 2, \ldots, n-1\}$. Wenn w größer als *n* ist, dann kann der Klartext nicht verschlüsselt werden. Deshalb wird der ASCII codierte Klartext in Blöcke unterteilt, da dadurch w immer kleiner als *n* ist.[9]

3 Die Mathematik hinter dem RSA-Verfahren

In diesem Teil wird auf die mathematischen Grundlagen des Verfahrens eingegangen, mit deren Hilfe der Private Key und Public Key erzeugt werden, sowie die Ver- und Entschlüsselung zustande kommt. Im Anhang befinden sich die Verfahren, die zur Bestimmung der Zahlen verwendet werden. Im weiteren Verlauf wird bei Gleichungen das Wort modulo durch mod abgekürzt.

3.1 Komponenten zur Erzeugung des Public Key und Private Key

Hier wird auf die einzelnen Komponenten eingegangen, mit denen Public Key und Private Key erstellt werden (vgl. Kapitel 2.1). Begonnen wird mit den Zahlen *p* und *q*. Danach werden die Zahlen *n* und $\varphi(n)$ vorgestellt. Abschließend wird auf die Zahlen *e* und *d* eingegangen. Die Zahlen *p* und *q* sind zwei zufällig ausgewählte große Primzahlen, die multipliziert das Produkt *n* ergeben. Mit ihnen lässt sich die Eulersche Zahl berechnen. Je größer *p* und *q* gewählt werden, desto größer ist am Ende die Zahl *n* und die Zahl $\varphi(n)$.

[8] Vgl. Freiermuth u.a., a.a.O., S.186-190, S. 327
[9] Ebd.

Je größer *p* und q ausfallen, desto schwerer wird es, diese durch die Faktorisierung von *n* zu ermitteln. Somit sind die Zahlen *p* und *q* ein wichtiger Bestandteil zur Erstellung des Private Keys. *p* und *q* bilden den Anfang, wenn man den Public Key/Private Key generieren will.[10] Die Zahl *n* ist das Produkt der beiden Primzahlen *p* und *q*. Durch sie wird die Länge des Schlüssels in Bit festgelegt, mit der ein Klartext verschlüsselt werden kann. Die Länge des Klartextes muss dabei kleiner sein als die Länge des Schlüssels, die durch *n* angegeben wird.[11] Um den Klartext verschlüsseln zu können, muss dieser kleiner als die Zahl *n* ausfallen. Von der Zahl *n* ist damit die Blocklänge (vgl. 2.2) abhängig. Aus *n* wird der Public Key generiert, wodurch es möglich wird, *n* auszulesen. Da *n* einer sehr großen Zahl entspricht, ist es nur schwer möglich, die Zahlen *p* und *q* zu errechnen. Falls es doch gelingt, kann der Private Key reproduziert werden. Durch die Eulersche Zahl $\varphi(n)$ (auch als $\varphi(p*q) = (p–1) * (q–1)$ darstellbar) lässt sich die Anzahl der Elemente in $\mathbb{Z}_n$ berechnen, die als gemeinsamen Teiler nur die Zahl 1 haben, dementsprechend teilerfremd zu den beiden Primzahlen *p* und *q* sind.[12] Die Berechnung lautet somit $\varphi(n) = (p – 1) * (q – 1)$. Die Darstellungsweise $\varphi(p*q) = (p – 1) * (q – 1)$ kann auch verwendet werden, da die Zahl *n* das Produkt der beiden Primzahlen *p* und *q* darstellt. Die Zahl *e* multipliziert mit der Zahl *d* modulo $\varphi(n)$ ergibt 1.[13] Das bedeutet im Umkehrschluss, dass man zusammen mit *e* und $\varphi(n)$ die Zahl *d* berechnen kann, die ein essentieller Bestandteil zur Generierung des Private Keys ist. Die Zahl *e* ist der Exponent des Klartextes *w*, welcher modulo der Zahl *n* den Kryptotext *c* liefert (siehe Kapitel 3.2). Damit ist sie ein essenzieller Bestandteil für die Generierung des Public Keys. Zu Anfang wurde aus Gründen der Effizienz die Zahl 3 gewählt, dies galt allerdings als unsicher. Deshalb wird als Exponent eine möglichst große Zahl gewählt, zum Beispiel die Zahl $2^{16}+1$, wodurch die Verschlüsselung immer noch schnell abläuft.[14] Die Zahl *d* ist der Exponent des Kryptotextes *c*, welcher modulo der Zahl *n* den Klartext *w* liefert (siehe Kapitel 3.2). Damit ist sie ein essenzieller Bestandteil für die Generierung des Private Keys. Das Inverse der Zahl *d* (d^{-1}) lässt sich aus *e* mod $\varphi(n)$ bestimmen. Dies bedeutet, dass man *d* aus dem Inversen der Zahl e^{-1} mod $\varphi(n)$ bestimmen kann. Dieses Element kann mithilfe des erweiterten euklidischen Algorithmus angeben werden, sodass man in der Lage ist, den Wert für *d* zu bestimmen.[15]

---

[10] Vgl. Freiermuth u.a., a.a.O., S. 282, S. 326-328
[11] https://www.heise.de/security/artikel/Weitere-Gefahren-271156.html
[12] Vgl. Freiermuth u.a., a.a.O., S. 282, S. 326-328
[13] Ebd.
[14] http://www.inf.fh-flensburg.de/lang/krypto/protokolle/rsa.htm
[15] Vgl. Freiermuth u.a., a.a.O., S.130-131, S. 279

3.2 Die Anwendung des RSA-Verfahrens

Das RSA-Verfahren benötigt die Zahlen *e* und *n* zur Erstellung des Public Keys. Die Zahlen *p, q, φ(n)* (oder *φ(p*q)*) und *d* werden benötigt, um den Private Key zu erstellen. „Der Kryptotext *c* wird berechnet durch: $c = w^e \bmod n$“.[16] „Für einen Kryptotextblock *c* berechnen wir den entsprechenden Klartextblock *w* mittels: $w = c^d \bmod n$“.[17]

4 Die Implementation in Java

In diesem Abschnitt wird auf die Implementation des Verfahrens in Java eingegangen. Das Programm wurde mithilfe von BlueJ, einem Compiler für Java, erstellt. Das von mir erstellte Programm wurde unter Verwendung der importierten Klassen *BigInteger* und *SecureRandom* implementiert. Die Erläuterungen zu den beiden verwendeten Klassen *BigInteger* und *SecureRandom* befinden sich im Anhang, sowie eine Erläuterung zu den von BigInteger verwendeten Methoden. Die zusätzliche Erstellung einer GUI wird im Kapitel 7.6 behandelt. Das Projekt soll am Ende auch in einer .jar-File vorliegen. Die in Kapitel 2 und Kapitel 3 erwähnte Blocklänge wird für jeden Text auf eine Länge von 4 Bits festgelegt. Die Variablennamen werden in den Kapiteln 4.1 bis 4.3 *kursiv* geschrieben. Mit den vor mir erstellten Konstruktoren und Methoden wird ebenfalls *kursiv* geschrieben. Verwendete Konstruktoren und Methoden der beiden importierten Klassen werden auch *kursiv* geschrieben.

4.1 Die Festlegung der Variablen

Zu Beginn werden die Variablen *p, q, n, e, phi (φ(n)), e, d, newe, newn* und *eukl* (erweiterter euklidischer Algorithmus, auf diesen wird im Anhang eingegangen) als BigInteger deklariert, jedoch werden ihnen noch keine Werte zugewiesen. Die Bitlänge von *p* und *q* wird als Integer deklariert und heißt *bitLength*. Ihr wird auch noch kein Wert zugewiesen. Die Blocklänge wird in den nachfolgenden Methoden als Integer auf 4 festgelegt und als Standard verwendet. Die Strings *encrypt* und *decrypt* werden als leere Strings erzeugt und dienen später zur Speicherung des verschlüsselten Klartextes und des entschlüsselten Kryptotextes. Die Variablen *start* und *ende* sind als double deklariert und werden dazu verwendet, die benötigte Zeit zur Erzeugung der beiden Schlüssel anzugeben.

---

[16] Freiermuth u.a., a.a.O., S. 186-190, S. 327
[17] Ebd.

4.2 Konstruktoren zur Generierung der Schlüssel

Beim Aufruf des ersten Konstruktors *public RSA(BigInteger newe, BigInteger newn)* lassen sich bei einem schon bestehenden Public Key die Werte für dessen Komponenten übergeben. Mithilfe des Konstruktors ist es möglich, einen schon vorhandenen Public Key zu verwenden. *newe* entspricht *e* und *newn* entspricht *n*. Mit dem zweiten Konstruktor *public RSA(int bits)* lassen sich der Public Key und der Private Key generieren. Am Anfang der Methode wird die Systemzeit zur Ausführung des Konstruktors festgehalten und speichert diese in der schon vorhandenen Variable *start*. Danach erfolgt die Generierung der Schlüsselpaare. Dies geschieht in einer *while*-Schleife, die solange den in ihr angegebenen Vorgang durchführt, wie *d* gleich *null* ist. Der Vorgang in der *while*-Schleife beginnt mit der Erzeugung des Objekts *r* der Klasse SecureRandom. Der dem Konstruktor übergebene Integer *bits* wird der bis dahin leeren Variable *bitLength* als Wert zugeschrieben. Danach erfolgt die Bestimmung der beiden Primzahlen *p* und *q*, die als BigInteger deklariert sind. Der Aufruf des Konstruktors *new BigInteger*(int *bitLength*, int *certainty*, Random *rnd*)) der Klasse BigInteger geniert eine Zufallszahl. Dieser Konstruktor wird für die Bestimmung der beiden Primzahlen *p* und *q* verwendet. Er schafft dies mithilfe der Variable *bitLength*, die bereits festgelegt wurde. *bitLength* entspricht dem ersten Integer des oben im Text angeführten Konstruktors. Desweitern muss ihm ein weiterer Integer (int *certainty*) und die Zufallszahl *rnd* zugewiesen werden. Die Zufallszahl *rnd* entspricht dem Objekt *r*, die zuvor als Objekt der Klasse SecureRandom erzeugt wird. *certainty* gibt die Sicherheit an, dass es sich bei der generierten Zufallszahl wirklich um eine Primzahl handelt. Je näher der Wert von *certainty* an 0 ist, desto sicherer ist es, dass es sich bei der erzeugten Zahl um eine Primzahl handelt. Der Wert, der später durch *certainty* angegeben wird, entspricht dem Wert $1-1/2^{certainty}$. Der Wert von *certainty* wurde hier auf 500 festgelegt. Die Bitlänge der erzeugten Zufallszahl entspricht am Ende dem Wert, der dem Konstruktor am Anfang zugewiesen wurde (*bits*). Es werden die Primzahlen *p* und *q* der eingegebenen Bitlänge (*bitLength*; dem Parameter *bits* des Konstruktors *public RSA(int bits)*) entsprechend erzeugt. Daraufhin erfolgt die Berechnung der Zahl *n*, welche sich durch Einsetzen der Methode *multiply()* der Klasse BigInteger berechnen lässt. Mithilfe der Methode *multiply()* werden die beiden Zahlen *p* und *q* multipliziert. Das Ergebnis dieser Multiplikation ist der Wert für *n*. Der Wert der Variable *phi* wird durch Einsetzen der Methoden *subtract()* und *multiply()* der Klasse BigInteger berechnet. Die Werte von *p* und *q* werden jeweils mit *subtract()* um den Wert 1 subtrahiert. Anschließend werden die

veränderten Werte von *p* und *q*, also den Werten *p*-1 und *q*-1, mit *multiply()* multipliziert. Das Ergebnis ist der Wert von *phi*. Der Wert für die Variable *e*, welcher noch kein Wert zugeschrieben wurde, wird mittels der Methoden *pow()* und *add()* der Klasse BigInteger bestimmt. Sie wird aus keinem anderen Komponenten (aus keiner der bisher bestimmten Zahlen) bestimmt, sondern wird hier festgelegt. Der Wert von *e* ist hier $2^{128}+1$ und eine Primzahl. Zuerst wird die Zahl 2 mit der Zahl 128 mittels *pow()* potenziert. Anschließend wird mit *add()* die Zahl 1 zum Ergebnis dazu addiert. Der BigInteger *gcd* ist der ggT (größter gemeinsamer Teiler) der Werte von *phi* und *e* und lässt sich durch die Methode *gcd()* der Klasse BigInteger bestimmen. Die darauffolgende *if*-Anweisung läuft unter der Bedingung, dass *gcd* gleich 1 ist. In der *if*-Anweisung erfolgt die Berechnung des Wertes für die Variable *d*, welche bis dahin noch keinen Wert zugeschrieben wurde. Indem die Methode modInverse() der Klasse BigInteger eingesetzt wird, lässt sich der Wert für *d*, aus *e* und *phi* bestimmen (*e*.modInverse(*phi*)). Mithilfe der Methode lässt sich der inverse Wert der Variable *e* verwenden. Gleichzeitig wird der Wert von *e* modulo dem Wert von *phi* gerechnet. Die Bestimmung läuft unter der Bedingung, dass *gcd* nicht größer als 1 ist. Wurde *d* ermittelt, bricht die *while*-Schleife ab. Dies wird durch die Aufforderung *break* am Ende der *if*-Anweisung mitgeteilt, da ein Wert für *d* gefunden wurde. Falls *break* weglassen wird, wiederholt die *while*-Schleife den in ihr angegebenen Vorgang immer weiter (auch wenn *d* längstens bestimmt wurde). Der Wert für die Variable *eF*, welche ein BigInteger ist, wird aus den zuvor ermittelten Werten der Variablen *d* und *phi* durch die Anwendung der Methode *modInverse()* der Klasse BigInteger berechnet (*d*.modInverse(*phi*)). Die Schritte, die durch *modInverse()* gemacht werden, sind weiter oben im Text erklärt worden. Die Variablen *eF*, *d* und *phi* werden zur Ermittlung des Wertes der Variable *eukl* verwendet. Dies dient nur zum Beweis, dass die Erzeugung der Schlüssel erfolgreich war. Wenn *eukl* den Wert 1 hat, war die Schlüsselerzeugung erfolgreich. Zur Bestimmung von *eukl* werden zunächst die Variablen *d* und *e* durch Benutzung der Methode *multiply()* der Klasse BigInteger multipliziert. Anschließend wird die Methode *mod()* der Klasse BigInteger benutzt, um *eukl* aus dem Produkt von *d* und *e*, modulo der Zahl *phi* zu berechnen. Der Wert von *eukl* sollte am Ende 1 sein. Wenn dies der Fall ist, wird auf der Konsole „true“ angezeigt. Vor der Ausgabe auf der Konsole wird die verstrichene Zeit durch die Variable *ende* gespeichert. Der Wert ergibt sich aus der aktuellen Systemzeit, abgezogen der von *start* gespeicherten Systemzeit und wird in Sekunden angegeben. Zum Schluss wird die verstrichene Zeit zusammen mit den Komponenten des Public Keys und des Private Keys auf der Konsole ausgegeben.

4.3 Die Methoden zur Ver- und Entschlüsselung

Da sich die Methoden *public String verschluesselung(String s)* und *public String entschluesselung(String s)* vom Aufbau her gleichen, wird der Aufbau beider Methoden in Einem beschrieben. Es liegen außerdem die Methoden *public String verschluesselungEinfach(String s)* und *public String entschluesselungEinfach(String s)* vor. Der übergebene String in diesen Methoden wird nicht in Blöcke unterteilt. Die Methoden bieten sich deshalb am besten zur Ver- und Entschlüsselung von einzelnen Wörtern, kleinerer oder einzelner Sätze an. Die Methoden *public String verschluesselung(String s)* und *public String entschluesselung(String s)* erweitern die Methoden *verschluesselungEinfach* / *entschluesselungEinfach* nur um das Unterteilen des übergebenen Strings in die vorgegebene Blocklänge und ähneln sich vom Verfahren zur Unterteilung der Blocklänge. Dazu wird der übergebene String in einen String-Array umgeformt und mittels einer *for*-Schleife durchlaufen. Danach werden die Elemente des Arrays zu einem BigInteger umgeformt und der Block verschlüsselt/entschlüsselt. Der verschlüsselte BigInteger wird nun wieder in einen String umgewandelt. Die Verschlüsselung der Blöcke/der BigInteger wird mithilfe der Methode *modPow()* der Klasse BigInteger vollzogen. Der Methode *modPow()* wird als erstes entweder der Wert von *e* oder *d* übergeben. Als zweiten Wert wird ihr der Wert von *n* übergeben. Alle Werte wurden zuvor im Konstruktor bestimmt.

5 Sonstiges zum RSA-Verfahren

5.1 Die Geschichte des RSA-Verfahrens

Das RSA-Verfahren wurde 1977 von den Mathematikern Ronald L. Rivest, Adi Shamir und Leonard Adleman am Massachusetts Institute of Technology (MIT) entwickelt. Es basiert auf der Idee der Wissenschaftler und Kryptologen Diffie, Hellman und Merkle, die das Konzept der Public Key-Kryptografie ein Jahr zuvor vorgestellt hatten. Der Name des Verfahrens setzt sich aus den ersten Buchstaben der Nachnamen seiner Entwickler zusammen. Das RSA-Verfahren entspricht allen Anforderungen eines asymmetrischen Kryptosystems und ermöglicht neben der Ver- und Entschlüsselung von Texten auch das Erstellen von digitalen Signaturen.[18]

5.2 Asymmetrische Kryptosysteme

Bei einem asymmetrischen Kryptoverfahren haben die teilhabenden Parteien jeweils ein Schlüsselpaar, basierend auf einem Private-Key (privater Schlüssel) und einem Public-

[18] http://deacademic.com/dic.nsf/dewiki/107671

Key (öffentlicher Schlüssel). Mithilfe des Public Keys kann jeder, der mit dem Inhaber dieses Keys kommunizieren will, Daten verschlüsseln, auf die nur der Inhaber mit dem zugehörigen Private Key Zugriff hat. Somit ist es einem Außenstehenden nicht möglich, diese Daten auszulesen, wenn er nicht die nötigen Informationen zur Erstellung des benötigten Private Keys besitzt. Mit dem öffentlichen Schlüssel ist es zudem für andere Parteien möglich, den Inhaber des Keys zu identifizieren. Mithilfe des Private Keys ist es dem Inhaber möglich, Daten, die mit dem zugehörigen Public Key verschlüsselt wurden, zu entschlüsseln und sich als Inhaber des Public Keys zu authentisieren.[19] Dies lässt sich an einem Beispiel vereinfacht darstellen: Es wird angenommen, dass eine Schule für jeden Schüler einen Spint zur Verfügung stellt. An diesem Spint befinden sich ein Schild mit dem Namen des Schülers und eine Klappe, die es für andere Schüler möglich macht, dem Schüler, der diesem Spint zugeordnet wird, Nachrichten zu schicken. Der Spint ist mit einem Zahlenschloss gesichert. Der Code, mit dem sich das Schloss entsperren lässt und n-Ziffern lang ist, kennt nur der Schüler, dem der Spint gehört (sofern er diesen nicht an Dritte weitergegeben hat). Der Schüler, dem der Spint gehört, kann sich so als Inhaber identifizieren und auf Nachrichten, die durch die Klappe in seinen Spint geworfen wurden, zugreifen.

5.3 Anwendungsbereiche

Da es sich beim RSA-Verfahren um ein asymmetrisches Kryptoverfahren handelt, sind die Anwendungsbereiche vielfältig. Es kann zur Verschlüsselung von E-Mails (OpenPGP, S/MIME), für digitale Signaturen (vgl. Kapitel 2.1) und kryptografische Protokolle wie SSH, SSL/TLS und https verwendet werden.[20] Die einzelnen Protokolle werden im Glossar erklärt.

5.4 Gleichwertige Alternativen

Beim RSA-Verfahren handelt es sich um ein auf Algorithmen basierendes Verfahren[21], das wiederum durch die Anwendung von Algorithmen geknackt werden kann, wie sich in der Vergangenheit gezeigt hat.[22] Daher werden Verfahren, die auf der Elliptischen-Kurven-Kryptografie (Elliptical Curve Cryptography, kurz: ECC) oder dem Faktorisierungsproblem[23] beruhen, bevorzugt, da diese effizienter sind.[24] Anbieter wie

[19] Ebd.
[20] https://www.philipphauer.de/info/info/asymmetrische-verschluesselung
[21] https://www.staff.uni-mainz.de/pommeren/DSVorlesung/KryptoBasis/RSA.html
[22] https://www.tecchannel.de/a/digitale-signaturen-werden-10-000-mal-schneller,2023433
[23] Ebd.
[24] https://www.elektronik-kompendium.de/sites/net/1910151.htm

Mozilla und Microsoft unterstützen mittlerweile ECC.[25] Eine aktuelle Alternative zum RSA-Verfahren ist die MQQ-Signatur (Multivariate Quadratic Quasigroups), das an der Norwegian University of Science and Technology (NTNU) entwickelt wurde.[26] Dieses Verschlüsselungsverfahren bietet den Vorteil, dass das Erstellen von Signaturen 10.000-mal schneller als mit der RSA- oder ECC-Methode erfolgt. Der Verifizierungsprozess soll 17.000-mal schneller ablaufen als bei RSA und ECC. Das Verfahren macht die parallele Verarbeitung mit Mehrkern-Prozessoren möglich. Das Signaturschema ist nicht durch Algorithmen zu umgehen. Das liegt unter anderem daran, dass das Verfahren, wie auch das RSA-Verfahren, mit der Falltürfunktion (auch: Einwegfunktion) arbeitet. Um das MQQ-Verfahren zu entschlüsseln, kann höchstens die Brute-Force-Methode angewendet werden.[27]

6 Fazit und Ausblick

Mit dem von mir implementierten RSA-Verfahren ist es möglich, Klartexte und Kryptotexte in eine Blocklänge von 4 Bits zu unterteilen. Diese können nacheinander ver- und entschlüsselt werden. Es bietet außerdem die Möglichkeit, mit einem schon vorhandenen Public Key, einen Klartext zu verschlüsseln. Mithilfe der GUI wird das RSA-Verfahren auf einer grafischen Bedienoberfläche dargestellt, die es dem Nutzer leichter macht, die Funktionen des von mir entwickelten Verschlüsselungsprogramms zu verwenden. In Zukunft werde ich das Verfahren auch in andere Programmiersprachen wie C++ und Python umsetzen. Die Umsetzung in diese beiden Programmiersprachen ist mir sehr wichtig, da C++ der Funktionsweise von Java ähnelt und Python immer häufiger verwendet wird.

[25] https://www.searchsecurity.de/definition/Elliptische-Kurven-Kryptografie-Elliptic-Curve-Cryptography-ECC
[26] https://www.researchgate.net/publication/262294724_MQQ-SIG_an_ultrafast_and_provably_CMA_resistant_digital_signature_scheme
[27] https://www.tecchannel.de/a/digitale-signaturen-werden-10-000-mal-schneller,2023433

7 Anhang

7.1 Erklärung mathematischer Verfahren

Die **Funktion von Euler** ($\varphi(n)$) ordnet jeder natürlichen Zahl n die Anzahl natürlicher Zahlen $a \in \{1, \ldots, n\}$ zu, die zu ihr teilerfremd sind. Falls es sich bei n um eine Primzahl p handelt, ist p zu den Elementen von $a \in \{1, \ldots, p\text{-}1\}$ teilerfremd (ggT(p, $a \in \{1, \ldots, p\text{-}1\}$) = 1, daher $\varphi(n) = p$ -1. Die φ-Funktion kann multiplikativ für zwei Primzahlen angewandt werden $\varphi(pq) = \varphi(p)\varphi(q) = (p\text{-}1)*(q\text{-}1)$. Das Produkt dieser Zahlen gibt die Anzahl der Elemente an, die zu p und q teilerfremd sind.

Durch den **euklidischen Algorithmus** kann man den ggT zweier natürlicher Zahlen (a und b) bestimmen. Der Teiler muss dabei die Bedingung erfüllen, dass er beide Zahlen teilt und ein Element der natürlichen Zahlen $\mathbb{N}$ ist. Aus der Schnittmenge (Elemente aus den Mengen von a und b, die Teiler von a und b sind) der beiden Zahlen a und b wird das größte Element genommen, der ggT der Zahlen a und b. Das Verfahren zur Berechnung des ggT läuft unter der Voraussetzung, dass a und b positiv sind und a größer b. Es wird solange durchgeführt, wie a ungleich 0 gilt. Die beiden folgenden Schritte beinhalten, dass a durch a mod b ersetzt wird und die Zahlen a und b vertauscht. Mit letzterem ist gemeint, dass die Zahl b aus dem vorhergehenden Schritt an die Stelle von a im nächsten eingesetzt wird. Anstelle von b wird der Rest (das Ergebnis) des vorherigen Schritts eingefügt.[28]

Beispiel:

1. a_1 mod $b_1 = q_1$ Rest r_1

2. $((a_2=b_1)$ mod $(b_2= r_1)) = q_2$ Rest r_2

3. $((a_3=b_2)$ mod $(b_3= r_2)) = q_3$ Rest r_3

n. $((a_n=b_{n-1})$ mod $(b_n= r_{n-1})) = q_n$ Rest r_n

Der **erweiterte euklidische Algorithmus** baut auf dem euklidischen Algorithmus auf.[29] Der wichtigste Unterschied zum euklidischen Algorithmus ist jedoch, dass die Zahlen a und b durch die ganzzahlige Linearkombination $\alpha*a+\beta*b$ dargestellt werden und sich der ggT als Linearkombination von a und b darstellen lässt. Dadurch ist es möglich, den

[28] Vgl. Freiermuth u.a., a.a.O., S. 130-131, S. 279
[29] Ebd.

ggT(a, b) zu bestimmen, mit $a, b \in \mathbb{N}$ (a, b sind Elemente aller natürlichen/positiven Zahlen) und $\alpha, \beta \in \mathbb{Z}$ (α, β sind Elemente aller ganzen Zahlen). Das Verfahren entspricht dem des euklidischen Algorithmus. Die Zahlen a und b ersetzt man durch ihre Linearkombination. Zu Beginn wird a durch $a=1*a+0*b$ und b durch $b=0*a+1*b$ beschrieben. Das Ergebnis, welches zur weiteren Berechnung im nächsten Schritt benötigt wird (der Rest), wird durch die Linearkombination der Zahlen a und b im nächsten Schritt beschrieben. Bei einem neuen Schritt verändern sich nur die Skalare (α und β), a und b bleiben bis zum letzten Schritt an ihrer Stelle stehen und entsprechen ihren Anfangswerten ($a=1*a+0*b$ und $b=0*a+1*b$). Das Ergebnis wird nach jedem Schritt (der Rest) auch als Linearkombination angegeben (die Linearkombination enthält die beiden Zahlen a und b) und wird wie beim euklidischen Algorithmus im nächsten Schritt mitverwendet. Mithilfe des erweiterten euklidischen Algorithmus lässt sich das multiplikativ Inverse effizient berechnen, das zur Berechnung von d benötigt wird.[30]

Mit der **Primitivwurzel** lässt sich die kleinste Zahl $k > 0$ mit $a^k \equiv 1 \bmod m$ als Ordnung darstellen. $a^k \equiv 1 \bmod m$ bezeichnet man als die Ordnung von a modulo m. $k = \text{ord}_m(a)$. Ist der ggT(a, m) = 1, dann gilt für alle $t \in \mathbb{Z}$, $\text{ord}_m(a) = \text{ord}_m(a + t*m)$. Wenn m ein Element der natürlichen Zahlen $\mathbb{N}$ und a ein Element der elementaren Zahlenmenge $\mathbb{Z}$ ist, sowie die Elemente a und m den größten gemeinsamen Teiler 1 haben, dann gilt auch $k = \text{ord}_m(a)$. Daraus folgt, dass k nicht teilerfremd zu $\varphi(m)$ ist.[31] Die Primitivwurzel ist eine auf Euler zurückzuführende Definition. Wenn $m > 0$ und a ganze teilerfremde Zahlen sind, wenn $\text{ord}_m(a) = \varphi(m)$, dann ist a eine Primitivwurzel modulo m. Die Zahlen $\varphi(m)$ sind alle teilerfremd zu m. „Ist p eine Primzahl und kennt man eine Primitivwurzel t_0 modulo p, so sind die $\varphi(p-1)$ Potenzen t_0^i mit zu p-1 teilerfremden $i \in \{1, \ldots, p\text{-}1\}$ genau die sämtlichen Primitivwurzeln modulo p.“[32] Für t_0 bedeutet das, dass es sich um eine Primitivwurzel modulo p handelt, wenn das Ergebnis aus $t_0^i \bmod p$ immer ein Ergebnis darstellt, welches ein Element der Zahlenmenge $\{1,\ldots, p\text{-}1\}$ darstellt. Dies wiederum bedeutet, dass nur Primzahlen t_0 entsprechen können, wenn $t_0^i \bmod p$ gilt und es sich bei p um eine Primzahl handelt. Ist t_0 eine Primitivwurzel $t_0^i \bmod p$ (p ist eine Primzahl), dann ist t_0 auch eine Primitivwurzel $t_0^i \bmod q$ (q ist eine Primzahl). Wenn man nun p und q multiplikativ mit $\varphi(pq) = (p\text{-}1)*(q\text{-}1)$ darstellt, dann steht dieses Ergebnis für die Anzahl

[30] http://www.inf.fh-flensburg.de/lang/krypto/algo/euklid.htm
[31] https://www.frustfrei-lernen.de/mathematik/kongruenz-zahlentheorie.html
[32] Bundschuh, Einführung in die Zahlentheorie, Heidelberg^4 1998, S. 109-118

der Elemente, die teilerfremd zu *p*-1 und *q*-1 sind. Daraus lässt sich schließen, dass die Anzahl $x \in \mathbb{Z}_p \times \mathbb{Z}_q$, gemeinsam mit *p*q*, den ggT(*pq*, *x*) = 1 besitzen.[33]

7.2 Erklärung der Klassen BigInteger/SecureRandom

Alle arithmetischen Operationen, die zur Generierung der Bestandteile des Public Keys und Private Keys benötigt werden, finden unter Verwendung der **Klasse BigInteger** statt. Die Primzahlen *p* und *q* werden mithilfe der Klasse SecureRandom erzeugt. BigInteger-Zahlen sind unveränderbare, zufällig ganze Präzisionszahlen („Immutable, arbitary-precision Intergers"), die sich wie Integer verhalten. Mit ihnen sind alle Operationen möglich, die auch mit normalen Integern möglich sind. BigInteger erweitern Integer um Operationen der modularen Arithmetik, den ggT, das Arbeiten mit Primzahlen und die Bit-Manipulation (Handhabung mit Bits). Diese fallen so groß wie möglich aus (können größere Zahlen als Integer darstellen). Dadurch können Ergebnisse von Operationen so genau wie möglich ausfallen (sie können zum Beispiel so groß wie der Datentyp double werden, aber keine Fließkommazahlen darstellen, da sie sich wie Integer verhalten). Modulare arithmetische Operationen können exponenzieren, das multiplikativ Inverse darstellen und geben immer ein nicht negatives Ergebnis zurück. Bit Operationen arbeiten immer mit einem einzelnen Bit (single-Bit) ihres Operanden. Nötigenfalls wird der Operand erweitert (die Anzahl der Bits erhöht; sign-extension), ohne dass sich sein Vorzeichen oder der Wert des Operanden verändert, um den benötigten Bit zu erhalten. Dadurch ist es möglich, immer den BigInteger darzustellen, mit dem die Operation durchgeführt wird, da immer nur ein einzelner Bit bearbeitet wird. BigInteger ist in der Lage, sich auf ein einzelnes Bit zu konzentrieren. Dies bietet den Vorteil, dass Buchstaben, Zahlen und Zeichen nicht verändert werden, auch wenn diese in verschiedene Blöcke aufgeteilt werden. Bei der Zusammenführung der verschlüsselten/entschlüsselten Blöcke ist es daher möglich, diese ohne Veränderung des Klartextes/Kryptotextes in ihre Ursprungsform zurückzugeben. Bei der Trennung eines Bits auf mehrere Blöcke wird dieser erweitert, sodass er am Ende unverändert trotz Spaltung vorliegt. Bit-Operationen stellen eine unendliche Wertegröße zur Verfügung. Alle Methoden geben eine *NullPointerException* zurück, wenn eine Operation mit einem Wert arbeitet, der *null* ist.[34]

---

[33] Vgl. Freiermuth u.a., a.a.O., S. 282, S. 326-328
[34] https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html

- **Methode *multiply()*:** zu multiplizierende Zahl.muliply(Variable/Wert mit der multipliziert wird)
- **Method *subtract()*:** zu subtrahierende Variable/ Wert.subtract(Variable/Wert mit der subtrahiert wird)
- **Methode *gcd()*:** erste Variable/erster Wert zur Berechnung des ggT.gcd(zweite Variable/zweiter Wert zur Berechnung des ggT)
- **Methode *modInverse()*:** Element^{-1} (= das inverse Element).modInvers(Variable/Wert, mit dem modulo gerechnet wird) = Element^{-1} modulo Variable/Wert
- **Methode *mod()*:** Variable/Wert.mod(Variable/Wert) = Variable/Wert mod Variable/Wert
- **Methode *modPow()*:** Variable/Wert als Basis.modPow(erster Wert: Variable/Wert als Exponent; zweiter Wert: Variable/Wert, mit dem modulo gerechnet wird)

Die **Klasse SecureRandom** stellt einen kryptologisch, starken Zufallsgenerator (RNG = random number generator) zur Verfügung. Ein Objekt, welches von SecureRandom erzeugt wird, gleicht nur wenig statistisch erzeugten Zufallszahlen. Dadurch wird ein nicht deterministischer Output (eine Ausgabe) erzeugt, sodass jeder Output kryptologisch stark ist. Ein erzeugtes Objekt von SecureRandom ist unvorhersehbar.[35]

7.3 Erklärung der Entwicklung einer GUI

Die GUI zum RSA-Verfahren wird mit der BlueJ Extension *Simple GUI Extension* in Java implementiert. Durch das Programm wird die Erstellung der GUI vereinfacht, da man Komponenten grafisch erzeugen kann. Zudem wird zu den eingefügten Objekten sofort Code erzeugt, wodurch sich wiederholende Abläufe, wie die Positionierung von Textfeldern und Labeln, übernommen werden. Ziel der GUI ist die Schaffung einer benutzerfreundlichen Umgebung für den Einsatz des RSA-Verfahrens. Die GUI greift auf die Klasse RSA zu, indem sich durch den *JButton „Schlüsselgenerierung“* der Konstruktor der Klasse RSA *public RSA(int bits)* aufrufen lässt. Der Wert für den Parameter *bits* des Typen Integer wird dabei dem Wert des *JTextFields „Bitlänge der Zahlen p und q“* entnommen und im angegebenen Konstruktor verwendet. Der *JButton „Reset“* übernimmt die Eigenschaften der Methode *Reset()* aus der Klasse RSA-Verfahren. Wird dieser betätigt, werden ausgegebene Werte, die in den *JTextfields* stehen,

[35] https://docs.oracle.com/javase/7/docs/api/java/security/SecureRandom.html

gelöscht und es ist möglich, erneut die Schlüssel generieren zu lassen. Außer den *JTextFields* „*Bitlänge der Zahlen p und q*“, „*Klartext*“, „*Kryptotext*“, „*e*“ und „*n*“ werden von keinen anderen Felder Eingabeaufforderungen verlangt, sondern nur die Werte ausgegeben. Um den *JButton* „*Public Key*“ zu verwenden, ist es erforderlich, Werte für *e* und *n* einzutragen, da der Button „*Public Key*“ den Konstruktor *public RSA* (BigInteger *newe*, BigInteger *newn*) der Klasse RSA aufruft und sich somit dessen Funktionen verwenden lassen. Gibt man einen Text in einem der *JTextFields* „*Klartext*“ oder „*Kryptotext*“ an, so kann man entweder den *JButton* „*Verschlüsseln*“ oder „*Entschlüsseln*“ betätigen. Die *JButtons* rufen die gleichnamigen Methoden aus der Klasse RSA auf und ermöglichen die Ausgabe eines Kryptotextes oder Klartextes. Nachdem man auf ein *JButton* geklickt hat, wird der eingegebene Text im dazugehörigen *JTextField* gelöscht und durch seinen Gegenpart ersetzt (entschlüsseln = Kryptotext und verschlüsseln = Klartext).

7.4 Erklärung der ASCII-Codierung

Im American Standard Code for Information Interchange (ASCII) wird jeder Buchstabe, jede Ziffer, jedes Zeichen durch eine entsprechende Zahlenfolge zwischen 255 und 00 dargestellt. Die ASCII-Zahlenfolge wird mit 7 Bit oder auch mit 8 Bit dargestellt. Die 8 Bit-Codierung wurde nachträglich eingeführt und macht die Darstellung von Umlauten in ASCII möglich.[36] ASCII ist in Java durch Integer darstellbar.[37] Dies bedeutet, dass auch BigInteger die ASCII-Codierung ermöglicht, da es die Eigenschaften von Integer übernimmt und erweitert.

[36] https://www.xovi.de/wiki/ASCII-Code
[37] https://www.mkyong.com/java/how-to-convert-character-to-ascii-in-java

7.5 Glossar

Brute-Force-Methode: „reines Ausprobieren“, solange keine Lösung gefunden wurde

Compiler: Quellcode (z.B. Java) wird in eine für den Computer lesbare Sprache übersetzt.

Digitale Signaturen: mithilfe des privaten Schlüssels des Inhabers wird aus den Daten eine Zahl generiert. Durch eine digitale Signatur ist es möglich, den Urheber der Daten zuzuordnen

Elliptic Curve Cryptography: (kurz: **ECC**) Verfahren auf Basis der elliptischen Kurve, basiert auf dem diskreten Logarithmus von reellen Zahlen und auf Operationen mit Punkt-Paaren bei bestimmten elliptischen Kurven

Faktorisierung: ein Term, der eine Summe oder Differenz ist, wird in ein Produkt verwandelt

Faktorisierungsproblem: Unterteilung einer vorgegebenen Zahl in ein Produkt aus Primfaktoren

Falltürfunktion: (Einwegfunktion) mathematische Funktionen, die schwer umkehrbar sind

ggT: größter gemeinsamer Teiler zweier oder mehrerer Zahlen

GUI: Graphical User Interface

HTTPS: (Hyper Text Transfer Protocol Secure) wird in der Adressleiste eines Browsers angezeigt, wenn eine Verbindung durch SSL/TLS abgesichert ist

.jar-File: ein ausführbares Java-Programm

kongruent: (≡) zwei Zahlen, die einen gleichen Rest liefern, wenn diese durch dritte geteilt werden

Linearkombination: werden häufig bei Gleichungen mit Vektoren verwendet. Zahlen die addiert werden, erhalten einen Vorfaktor; Umformung einer Addition in ein lineares Gleichungssystem

null: ein leerer Wert; Variablen in Programmiersprachen kann dieser Wert zugeschrieben werden (nicht der Zahl 0 entsprechend)

OpenPGP: basierend auf PGP (Pretty Good Privacy), wird zur Verschlüsselung des E-Mailverkehrs benutzt

Skalar: reelle Zahl; eine Größe, die durch eine Zahl vollständig beschrieben ist

SSH: Secure Shell (oder Secure Socket Shell); Protokoll, um Server fernzuwarten, das einen sicheren Zugriff auf andere Computer ermöglicht, da die Kommandos verschlüsselt sind

SSL: Secure Sockets Layer; Standard zur Absicherung von Internetverbindungen und sensibler Daten beim Verkehr zwischen zwei Systemen unter Verwendung von Verschlüsselungsalgorithmen

S/MIME: Secure/Multipurpose Internet Mail Extensions; weltweit etablierter Standard für die Verschlüsselung von E-Mails, basiert auf einem hybriden Kryptosystem

TLS: Transport Layer Security; aktualisierte Version von SSL, die mehr Sicherheit bietet

$\mathbb{Z}$**:** elementare Zahlenmenge der ganzen Zahlen; enthält 0 sowie alle ganzen natürlichen Zahlen (keine Fließkommazahlen) und alle Gegenzahlen (negative ganze Zahlen)

$\in$**:** Elemente einer Zahlenmenge; z.B. $x \in \mathbb{Z}_n$ bedeutet: x ist ein Element der Zahlenmenge bis n

{}: Sammlung von Elementen; z.B. $w \in \{0, \ldots, n-1\}$ bedeutet: w ist ein Element der Sammlung {0, …, n-1}

7.6 Literaturverzeichnis

1 Freiermuth, Karin; Hromkovič, Juraj; Keller, Lucia; Steffen, Björn, Einführung in die Kryptologie, Lehrbuch für Unterricht und Selbststudium, Zürich 2010

2 Bundschuh, Peter, Einführung in die Zahlentheorie, Heidelberg[4] 1998

7.7 Quellenverzeichnis

1 http://deacademic.com/dic.nsf/dewiki/107671

2 http://deacademic.com/dic.nsf/dewiki/335478

3 http://deacademic.com/dic.nsf/dewiki/428220

4 https://www.elektronik-kompendium.de/sites/net/1910131.htm

5 https://www.elektronik-kompendium.de/sites/net/1910151.htm

6 https://www.openpgp.org/about

7 https://www.psw-group.de/smime

8 https://www.searchsecurity.de/definition/Secure-Shell-SSH

9 https://www.websecurity.symantec.com/de/de/security-topics/what-is-ssl-tls-https

10 https://www.philipphauer.de/info/info/asymmetrische-verschluesselung

11 https://www.staff.uni-mainz.de/pommeren/DSVorlesung/KryptoBasis/RSA.html

12 https://www.tecchannel.de/a/digitale-signaturen-werden-10-000-mal-schneller,2023433

13 https://www.inf-schule.de/grenzen/komplexitaet/primfaktorzerlegung/primzahlen

14 https://www.searchsecurity.de/definition/Elliptische-Kurven-Kryptografie-Elliptic-Curve-Cryptography-ECC

15 https://www.researchgate.net/publication/262294724_MQQ-SIG_an_ultra-fast_and_provably_CMA_resistant_digital_signature_scheme

16 http://www.was-ist-malware.de/it-sicherheit/brute-force-attacke

17 http://www.inf.fh-flensburg.de/lang/krypto/protokolle/rsa.htm

18 http://www.inf.fh-flensburg.de/lang/krypto/algo/euklid.htm

19 https://www.mathebibel.de/faktorisieren

20 https://www.mathebibel.de/skalar

21 https://www.rapidtables.com/math/symbols/Set_Symbols.html

22 https://www.frustfrei-lernen.de/mathematik/kongruenz-zahlentheorie.html

23 https://www.duden.de/rechtschreibung/kongruent

24 https://www.studimup.de/lineare-algebra/vektorr%C3%A4ume/linearkombinationen

25 https://www.heise.de/security/artikel/Asymmetrische-Verfahren-271154.html
26 https://www.heise.de/security/artikel/Weitere-Gefahren-271156.html
27 https://docs.oracle.com/javase/7/docs/api/java/security/SecureRandom.html
28 https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html
29 https://nanopdf.com/download/kapitel-7-primitivwurzeln_pdf
30 https://www.xovi.de/wiki/ASCII-Code
31 https://www.xovi.de/wiki/Compiler
32 https://www.mkyong.com/java/how-to-convert-character-to-ascii-in-java/
33 http://www.mathepedia.de/Eulersche_Phi-Funktion.html
34 https://fileinfo.com/extension/jar

Letzter Zugriff auf die Internetseiten (1-34) erfolgte am 12.04.2018.

7.8 Abbildungsverzeichnis

Abbildung 1 (Deckblatt):
http://www.lerevenu.com/sites/site/files/field/image/cyberattaque.-fotolia.jpg

Abbildung 2 (Deckblatt):
https://www.staff.uni-mainz.de/pommeren/DSVorlesung/KryptoBasis/rsasq192.jpg

Letzter Zugriff auf die Internetseiten (Abbildung 1 und 2) erfolgte am 12.04.2018.

8 Versicherung der selbstständigen Erarbeitung

Ich versichere, dass ich die vorliegende Arbeit einschließlich evtl. beigefügter Zeichnungen, Kartenskizzen, Darstellungen u. ä. m. selbstständig angefertigt und keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle Stellen, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem Fall unter genauer Angabe der Quelle deutlich als Entlehnung kenntlich gemacht.

Swisttal-Morenhoven, den 12.04.2018

(Ort) (Datum)

_____

(Unterschrift)