Facharbeit im Leistungskurs Mathematik

Schuljahr 2008/2009

Public-Key-Kryptographie am Beispiel des RSA-Verfahrens: heutige Umsetzung und Implementierung

Kai Lüke

Betreuung: Herr Karmann

Abgabetermin: 20.04.2009

Kurzfassung

Diese Facharbeit behandelt das Prinzip der asymmetrischen Verschlüsselungsverfahren anhand des RSA-Algorithmus, welcher 1977 entwickelt wurde. Für Verschlüsselung und Entschlüsselung werden verschiedene Schlüssel verwendet, sodass der Schlüssel, welcher nur für die Verschlüsselung zuständig ist, öffentlich bekannt sein darf. Die vorliegende Arbeit liefert einen Einblick in die mathematischen Grundlagen, die Funktionsweise und die möglichen Angriffe und Aspekte, welche bei der Umsetzung beachtet werden sollten. RSA gilt bei den richtigen Parametern und Voraussetzungen als sicher, die Angriffe richten sich an Implementierungen und Protokolle. Es werden einige Optimierungen erläutert, die in der anschließenden Implementierung in C++ teilweise umgesetzt werden.

Inhaltsverzeichnis

1 Einleitung

Bis zur Mitte der zweiten Hälfte des 20. Jahrhunderts ging man davon aus, dass nur symmetrische Kryptographie möglich ist, d.h. Sender und Empfänger den gleichen geheimen Schlüssel benötigen. Die Kryptographie wurde 1976 jedoch um die Public-Key-Verfahren erweitert, als Whitfield Diffie und Martin Hellman in ihrer Arbeit *„New Directions in Cryptography“*[1] nicht nur einen sicheren Schlüsselaustausch über einen unsicheren und abhörbaren Kanal publizierten, sondern auch Ideen zu asymmetrischer Verschlüsselung vorschlugen, die das Schlüsselverteilungsproblem bei symmetrischen Verfahren, sowie das Signaturproblem lösten, ohne jedoch ein Public-Key-Verfahren zu kennen. 1977 begann die mehrere Monate dauernde Suche nach einem solchen Verfahren durch Ronald L. Rivest, Adi Shamir und Leonard Adleman, welche ursprünglich zu zeigen versuchten, dass Public-Key-Kryptographie unmöglich sei.[2] Ein Stück klassischer Zahlentheorie erwies sich als nützlich, sodass sie den spektakulärsten Einzelbeitrag zur Public-Key-Kryptographie leisteten und dem gefundenen und bis heute als sicher geltenden Verfahren ihre Anfangsbuchstaben als Namen gaben.[3] RSA war das erste bekannte Public-Key-Verfahren und hat bis heute, vor allem in der Informationstechnik, nur wenig an Bedeutung verloren. Es wird in E-Mail-Verschlüsselung wie *PGP/GnuPG*, sicheren Internet-Verbindungen über *SSL* und digitalen Zertifikaten verwendet. RSA und die mathematischen Grundlagen haben die Kryptographie revolutioniert und so lohnt es sich, die Funktionsweise und deren Schwächen zu verstehen, da sie unweigerlich im digitalen Alltag vorkommen. Einen weiteren Bestandteil dieser Arbeit bildet die Optimierung des RSA-Algorithmus durch zahlentheoretische Sätze. Zudem wird die Sicherheit von RSA und dessen Anwendung hinterfragt und Angriffe auf das Verfahren bzw. dessen Umsetzung gezeigt. Abschließend soll diese Arbeit in einer beispielhaften Implementierung des RSA-Verfahrens in C++ unter Verwendung der *GNU Multiple Precision Arithmetic Library* münden.

1 Diffie, W. und Hellman, M.E.: New Directions in Cryptography. IEEE Transactions on Information Theory, IT 22,6. 1976. S. 644-654.
2 Vgl. Beutelspacher, A., Schwenk, J. und Wolfenstetter, K.-D.: Moderne Verfahren der Kryptographie. 6., überarb. Aufl. Wiesbaden: Vieweg 2006. S. 17.
3 Vgl. Diffie, W.: The First Ten Years of Public-Key Cryptography. Proceedings of the IEEE 76,5. 1988. S. 560-577. In dt. Übers.: Beutelspacher, A.: Kryptologie. 5., überarb. Aufl. Wiesbaden: Vieweg 1996. S. 123.

2 Public-Key-Kryptographie

2.1 Das Prinzip der asymmetrischen Kryptosysteme[4]

Jeder Teilnehmer T besitzt ein Schlüsselpaar, welches aus einem öffentlichen Schlüssel $E = E_T$ zur Verschlüsselung („*E*ncryption“) und einem geheimen Schlüssel $D = D_T$ besteht, welcher für die Entschlüsselung („*D*ecryption“) verantwortlich ist. Diese zeichnen sich durch folgende Eigenschaften aus:

- Aus der Kenntnis von E_T, welcher jedem anderen Teilnehmer bekannt sein sollte, darf man nicht auf D_T, schließen können
- Es muss mit einem geeigneten Algorithmus f gelten: $f_D(f_E(m)) = m$
- Es kann also nur T eine mit E_T verschlüsselte Nachricht $f_{E_T}(m) = c$ entschlüsseln, indem er $f_{D_T}(c) = m$ berechnet

Will Alice an Bob[5] eine Nachricht m schicken, so sucht sie seinen öffentlichen Schlüssel E_B heraus und wendet f_E mit E_B auf m an. Das Ergebnis c sendet sie an Bob, welcher als einziger die Nachricht entschlüsseln kann, indem er f_D mit D_B auf c anwendet.

2.2 Funktionsweise

Ein Public-Key-Verfahren ist mit einer Reihe von Briefkästen vergleichbar. Alice wirft eine Nachricht in Bobs Briefkasten. Niemand kann die Nachricht lesen außer Bob, welcher den Briefkasten mit seinem Schlüssel öffnet und die Nachricht entnimmt. Dieses Falltürprinzip liegt den asymmetrischen Verschlüsselungsverfahren zugrunde. Es wird eine annähernd kollisionsfreie[6] Falltürfunktion (auch Trapdoor-Einwegfunktion genannt) für jeden Teilnehmer benötigt, die sich nur mit einer Geheiminformation rückgängig machen lässt und für alle anderen Teilnehmer eine Einwegfunktion darstellt.

Eine Falltürfunktion wäre z.B. die Potenzfunktion $x \to x^e \bmod n$ mit $n = p \cdot q$, welche ohne Kenntnis von p und q nur schwer umkehrbar ist.[7]

4 Vgl. Beutelspacher, A.: Kryptologie. 5., überarb. Aufl. Wiesbaden: Vieweg 1996. S. 114f.
5 Siehe Anhang 6.1
6 d.h. es ist höchst unwahrscheinlich, dass die Funktion bei unterschiedlichen Parametern das gleiche Ergebnis liefert
7 Vgl. Beutelspacher, A., Schwenk, J. und Wolfenstetter, K.-D.: Moderne Verfahren der Kryptographie. 6., überarb. Aufl. Wiesbaden: Vieweg 2006. S. 13.

2.3 Unterschiede zu symmetrischen Kryptosystemen

Da der Gesprächspartner nur den öffentlichen Schlüssel benötigt, kann der Schlüsselaustausch über einen abhörbaren Kanal erfolgen. Bei symmetrischen Verfahren würde dies die Verschlüsselung unbrauchbar machen, falls nicht der Diffie-Hellman-Schlüsselaustausch zur Berechnung eines gemeinsamen Schlüssels verwendet wird. Die Gesamtanzahl der Schlüssel bei Public-Key-Verfahren beträgt das Doppelte der Teilnehmeranzahl. Da bei einem symmetrischen Verfahren mit jedem Teilnehmer ein einzelner geheimer Schlüssel vereinbart werden muss, steigt die Schlüsselanzahl quadratisch und beträgt bei n Teilnehmern $n \cdot (n-1)/2$. [8] Jeder Teilnehmer muss $n-1$ Schlüssel speichern, während bei einem Public-Key-Verfahren die Schlüssel auch in einer öffentlichen Liste verfügbar sein können.

2.4 Anwendung als Signaturverfahren

Ein Public-Key-Verfahren kann auch unter der Bedingung, dass $f_E(f_D(m)) = m$ gilt, als Signaturverfahren benutzt werden. Dazu berechnet Alice für ihre Nachricht m $f_{D_A}(m) = s$ und veröffentlicht das Ergebnis s als Unterschrift mit ihrer Nachricht. Nur sie kann s berechnen, da niemand sonst ihren privaten Schlüssel D_A kennt. Bob kann überprüfen, ob Alice wirklich diese Nachricht geschrieben hat oder ob m unterwegs verändert wurde, indem er $f_{E_A}(s) = m'$ berechnet. Das Ergebnis m' ist im korrekten Fall m. Falls aber die Nachricht verändert wurde oder die Signatur nicht von Alice stammt, so stimmt m' nicht mit m überein.

2.5 Probleme in Public-Key-Strukturen

Public-Key-Verfahren ermöglichen zwar eine spontane und, anders als der Diffie-Hellman-Schlüsselaustausch, eine einseitige oder zeitlich versetzte Kommunikation, jedoch keine Sicherheit gegen *Man-in-the-middle-Angriffe*. Es würde Mallory[9] gelingen, sich zwischen die Kommunikation von Alice und Bob zu schalten und Alice und Bob seinen öffentlichen Schlüssel anstelle der auszutauschenden Schlüssel von Alice und Bob zu schicken und ihn als solchen zu kennzeichnen. Dies würde ein Abhören und Modifizieren der gesendeten Nachrichten ermöglichen, da er sie mit

8 Vgl. Beutelspacher, A.: Kryptologie. 5., überarb. Aufl. Wiesbaden: Vieweg 1996. S. 118.
9 Siehe Anhang 6.1

seinem privaten Schlüssel entschlüsseln könnte und eine beliebige Nachricht mit dem passenden Public-Key verschlüsselt verschicken könnte.

Das Problem liegt darin, dass sie nicht wissen, ob der erhaltene Schlüssel wirklich dem erwarteten Gesprächspartner gehört. Außerdem ist nicht bekannt, wer die Nachricht schrieb, die sie erhalten haben, wenn sie nicht von dem Absender signiert wurde. Es ist also, damit kein gefälschter Schlüssel angenommen wird, eine dritte Person Trent[10] nötig, der Alice und Bob vertrauen und daher von ihr keine Manipulation erwarten. Trent signiert öffentliche Schlüssel nur, wenn diese auch dem angegebenen Besitzer gehören. Dies geschieht z.B. über eine Personalausweiskontrolle. So kann Mallory nicht seinen Schlüssel als den von Alice oder Bob ausgeben, weil er für diesen keine Signatur von Trent erhält, welche von Alice oder Bob auf Gültigkeit überprüft werden muss.

2.6 Eine Auswahl von Public-Key-Algorithmen

RSA ist zwar das erste Public-Key-Kryptosystem, jedoch gab es nach 1977 auch andere Alternativen. So hat Robert McEliece 1978 ein Verfahren entwickelt, welches Matrizen und einen fehlerkorrigierenden Code verwendet. Das McEliece-Kryptosystem gilt als sicher, wird aber fast nicht verwendet, da es einen langen öffentlichen Schlüssel benötigt und die Chiffre etwa doppelt so lang wie der Klartext ist.[11] 2008 wurde ein Angriff bekannt, welcher das System unter Verwendung der von McEliece benutzten Parameter erfolgreich brach[12]. Ein Cluster von 200 2,4 GHz Core 2 Quad CPUs benötigt 7 Tage.

Michael O. Rabin veröffentlichte 1979 ein mit RSA verwandtes Verfahren. Auch dieses findet fast keine Verwendung, da es durch einen *Angriff mit frei wählbarem Geheimtext* unter Vergleich zweier Datenpaare brechbar ist.[13]

Ein heute oft benutztes und wegen der Patentfreiheit weit verbreitetes Verfahren wurde 1985 von Taher Elgamal entwickelt. Es beruht auf dem Diffie-Hellman-Schlüsselaustausch und die Sicherheit somit auf dem *diskreten Logarithmus-Problem*.[14] Es gehört mit RSA zu den wichtigsten Verfahren, kommt aber schon mit

10 Siehe Anhang 6.1
11 http://glossar.hs-augsburg.de/index.php?title=McEliece-Kryptosystem&oldid=8170. 22.03.2009.
12 http://en.wikipedia.org/w/index.php?title=Attacking_McEliece_by_finding_low-weight_codewords&oldid=277511445. 22.03.2009.
13 http://de.wikipedia.org/w/index.php?title=Rabin-Kryptosystem&oldid=55128885. 22.03.2009
14 Vgl. Beutelspacher, A.: Kryptologie. 5., überarb. Aufl. Wiesbaden: Vieweg 1996. S. 141f.

kleineren Schlüsseln aus. Von den vielen Modifikationen lohnt es sich noch, Elliptische-Kurven-Kryptosysteme zu nennen, welche bei gleicher Sicherheit die Schlüssellänge erheblich verkleinern und somit auf Smartcards Anwendung finden.[15]

3 Das RSA-Verfahren

3.1 Zahlentheoretische Voraussetzungen

Der Gültigkeit des RSA-Algorithmus liegen einige zahlentheoretische Sätze zugrunde. Diese sind für das Verständnis von RSA unabdingbar und werden hier erklärt.

3.1.1 Der Satz von Euler-Fermat

Der *Kleine Satz von Fermat* besagt, dass für eine Primzahl p und eine nicht durch p teilbare kleinere ganze Zahl a gilt $a^{p-1} \bmod p = 1$ und $a^p \bmod p = a$.

Leonhard Euler bewies die Verallgemeinerung dieses Satzes, welche als *Satz von Euler-Fermat* bekannt ist. Es wird die E*ulersche φ-Funktion* benutzt, welche für eine natürliche Zahl n die Anzahl der zu n teilerfremden $(ggT(n,x) = 1)$ natürlichen Zahlen liefert, die nicht größer sind als n. Für eine Primzahl p ist $\varphi(p) = p-1$, da die Zahlen $1 \ldots p-1$ teilerfremd zu p sind. Für p ist $ggT(p,p) = p$ und nicht 1, d.h. p ist nicht teilerfremd zu p. Für eine Zahl $n = p \cdot q$ gilt $\varphi(n) = (p-1) \cdot (q-1)$. Das zeigt sich aus folgenden Tatsachen[16]; es gibt $p \cdot q - 1$ mögliche Zahlen, von denen die Anzahl der zu n nicht teilerfremden Zahlen abgezogen werden muss, welche sind: $p, 2 \cdot p, \ldots, (q-1) \cdot p$ sowie $q, 2 \cdot q, \ldots, (p-1) \cdot q$. So ergibt sich $\varphi(p \cdot q) = p \cdot q - 1 - (q-1) - (p-1) = p \cdot q - q - p + 1 = (p-1) \cdot (q-1)$. Der *Satz von Euler-Fermat* besagt, dass für jede natürliche Zahl a mit $a \leq n \ \wedge \ a \nmid n \ \wedge \ n \nmid a$ gilt $a^{\varphi(n)} \bmod n = 1$.

Daraus folgt $a^{k \cdot \varphi(n)+1} \bmod n = a^{\varphi(n)^k} \cdot a \bmod n = 1^k \cdot a \bmod n = a$.

Wird der spezielle Fall mit $n = p \cdot q$ betrachtet, so lautet der Satz $a^{\varphi(n)} \bmod n = a^{(p-1)\cdot(q-1)} \bmod p \cdot q = a^{(p-1)^{(q-1)}} \bmod p \cdot q = 1^{(q-1)} \bmod q = 1$.

15 Vgl. Schmeh, K.: Kryptografie. 3. erw. Aufl. Heidelberg: dpunkt.verlag 2007. S. 186ff.
16 Vgl. Beutelspacher, A.: Kryptologie. 5., überarb. Aufl. Wiesbaden: Vieweg 1996. S. 124.

Beweis des Satzes von Euler[17]:

Es sei $(\mathbb{Z}/n\mathbb{Z})^x=\{r_1,\ldots,r_{\varphi(n)}\}$ die Menge der multiplikativ modulo n invertierbaren Elemente, also aller Reste, die bei der Division durch n auftreten können. Für jedes a mit $ggT(a,n)=1$ ist dann $x\to a\cdot x$ eine Permutation von $(\mathbb{Z}/n\mathbb{Z})^x$, d.h. die Multiplikation aller Elemente mit $a\ (mod\, n)$ ergibt lediglich eine andere Reihenfolge der Elemente, denn aus $a\cdot x \bmod n = a\cdot y \bmod n$ folgt $x \bmod n = y \bmod n$ und somit kann kein Element mehrfach vorkommen. Weil die Multiplikation kommutativ ist, folgt $r_1\cdots r_{\varphi(n)} \bmod n = (a\cdot r_1)\cdots(a\cdot r_{\varphi(n)}) \bmod n = r_1\cdots r_{\varphi(n)}\cdot a^{\varphi(n)} \bmod n$ und da die r_i alle invertierbar sind für alle i, gilt $1 \bmod n = a^{\varphi(n)} \bmod n$.

Der Sonderfall für $n=p\cdot q$ und $a|n$: Für $a|n$, also $a=p\vee a=q$ gilt zwar nicht $a^{k\cdot\varphi(n)} \bmod n=1$, aber $a^{k\cdot(p-1)(q-1)+1} \bmod n = a$. Dies wird für $p<q$ mit dem *Chinesischen Restsatz* aus 3.1.3 gezeigt. Die Berechnung von $m=p^{k\cdot\varphi(n)+1} \bmod p\cdot q$ wird in zwei Teile aufgeteilt: $m_p=p^{k\cdot\varphi(n)+1} \bmod p=0$ und $m_q = p^{k\cdot(p-1)\cdot(q-1)+1} \bmod q = p$. Es gilt $m = m_p+(u\cdot(m_q-m_p) \bmod q)\cdot p$ für $ggT(p,q) = u\cdot p+z\cdot q = 1 = u\cdot p \bmod q$ und somit $m = 0+(u\cdot p \bmod q)\cdot p = p$.

Das gilt auch für $m = q^{k\cdot\varphi(n)+1} \bmod p\cdot q$: $m_p = q^{k\cdot\varphi(n)+1} \bmod p = q-p$ und $m_q=q^{k\cdot\varphi(n)+1} \bmod q=0$ ergeben die Lösung $m = q-p+(-u\cdot q+u\cdot p \bmod q)\cdot p = q$.

3.1.2 Der Erweiterte Euklidische Algorithmus[18]

Um den größten gemeinsamen Teiler (ggT) zweier Zahlen effektiv zu bestimmen, kann der *Euklidische Algorithmus* verwendet werden. Seinen a und b natürliche Zahlen und $a>b$. Sei b kein Teiler von a. Dann gibt es eine Folge von Zahlen $r_1,\ldots,r_n$, sodass

$$
\begin{aligned}
a &= q_1\cdot b + r_1 \quad mit \quad 0 < r_1 < b,\\
b &= q_2\cdot r_1 + r_2 \quad mit \quad 0 < r_2 < r_1,\\
r_1 &= q_3\cdot r_2 + r_3 \quad mit \quad 0 < r_3 < r_2,\\
r_2 &= q_4\cdot r_3 + r_4 \quad mit \quad 0 < r_4 < r_3,\\
&\ldots\\
r_{n-2} &= q_n\cdot r_{n-1} + r_n \quad mit \quad 0 < r_n < r_{n-1},\\
r_{n-1} &= q_{n+1}\cdot r_n
\end{aligned}
$$

gilt. Es ist $ggT(a,b) = r_n$.

17 Vgl. http://de.wikipedia.org/w/index.php?title=Satz_von_Euler&oldid=55334706. 03.04.2009.
18 Vgl. Beutelspacher, A.: Kryptologie. 5., überarb. Aufl. Wiesbaden: Vieweg 1996. S. 125ff.

Nach dem *Lemma von Bézout* ist eine Vielfachsummendarstellung des ggT g von a und b möglich: $g = x \cdot a + y \cdot b$. Die Faktoren x und y lassen sich durch den *Erweiterten Euklidischen Algorithmus* bestimmen. Parallel zum Fortschritt der Berechnungen von z.B. r_2 wird der Rest r_4 als Linearkombination von a und b dargestellt. $r_4 = 1 \cdot r_2 - q_4 \cdot r_3 = 1 \cdot (1 \cdot b - q_2 \cdot r_1) - q_4 \cdot (1 \cdot r_1 - q_3 \cdot r_2) = 1 \cdot (1 \cdot b - q_2 \cdot (1 \cdot a - q_1 \cdot b)) - q_4 \cdot (1 \cdot (1 \cdot a - q_1 \cdot b) - q_3 \cdot (1 \cdot b - q_2 \cdot r_1)) = 1 \cdot (1 \cdot b - q_2 \cdot (1 \cdot a - q_1 \cdot b)) - q_4 \cdot (1 \cdot (1 \cdot a - q_1 \cdot b) - q_3 \cdot (1 \cdot b - q_2 \cdot (1 \cdot a - q_1 \cdot b))) = (-q_2 - q_4 - q_3 \cdot q_2) \cdot a + (q_2 \cdot q_1 + 1 + q_4 \cdot q_1 + q_3 + q_3 \cdot q_2 \cdot q_1) \cdot b = x \cdot a + y \cdot b$ Da die vorherigen Werte im Normalfall gemerkt und eingesetzt werden können, ist eine schnelle Berechnung der Faktoren möglich.

3.1.3 Der Chinesische Restsatz[19]

Eine Berechnung von $x \mod n$ mit $n = p \cdot q$ und $ggT(p,q) = 1$ kann auf die Berechnungen modulo p und q zurückgeführt werden. Nach dem *Chinesischen Restsatz* werden für den hier benötigten Spezialfall die Teilergebnisse zusammengesetzt. Die Zahlen p und q sind teilerfremd, es gilt $1 = s \cdot p + t \cdot q$. Der *Erweiterte Euklidische Algorithmus* liefert s und t. Die Teillösungen werden berechnet: $a = x \mod p$ $b = x \mod q$

Für die Gesamtlösung gilt $x \mod n = b \cdot sp + a \cdot tq \mod n$.

Beweis: Es ist $s \cdot p \bmod p = 0$, $t \cdot q \bmod p = 1$, $s \cdot p \bmod q = 1$, $t \cdot q \bmod q = 0$.

$$x \mod p = b \cdot 0 + a \cdot 1 \mod p = b \cdot sp + a \cdot tq \mod p$$

$$x \mod q = b \cdot 1 + a \cdot 0 \mod q = b \cdot sp + a \cdot tq \mod q$$

Das kleinste gemeinsame Vielfache von p und q ist $p \cdot q$. Beidseitige Addition von $p \cdot q$ ändert die Gültigkeit der Gleichungen nicht, also $x \equiv b \cdot sp + a \cdot tq \pmod{pq}$.

Eine andere Möglichkeit, das System aus Kongruenzen aufzulösen ergibt sich wie folgt:[20] Es ist $x = a + u \cdot p = b + v \cdot q$ und daher $b - a = u \cdot p + (-v) \cdot q$. Die oben genannte Gleichung $1 = s \cdot p + t \cdot q$ lässt sich mit $b - a$ multiplizieren.

$b - a = s \cdot p \cdot (b - a) + t \cdot q \cdot (b - a) \Rightarrow u = s \cdot (b - a) \wedge v = -t \cdot (b - a) = t \cdot (a - b)$

$\Rightarrow x = a + s \cdot (b - a) \cdot p = b + t \cdot (a - b) \cdot q$

19 Vgl. Beutelspacher, A., Schwenk, J. und Wolfenstetter, K.-D.: Moderne Verfahren der Kryptographie. 6., überarb. Aufl. Wiesbaden: Vieweg 2006. S. 112.

20 Vgl. Bartholomé, A., Rung, J. und Kern, H.: Zahlentheorie für Einsteiger. 6. Aufl. Wiesbaden: Vieweg+Teubner 2008. S. 103.

3.2 Der RSA-Algorithmus

3.2.1 Die Schlüsselerzeugung[21]

Es werden zwei Primzahlen p und q benötigt, aus denen das Produkt $n=p\cdot q$ berechnet wird. Es gilt nun für jede natürliche Zahl $m\leq n$ und jede natürliche Zahl k folgende Gleichung: $m^{k\cdot\varphi(n)+1} \bmod n = m^{k\cdot(p-1)\cdot(q-1)+1} \bmod n = m$.

Es werden zwei natürliche Zahlen e und d gewählt, für die gilt $e\cdot d \bmod \varphi(n)=1$ oder anders ausgedrückt $e\cdot d=k\cdot(p-1)\cdot(q-1)+1$. Wenn e mit der Eigenschaft $ggT(e,\varphi(n))=1$ festgelegt wird, kann man d durch den *Erweiterten Euklidischen Algorithmus* bestimmen, da die Beziehung $ggT(e,\varphi(n)) = x\cdot e+y\cdot\varphi(n)$ aufgestellt werden kann. Mit d als x und $-k$ als y ergibt sich die Gleichung $1 = d\cdot e-k\cdot(p-1)\cdot(q-1)$, die durch Anwenden des *Erweiterten Euklidischen Algorithmus* auf e und $\varphi(n)$, wie vorauszusehen, $ggT(e,\varphi(n))=1$ und $d=x$ liefert. d ist das multiplikative Inverse zu $e \bmod \varphi(n)$.
Die Zahlen n,e und d werden zur Ver- und Entschlüsselung benötigt, wobei $d,\varphi(n),p$ und q geheim bleiben müssen und e sowie n veröffentlicht werden.

3.2.2 Verschlüsselung und Entschlüsselung

Für die Verschlüsselung ergibt sich das Potenzieren mit *e* modulo *n:*

$$f_e(m) = m^e \bmod n = c$$

Das korrekte Entschlüsseln bildet durch den *Satz von Euler-Fermat* das Potenzieren mit e *modulo* n: $f_d(c) = c^e \bmod n = m$

In der Gesamtheit: $f_d(f_e(m)) = (m^e)^d \bmod n = (m^d)^e \bmod n = m$

Die Falltürfunkton bildet das Potenzieren mit e *modulo* n, welches nur mit der Kenntnis von d bzw. p, q oder $\varphi(n)=(p-1)\cdot(q-1)$ rückgängig gemacht werden kann. Allein mit e und n kann man nicht d als multiplikativ Inverses von $e \bmod \varphi(n)$ bestimmen, sodass die Public-Key-Eigenschaft gegeben ist.

21 Vgl. Beutelspacher, A., Schwenk, J. und Wolfenstetter, K.-D.: Moderne Verfahren der Kryptographie. 6., überarb. Aufl. Wiesbaden: Vieweg 2006. S. 17.

3.2.3 Signieren und Verifizieren

RSA kann in beide Richtungen verwendet werden. Das „Verschlüsseln“ einer Nachricht m mit dem privaten Schlüssel d kann somit auch als Signieren aufgefasst werden. Das „Entschlüsseln“ mit dem passenden öffentlichen Schlüssel e liefert die ursprüngliche Nachricht m'. Wenn die Signatur mit der Nachricht zusammen geschickt wird, kann überprüft werden, ob $m = m'$ gilt. Ist dies der Fall, kann man aufgrund der Kollisionsfreiheit sicher sein, dass die Signatur nur mit dem zu e passenden privaten Schlüssel d erzeugt werden konnte und somit von der angenommenen Person stammt.

3.3 Die Sicherheit von RSA

Für kleine verwendete Zahlen ist die Wahrscheinlichkeit auf Kollisionen in den Ergebnissen der Falltürfunktion wegen des kleinen Modulus sehr groß. Auch die Bestimmung des privaten Schlüssels ohne Kenntnis von $\varphi(n)$, und somit das Brechen der Schlüssels, ist für kleine Primfaktoren p und q durchführbar, indem der Modulus n faktorisiert wird und $\varphi(n)=(p-1)\cdot(q-1)$ bestimmt wird. Nun kann die Gleichung zur Bestimmung von einem gültigen d aufgestellt und mit dem *Erweiterten Euklidischen Algorithmus* gelöst werden: $1 = d\cdot e-k\cdot\varphi(n)$. Die Faktorisierung von n ist somit der beste Angriff zur Schlüsselbestimmung, da nur bis maximal $\sqrt{n}$ nach Primfaktoren gesucht werden muss und die vollständige Schlüsselsuche von d länger dauert, da d wegen $e\cdot d=k\cdot\varphi(n)+1$ sehr viel größer als e ist. Auch die vollständige Suche nach $\varphi(n)$ muss genauso schwer oder schwerer sein als die Faktorisierung, da man aus der Kenntnis von $\varphi(n)$ und n sehr schnell auf p und q schließen kann.

$$\varphi(n) = p\cdot q-q-p+1 = n-q-\tfrac{n}{q}+1 \;\Leftrightarrow\; -q-\tfrac{n}{q}-(\varphi(n)-n-1) = 0 \quad |\cdot(-q)$$

$$\Leftrightarrow\; q^2+(\varphi(n)-n-1)\cdot q+n=0 \quad\Rightarrow\quad q_{1/2}=\frac{-(\varphi(n)-n-1)\pm\sqrt{(\varphi(n)-n-1)^2-4\cdot n}}{2}$$

3.3.1 Faktorisierung und Schlüsselgröße

Die Zerlegung einer Zahl in ihre Primfaktoren ist ein aufwändiges Problem, für das es trotz Verbesserungen der Algorithmen, wie z.B. dem *speziellen Zahlenkörpersieb*[22], für große Zahlen immer noch keine effektive Lösung gibt, da die benötigten

22 Vgl. Schneier, B.: Angewandte Kryptographie. Dt. Übers. München: Pearson Studium. 2006. S. 188f.

Rechenoperationen und somit die Laufzeit exponentiell zu der Anzahl der Ziffern steigen. In der Praxis werden daher für RSA zwei Primzahlen gewählt, die multipliziert eine Zahl mit über 309 Dezimalstellen ergeben. Die Schlüssellänge wird meistens in Bit angegeben, d.h. die Anzahl der Binärstellen des Modulus n. Die als sicher geltende Länge hängt von den aktuellen Faktorisierungsrekoden ab. Die größte für RSA sinnvoll einsetzbare Zahl, die faktorisiert wurde, wird *RSA-200* genannt und bestitzt 200 Dezimalstellen bzw. 663 Binärstellen.[23] Es wurde von 2003 bis 2005 ein Zusammenschluss von 80 2,2 GHz Opteron-Prozessoren für die in der Gesamtheit drei Monate dauernde Faktorisierung verwendet. Der Kryptologe Bruce Schneier wagte 1996 eine Empfehlung zur Wahl einer öffentlichen Schlüssellänge mit einem den Anforderungen entsprechenden Sicherheitspuffer:

Jahr	Einzelperson (in Bit)	Unternehmen (in Bit)	Regierung (in Bit)
1995	768	1280	1536
2000	1024	1280	1536
2005	1280	1536	2048
2010	1280	1536	2048
2015	1536	2048	2048

Diese Angaben sind jedoch, so wie er auch einräumt[24], mit Vorsicht zu benutzen, da Zukunftsprognosen durch bahnbrechende Erfindungen und nicht einberechnete Möglichkeiten zunichte gemacht werden. Ron Rivest empfahl 1990 für 2005-2010 die dem obigen Muster entsprechende Schlüssellängenkombination 439, 602 und 1628 Bit[25], wobei die ersten beiden Schlüssellängen schon 2005 wie oben genannt faktorisiert wurden.

3.3.2 Die Wahl des öffentlichen Exponenten

Der öffentliche Exponent e ist ein wichtiger Bestandteil der Sicherheit von RSA. Ergibt das Ergebnis c der Verschlüsselung einen kleineren Wert als n, so kann die *e*-te Wurzel zur Bestimmung der ursprünglichen Nachricht gezogen werden. Bei einer kleinen Differenz zwischen c und n für $c > n$ kann durch Addition von n das korrekte Wurzelziehen ermöglicht werden. Es ist somit also wichtig, dass m genügend groß ist, um bei Exponentieren mit e den Modulus n um ein Vielfaches

23 http://rsa.com/rsalabs/node.asp?id=2879. 28.03.2009.
24 Vgl. Schneier, B.: Angewandte Kryptographie. Dt. Übers. München: Pearson Studium. 2006. S. 197
25 Ebd. S. 191. Tabelle 7.8.

zu übertreffen, sodass die Möglichkeit des Wurzelziehens nicht gegeben ist. Unterstützend wird heute der Exponent $e = 65537$ (hexadezimal 10001, binär 10000000000000001) gewählt[26], da dieser trotz seiner Größe schnell für die Exponentation mit dem *Square-And-Multiply-Algorithmus* benutzbar ist. Auch die *Low Exponent Attack* aus 3.3.6 ist dann verhindert.

3.3.3 Angriff mit frei wählbarem Klartext

Der Angriff mit frei wählbarem Klartext (engl. chosen-plaintext attack) ist bei Public-Key-Verfahren immer möglich, indem alle möglichen Klartexte mit dem Public-Key verschlüsselt und die Ergebnisse mit dem gefundenen Chiffretext verglichen werden. Bei Übereinstimmung eines generierten Chiffretextes mit dem originalen ist der frei gewählte Klartext der originale Klartext. Um diesen Angriff zu verhindern, muss die Nachricht m groß genug sein, um eine vollständige Klartextsuche aus Zeitgründen unmöglich zu machen.

3.3.4 Padding

Neben der Auffüllung zur geeigneten Größe ist das Verknüpfen von *m* mit Zufallsdaten sinnvoll, da dies bei gleichen Klartexten unterschiedliche Chiffretexte liefert und somit auch bei korrekt gewähltem Klartext den Vergleich ungültig macht. Dieses sogenannte *Padding* hat auch den Vorteil, dass bei der mehrfachen Versendung einer Nachricht ein Unterschied im Chiffretext vorhanden ist, der es einen Angreifer nicht erkennen lässt, dass es sich um den gleichen Klartext handelt. Eine weitere Funktion des *Paddings* ist das Einfügen eines Kontrollwertes, mit dem überprüft werden kann, ob die entschlüsselte Nachricht gültig ist. In den Anfängen des *Public Key Cryptography Standards*[27] war dies eine Konstante mit zusätzlicher Auffüllung der Nachricht. Im *Optimal Asymmetric Encryption Padding*[28] wird jedoch eine Prüfsumme der restlichen Nachricht mit eingefügt, die die kryptographischen Eigenschaften im Vergleich zu einer Konstante verbessert. Das einfache *PKCS1*[29]-Padding würde eine Nachricht m_{bin} in der Binärdarstellung der Länge $m_{binLen} < n_{binLen} - 80$ vor dem Verschlüsselungsvorgang wie folgt zusammensetzen:

26 Vgl. RFC 3447 V.1.5. ftp://ftp.rsasecurity.com/pub/pkcs/ascii/pkcs-1.asc. 03.04.2009.
27 Ebd.
28 Bellare, M., Rogaway, P.: Optimal Asymmetric Encryption -- How to encrypt with RSA. Springer-Verlag. 1995. Auch unter: http://www-cse.ucsd.edu/users/mihir/papers/oae.pdf. 29.03.2009.
29 Nach RFC 3447 V.1.5. ftp://ftp.rsasecurity.com/pub/pkcs/ascii/pkcs-1.asc. 03.04.2009.

$m_{bin}' = 00 \,||\, Typennummer \,||\, Konstanter\ Auffüllungsblock\ der\ Länge\ n_{binLen} - 3 \,||\, 00 \,||\, m_{bin}$

Das *OAEP*[30] fügt die Nachricht bedeutend komplexer zusammen. Die Vorgehensweise ist aus *RFC 3447, Abschnitt 7.1.1: Figur 1* zu entnehmen.

3.3.5 Angriffsszenarien ohne Padding und Nachrichtenformatierung

Eine Implementierung ohne Padding ist kaum denkbar, weil sie neben den genannten Nachteilen höchste Aufmerksamkeit über dass, was ver- und entschlüsselt werden soll, erfordert. So kann bei Zugriff auf die Entschlüsselung der Implementierung oder durch Unvorsichtigkeit des Empfängers ein *Chosen Ciphertext* erfolgreich sein. Während der Empfänger bzw. die Implementierung eine verschlüsselte Nachricht erwartet, wird jedoch ein Klartext oder dessen Prüfsumme gesendet und der nach den obigen Voraussetzungen bekannt gewordene Rückgabewert der Entschlüsselung bildet eine gültige Signatur der gesendeten Nachricht. Dies wird durch die Identifikationsnummern, die z.B. der *PKCS1* verteilt, nicht mehr möglich, da die Implementierung erkennen wird, dass die empfangene Nachricht kein Chiffretext ist.

Da die Nachrichten Zahlen sind und $m_1^e \cdot m_2^e \mod n = (m_1 \cdot m_2)^e \mod n$ gilt, kann die Nachricht $m_1 \cdot m_2$ verschlüsselt werden, ohne m_1 oder m_2 zu kennen, was auch für das Signieren gilt, sodass aus den Signaturen $m_1^d \mod n$ und $m_2^d \mod n$ eine gültige Signatur $m_1^d \cdot m_2^d \mod n$ für $m_1 \cdot m_2$ erstellt werden kann, ohne d zu kennen.[31] Diese Nachrichten und Signaturen erkennt man nur an der fehlenden Redundanz der Sprache oder Kodierung nach z.B. *PKCS1*.

Weitere Angriffe auf RSA nach Don Coppersmith ergeben sich, wenn die Nachrichten in einer algebraischen Beziehung stehen. Sind z.B. zwei Chiffretexte bekannt, deren Nachrichten sich um den Betrag 1 unterscheiden - möglich wäre eine geheime Versuchsreihe von Experimenten - so können die ursprünglichen Nachrichten leicht wiederhergestellt werden, wenn der früher oft benutzte öffentliche Exponent 3 gewählt wurde:[32] $c_1 = m^3 \mod n \qquad c_2 = (m+1)^3 \mod n$

30 Nach RFC 3447 V.2.1. http://tools.ietf.org/html/rfc3447. 03.04.2009.
31 Vgl. Beutelspacher, A., Schwenk, J. und Wolfenstetter, K.-D.: Moderne Verfahren der Kryptographie. 6., überarb. Aufl. Wiesbaden: Vieweg 2006. S. 19.
32 Vgl. Ebd. S. 20.

$$\frac{c_2+2\cdot c_1-1}{c_2-c_1+2} = \frac{(m+1)^3+2\cdot m^3-1}{(m+1)^3-m^3+2} = \frac{3\cdot m^3+3\cdot m^2+3\cdot m}{3\cdot m^2+3\cdot m+3} = m$$

3.3.6 Low-Exponent-Attack

Wie genannt wurde lange aus Geschwindigkeitsgründen $e=3$ oder $e=17$ gewählt. Die Verwendung von z.B. 3 ermöglicht eine sogenannte *low-exponent-attack.* Wird eine Nachricht m von drei verschiedenen Teilnehmern verschlüsselt, so ergeben sich die Chiffretexte $y_1 = m^3 \mod n_1$, $y_2 = m^3 \mod n_2$ und $y_3 = m^3 \mod n_3$, welche mit dem *Chinesischen Restsatz* $y = m^3 \mod n_1 \cdot n_2 \cdot n_1$ berechenbar machen. Da $m_i < n_i$ ist, gilt auch $m^3 < n_1 \cdot n_2 \cdot n_1$ und $y = m^3$. D.h. das Ziehen der dritten Wurzel aus y ergibt m.

3.4 Die Implementierung des RSA-Verfahrens

3.4.1 Schnelles Exponentieren[33]

Wie bereits erwähnt, ist die modulo-Exponentiation durch den *Square-and-multiply-Algorithmus* schnell durchführbar. Eine Zahl c wird durch die Folge der Zahlen $b_i \in \{1,0\}$ mit $i = 0...n$ binär dargestellt. Es kann $y = a^c \mod m$ als $a^{1\cdot b_0} \cdot a^{2\cdot b_1} \cdot a^{4\cdot b_2} \cdot \ldots \cdot a^{2\cdot n\cdot b_n} \mod m$ geschrieben werden. Eine Umformung ergibt $a^c \mod m = a^{b_0} \cdot (a^{b_1} \cdot (\ldots a^{b_{n-2}} \cdot (a^{b_{n-1}} \cdot (a^{b_n})^2)^2 \ldots)^2)^2 \mod m$, wobei die modulo-Division nach jedem Rechenschritt erfolgen darf. Es ergibt sich der folgende Algorithmus:

```
y = 1
for i = n to 0
    y = y^2 mod m
    if b[i] = 1 then y = (y * a) mod m
next i
```

Es werden nur so viele Schleifendurchläufe gebraucht wie c Binärstellen hat. Ohne diese Optimierung wären c modulo-Multiplikationen nötig.

33 Vlg. http://de.wikipedia.org/w/index.php?title=Schnelles_Potenzieren&oldid=57520746. 05.04.2009.

3.4.2 Optimierung der Entschlüsselung

Da der private Schlüssel d im Gegensatz zu e sehr groß ist, ist das Dechiffrieren zeitintensiver. Eine Verbesserung wird durch Aufspalten der modulo-Exponentiation mit dem *Chinesischen Restsatz* in zwei Berechnungen mit einem ca. halbsolagen Modulus erzielt.[34] Gesucht ist $m = c^d \mod n$. Als erstes wird d entsprechend p und q verkleinert: $d_p = d \mod p-1 \qquad d_q = d \mod q-1$

Denn es gilt z.B. $c^d \mod p = c^{d_p + j\cdot(p-1)} \mod p = c^{d_p}\cdot c^{j\cdot(p-1)} \mod p = c^{d_p}\cdot 1 \mod p$.

Anschließend wird $m_p = c^{d_p} \mod p$ und $m_q = c^{d_q} \mod q$ berechnet. m erhält man durch Lösen der Kongruenz nach 3.1.3:

$$h = m_q - m_p \qquad falls \;\; h<0 \;\; dann \;\; h = h+q$$

Das multiplikativ Inverse u von $p \;\; modulo\; q$ wird mit dem *Erweiterten Euklidischen Algorithmus* gesucht, es gilt $p\cdot u \mod q = 1$.

$$h = u\cdot h \mod q$$

$$m = m_p + h\cdot p \qquad also\; insgesamt \;\; m = m_p + (u\cdot(m_q - m_p) \mod q)\cdot p$$

Dieser Term entspricht der Lösung der simultanen Kongruenzen aus 3.1.3, bis auf das hinzugefügte $modulo\; q$, welches die Multiplikation mit p einfacher macht.

3.4.3 Hybride Verfahren

Da RSA um etwa den Faktor 1000 langsamer ist als die meisten symmetrischen Verfahren[35], werden hybride Verfahren eingesetzt. Der *OpenPGP*-Standard und viele andere verschlüsseln mit dem Public-Key nur den Sitzungsschlüssel des symmetrischen Verfahrens, durch welches die Nachricht verschlüsselt wird.

3.4.4 Das Forced-Latency-Interlock-Protokoll

Um die Möglichkeit eines *Man-in-the-middle-Angriffes* während des Schlüsseltausches ohne Schlüsselsignatur zu erschweren, entwickelten Ron Rivest und Adi Shamir das Interlock-Protokoll, welches von Bryce Wilcox-O'Hearn zum Forced-Latency-Interlock-Protokoll weiterentwickelt wurde.[36] Alice und Bob führen ihren Schlüsselaustausch wie gewohnt durch, es ist möglich, dass Mallory, der sich

34 Vgl. http://williamstallings.com/Extras/Security-Notes/lectures/publickey.html. 05.04.2009.
35 http://de.wikipedia.org/w/index.php?title=RSA-Kryptosystem&oldid=58136538. 06.04.2009.
36 Vgl. http://en.wikipedia.org/w/index.php?title=Interlock_protocol&oldid=244337345. 07.04.2009.

zwischengeschaltet hat, die öffentlichen Schlüssel durch seinen ersetzt. Wenn Alice an Bob eine Nachricht schickt, wird diese zwar mit dem erhaltenen Public-Key verschlüsselt, aber nur die erste Hälfte der Nachricht losgeschickt. Sie wartet nun eine für das Protokoll festgelegte Zeit, nach der Bob die erste Hälfte seiner verschlüsselten Nachricht schickt. Anschließend schickt sie nach erneuter Verzögerung den zweiten Teil ihrer Nachricht und muss wiederum warten, bis Bob nach der Verzögerungszeit seine Nachricht geschickt hat. Der Vorteil dieser Methode ist, dass halbe Nachrichten nicht von Mallory entschlüsselt werden können. Somit muss er entweder selbst den ersten Teil einer ausgedachten Nachricht an Alice schicken, um ihren zweiten Teil zu erhalten, mit welchem man die Nachricht entschlüsseln und an Bob weiterleiten kann, oder aber den ersten Teil der ausgedachten Nachricht an Bob schicken, um von ihm den ersten Teil für Alice zu erhalten. Der Betrug kann schnell aufgedeckt werden, weil Mallory schlecht voraussagen kann, welche Nachrichten geschickt werden sollen. Auch wenn Alice eine Information anfordert, merkt sie an der doppelten Verzögerungszeit, dass die Verbindung infiltriert wurde.

4 Fazit

Das RSA-Verfahren ist, aufgrund seiner Einfachheit im Vergleich zu anderen Public-Key-Verfahren, ein praktisches, vielseitiges und ungebrochenes Kryptosystem. Auch wenn die Tendenz zu Elgamal, welches schon von Anfang an patentfrei war, da ist und die Elliptische-Kurven-Varianten wegen ihren geringeren Anforderungen im SmartCart-Bereich aufblühen, ist RSA noch der wichtigste Algorithmus. Wer seinen Schlüssel genügend groß wählt, braucht in den nächsten Jahren keine Paranoia vor der Faktorisierung zu haben. Ich gehe davon aus, dass weiterhin das *spezielle Zahlenkörpersieb* verwendet wird, jedoch der Fortschritt in der Parallelisierung liegt.

Doch bahnen sich Neuerungen an, welche nicht auf Prozessoren setzen. So haben Forscher der IBM im Jahr 2001 die Zahl 15 mit dem effektiveren *Shor-Algorithmus* auf einem Quantencomputer mit sieben Qubits faktorisiert.[37] Aber auch das Gebiet der Biocomputer kann die Schlüssellänge der Public-Key-Kryptographie hochtreiben. Leonard Adleman hat 1994 ein Hamiltonkreisproblem und 2002 ein 3-

37 Vgl. http://de.wikipedia.org/w/index.php?title=Shor-Algorithmus&oldid=57453119. 07.04.2009.

SAT-Problem mit Hilfe von DNA gelöst.[38] Adi Shamir veröffentlichte 1999 eine Idee zu einem optisch-elektronischen Apparat namens TWINKLE, welcher ca. 5000 US-Dollar kosten würde und 512-Bit-Zahlen faktorisieren könnte.[39] 2003 wurde der TWINKLE-Plan um die TWIRL-Maschine erweitert (Kosten: „einige Millionen Euro“), mit welcher es möglich wäre, eine 1024-Bit-Zahl innerhalb eines Jahres zu zerlegen.[40] Bis heute sind diese Maschinen wahrscheinlich fiktiv geblieben.[41]

Doch neben den theoretischen Sicherheitsaspekten ist die Implementierung die größte Schwachstelle des RSA-Algorithmus. Paul Kocher hatte 1996 mit einem Seitenkanalangriff Erfolg, welcher die Zeit des Entschlüsselungsvorgangs analysierte, sodass er dann auf den privaten Schlüssel schließen konnte.[42] 1998 gelang es, den Sitzungsschlüssel eines SSL-Dienstes mit *PKCS1v1* durch eine Adaptive-chosen-ciphertext-attack zu berechnen.[43] In meinem beiliegenden beispielhaften Quelltext für die grundlegenden RSA-Klassen wurde, eben wegen dieser Komplexität der möglichen Angriffe, nur die für die Praxis nicht allein ausreichenden Verfahren umgesetzt. Die Berechnung von großen Ganzzahlen wurde mit der *GNU Multiple Precision Arithmetic Library* realisiert.

Im privaten Gebrauch der Public-Key-Verschlüsselung rate ich die Benutzung von GnuPG[44], einer unter der freien *GNU General Public License* liegenden Alternative zum kostenpflichtigen PGP, welches auch den OpenPGP-Standard benutzt. Diese Software bietet auch Privatmenschen, die ihren privaten Schlüssel durch ein gut gewähltes Passwort und mit zusätzlicher Zugriffssicherung schützen, die Sicherheit von Militärstandards. Daher ist es zu verstehen, dass die USA Exportbeschränkungen für starke Kryptoprodukte bis 2000 hatte, wenn der Schlüssel nicht bei einer Behörde hinterlegt war, und dass Kryptographie als Waffe gilt.

Die Beschäftigung mit der Mathematik des RSA-Algorithmus und dessen Optimierungen ist eine Erweiterung zu den üblichen Themen des Schulunterrichts.

38 Vgl. http://de.wikipedia.org/w/index.php?title=Leonard_Adleman&oldid=56795826. 07.04.2009.
39 Vgl. Schmeh, K.: Kryptografie. 3. erw. Aufl. Heidelberg: dpunkt.verlag 2007. S. 170f.
40 Ebd. S. 171.
41 Ebd. S. 171.
42 Ebd. S. 262ff.
43 Vgl. http://en.wikipedia.org/w/index.php?title=Adaptive_chosen-ciphertext_attack&oldid=263705886. 07.04.2009.
44 http://gnupg.org/index.de.html benutzbar in der Thunderbird-Erweiterung Enigmail http://enigmail.mozdev.org/

5 Literaturverzeichnis

Bartholomé, A., Rung, J. und Kern, H.: Zahlentheorie für Einsteiger. 6. Aufl. Wiesbaden: Vieweg+Teubner 2008. S. 103.

Bellare, M., Rogaway, P.: Optimal Asymmetric Encryption -- How to encrypt with RSA. Springer-Verlag. 1995. Auch unter: http://www-cse.ucsd.edu/users/mihir/papers/oae.pdf. 29.03.2009.

Beutelspacher, A.: Kryptologie. 5., überarb. Aufl. Wiesbaden: Vieweg 1996.

Beutelspacher, A., Schwenk, J. und Wolfenstetter, K.-D.: Moderne Verfahren der Kryptographie. 6., überarb. Aufl. Wiesbaden: Vieweg 2006.

Diffie, W.: The First Ten Years of Public-Key Cryptography. Proceedings of the IEEE 76,5. 1988. S. 560-577.

Diffie, W. und Hellman, M.E.: New Directions in Cryptography. IEEE Transactions on Information Theory, IT 22,6. 1976. S. 644-654.

Schmeh, K.: Kryptografie. 3. erw. Aufl. Heidelberg: dpunkt.verlag 2007.

Schneier, B.: Angewandte Kryptographie. Dt. Übers. München: Pearson Studium. 2006.

http://de.wikipedia.org/w/index.php?title=Alice_und_Bob&oldid=58127321. 07.04.2009.

http://de.wikipedia.org/w/index.php?title=Leonard_Adleman&oldid=56795826. 07.04.2009.

http://de.wikipedia.org/w/index.php?title=Rabin-Kryptosystem&oldid=55128885. 22.03.2009

http://de.wikipedia.org/w/index.php?title=RSA-Kryptosystem&oldid=58136538. 06.04.2009.

http://de.wikipedia.org/w/index.php?title=Satz_von_Euler&oldid=55334706. 03.04.2009.

http://de.wikipedia.org/w/index.php?title=Schnelles_Potenzieren&oldid=57520746. 05.04.2009.

http://de.wikipedia.org/w/index.php?title=Shor-Algorithmus&oldid=57453119. 07.04.2009.

http://en.wikipedia.org/w/index.php?title=Adaptive_chosen-ciphertext_attack&oldid=263705886. 07.04.2009.

http://en.wikipedia.org/w/index.php?title=Attacking_McEliece_by_finding_low-weight_codewords&oldid=277511445. 22.03.2009.

http://en.wikipedia.org/w/index.php?title=Interlock_protocol&oldid=244337345. 07.04.2009.

http://glossar.hs-augsburg.de/index.php?title=McEliece-Kryptosystem&oldid=8170. 22.03.2009.

RFC 3447 V.1.5. ftp://ftp.rsasecurity.com/pub/pkcs/ascii/pkcs-1.asc. 03.04.2009.

RFC 3447 V.2.1. http://tools.ietf.org/html/rfc3447. 03.04.2009.

http://rsa.com/rsalabs/node.asp?id=2879. 28.03.2009.

http://williamstallings.com/Extras/Security-Notes/lectures/publickey.html. 05.04.2009.

6 Anhang

6.1 Platzhalternamen in der Kryptographie[45]

In der Kryptographie werden bestimmte Rollenverteilungen und Eigenschaften der Protokollteilnehmer durch deren Namen verdeutlicht. Die Rollenverteilung „Alice und Bob“ trat erstmals 1978 durch Verwendung der RSA-Erfinder auf.

Alice	Sie initiiert das Protokoll und ist Hauptaugenmerk des Betrachters.
Bob	Er ist der zweite Protokollteilnehmer.
Eve	(von engl. eavesdropper, LauscherIn) Sie kann die Kommunikation abhören, aber nicht verändern.
Mallory	(von engl. malicious, heimtückisch) Er hat die Möglichkeit, die Kommunikation abzuhören und zu verändern.
Trent	(von engl. TRusted ENTity) Er ist eine vertrauenswürdige Instanz, seine Signaturen für z.B. öffentliche Schlüssel werden akzeptiert.

Es kommen auch weitere und abweichende Namen für komplexere Protokolle vor.

6.2 Beispiel

Es folgt eine beispielhafte RSA-Schlüsselerzeugung, Ver- und Entschlüsselung unter Verwendung unsicherer kleiner Zahlen.

Es werden die Primzahlen $p=89$ und $q=97$ gewählt, sodass $n = p \cdot q = 89 \cdot 97 = 8633$ ist.

$\varphi(n)=(p-1)\cdot(q-1)=88\cdot 96=8448$ wird bestimmt, um das multiplikativ Inverse d von $e \bmod \varphi(n)$ zu berechnen. Es wird $e=17$ gewählt, damit $ggT(e,\varphi(n))=1$ gilt.

Es wird d gesucht, für das gilt $e\cdot d \bmod \varphi(n)=1$ bzw. $e\cdot d=k\cdot\varphi(n)+1 \quad \Leftrightarrow$ $1= d\cdot e+(-k)\cdot\varphi(n) = ggT(e,\varphi(n))$. Und somit ist bei $ggT(a,b)=x\cdot a+y\cdot b$ der Faktor $x=d$. Es wird der Erweiterte Euklidische Algorithmus verwendet, um d zu bestimmen. Falls d negativ sein sollte, gilt $d=d+\varphi(n)$, denn das Erhöhen um $\varphi(n)$ bedeutet ein dennoch gültiges d, da nur das unwichtige k sich in der Gleichung ändern würde: $e\cdot(d+\varphi(n))=k\cdot\varphi(n)+1$

$$\Leftrightarrow \quad e\cdot d+e\cdot\varphi(n)=k\cdot\varphi(n)+1 \qquad |-e\cdot\varphi(n)$$

$$\Leftrightarrow \quad e\cdot d=k\cdot\varphi(n)-e\cdot\varphi(n)+1 \quad \Leftrightarrow \quad e\cdot d=(k-e)\cdot\varphi(n)+1$$

45 Vgl. http://de.wikipedia.org/w/index.php?title=Alice_und_Bob&oldid=58127321. 07.04.2009.

Der *Erweiterte Euklidische Algorithmus:*

$8448 = 496 \cdot 17 + 16 \quad \Leftrightarrow \quad 16 = 1 \cdot 8448 - 496 \cdot 17$

$17 = 1 \cdot 16 + 1 \quad \Leftrightarrow \quad 1 = 1 \cdot 17 - 1 \cdot 16 = 1 \cdot 17 - 1 \cdot (1 \cdot 8448 - 496 \cdot 17) = \underline{497} \cdot 17 - 1 \cdot 8448$

$16 = 16 \cdot \underline{1} + 0 \quad \Rightarrow \quad ggT(e, \varphi(n)) = 1$

Es ist $1 = 497 \cdot 17 - 1 \cdot 8448$, d.h. $d = 497$. Der Public-Key wird aus $n = 8633$ und $e = 17$ gebildet. Eine geheime Information $m = 1984$ wird verschlüsselt:

$$c = 1984^{17} \ mod \ 8633 = 3266$$

Die Entschlüsselung mit dem privaten Schlüssel unter Verwendung des *Chinesischen Restsatzes*, es wäre aber auch möglich, $m = c^d \ mod \, n$ zu berechnen:

$d_p = d \ mod \ \varphi(p) = 497 \ mod \, 88 = 57$ *d wird als Exponent entsprechend mod p verkleinert*

$d_q = d \ mod \ \varphi(q) = 497 \ mod \, 96 = 17$ *d wird als Exponent entsprechend mod q verkleinert*

$m_p = c^{d_p} \ mod \ p = (3266 \, mod \, 89)^{57} \ mod \, 89 = 62^{57} \ mod \, 89 = 26$

$m_q = c^{d_q} \ mod \ q = (3266 \, mod \, 97)^{17} \ mod \, 97 = 65^{17} \ mod \, 97 = 44$

$h_1 = m_q - m_p = 44 - 26 = 18$ $\quad h_1 > 0$, *also wird* $h_1 = h_1 + q$ *nicht benötigt*

Suche nach u, dem multiplikativ Inversen von p *modulo* q:

$ggT(p, q) \equiv u \cdot p + z \cdot q \ \ (mod \, q) \quad \Leftrightarrow \quad u \cdot p \ mod \, q = 1$

Der *Erweiterte Euklidische Algorithmus:*

$97 = 1 \cdot 89 + 8 \quad \Leftrightarrow \quad 8 = 1 \cdot 97 - 1 \cdot 89$

$89 = 11 \cdot 8 + 1 \quad \Leftrightarrow \quad 1 = 1 \cdot 89 - 11 \cdot 8 = 1 \cdot 89 - 11 \cdot (1 \cdot 97 - 1 \cdot 89) = \underline{12} \cdot 89 - 11 \cdot 97$

$11 = 11 \cdot \underline{1} + 0 \quad \Rightarrow \quad ggT = 1 \quad und \quad \underline{12} \cdot 89 - 11 \cdot 97 = 1, \ also \ u = 12$

$h_2 = u \cdot h_1 \, mod \, q = 12 \cdot 18 \, mod \, 97 = 22$

$m = m_p + h_2 \cdot p = 26 + 22 \cdot 89 = 1984$ Es wurde also korrekt entschlüsselt.

Die Zahlen p, q, u, d, d_p und d_q bilden den Private-Key, sodass u, d_p und d_q nicht immer neu berechnet werden müssen.

6.3 Quelltext für RSA-Klassen in C++

```
/*
 *      rsa.cpp
 *      RSA-Beispielimplementierung durch Benutzung der GNU Multiple Precision
 *      Arithmetic Library <gmplib.org>
 *      Copyright (C) 2009 Kai Lüke <kai-tobias@web.de> oder auf kailueke.de.vu
 *
 *      This program is free software; you can redistribute it and/or modify
 *      it under the terms of the GNU General Public License as published by
 *      the Free Software Foundation; either version 3 of the License, or
 *      (at your option) any later version.
 *
 *      This program is distributed in the hope that it will be useful,
 *      but WITHOUT ANY WARRANTY; without even the implied warranty of
 *      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *      GNU General Public License for more details.
 *
 *      You should have received a copy of the GNU General Public License
 *      along with this program.  If not, see <http://www.gnu.org/licenses/>.
 *
 *      Kompilieren: g++ -Wall -O3 -s -march=native -o rsa rsa.cpp" -lgmpxx -lgmp
 *      Die Angaben -Wall -O3 -s -march=native sind optional. Wichtig ist jedoch,
 *      dass die GMP Library verfügbar ist
 *      (mehr dazu auf http://gmplib.org/manual/Installing-GMP.html#Installing-GMP)
 *      Die Ubuntu Pakete: libgmpxx4ldbl libgmp3c2 libgmp3-dev
 *
 *      Das Programm ist enthält nur die RSA-Grundlagen und ist in dieser Form noch
 *      nicht zur Praktischen Anwendung geeignet. Es fehlen:
 *      - OAEP
 *      - ein Protokoll
 *      - Hybride Verschlüsselung mit AES, Twofish, Serpent o.ä.
 *      Das alles liefert der OpenPGP-Standard, welcher von GnuPG unterstützt wird,
 *      daher empfehle ich die Benutzung von GnuPG.
 */
#include <iostream>
#include <iomanip>
#include <fstream>
#include <gmp.h>
#include <gmpxx.h>
#include <cstdlib>
#include <ctime>
using namespace std;
// Der Erweiterte Euklidische Algorithmus
// gibt die Zahlen x, y und den ggT(a, b) für die
// Gleichnung ggT(a, b) = x·a + y·b zurück
mpz_class* erwEuklid(mpz_class a, mpz_class b){
       mpz_class x=0;
       mpz_class lastx=1;
       mpz_class y=1;
       mpz_class lasty=0;
       mpz_class temp;
       mpz_class quotient;
       while(b!=0){
              temp=b;
              quotient=a/b;
              b=a%b;
              a=temp;
              temp=x;
              x=lastx-quotient*x; // Entspricht der Umformung
              lastx=temp;
              temp=y;
              y=lasty-quotient*y; // und Einsetzung nach ggT=x·a+y·b
              lasty=temp;
       }
       mpz_class *ret=new mpz_class[3];
```

```
	ret[0]=lastx;
	ret[1]=lasty;
	ret[2]=a;
	return ret;
}
// Diese Klasse ist für Public-Key-Objekte da, die die Funktion encrypt beherrschen.
class PublicKey{
	public:
	PublicKey(mpz_class, mpz_class);
	~PublicKey();
	mpz_class getE(){return e;};
	mpz_class getN(){return n;};
	mpz_class encrypt(mpz_class);
	void printInfo();
	private:
	mpz_class e; // Öffentlicher Exponent
	mpz_class n; // Modulus
};
// Konstruktor, für die Erzeugung eines Objektes
PublicKey::PublicKey(mpz_class publicExponent, mpz_class modul){
		e=publicExponent;
		n=modul;
	}
PublicKey::~PublicKey(){
}
void PublicKey::printInfo(){
	cout << endl << "PublicKey: Modul n: " << n << endl
	 << "Öffentlicher Exponent e: " << e << endl;
}
// Verschlüsselung
mpz_class PublicKey::encrypt(mpz_class m){
	if(!(m<n)){
		cout << "Klartext zu groß!";
		return NULL;
	}
	mpz_t c;
	mpz_init(c);
	// c = m^e mod n
	mpz_powm(c, m.get_mpz_t(), e.get_mpz_t(), n.get_mpz_t());
	return *new mpz_class(c);
}
// Diese Klasse besitzt ein Public-Key-Objekt und beherrscht die Funktion decrypt
class PrivateKey{
	public:
	PrivateKey(mpz_class, mpz_class, mpz_class, mpz_class, mpz_class, mpz_class);
	~PrivateKey();
	PublicKey getPK(){return *p;}; // Gibt den Public-Key zurück
	mpz_class decrypt(mpz_class);
	void printInfo();
	private:
	PublicKey *p; // Zeiger auf das dazugehörige Public-Key-Objekt mit den
		// Informationen e und n
	mpz_class d; // eigentlicher privater Schlüssel
	mpz_class faktor_p; // die restlichen Informationen werden für eine
			// schnellere Entschlüsselung
	mpz_class faktor_q; // oder Signierung durch den Chinesischen Restsatz
			// benötigt
	mpz_class modinv_u; // Multiplikativ Inverses von p mod q
	mpz_t d1; // Speichert das Ergebnis, welches in decrypt benötigt wird als Typ
		// und nicht als Klasse (schneller)
	mpz_t d2; // Somit sind für das Entschlüsseln/Signieren diese Schritte nicht
		// mehr durchzuführen
};
// Der Konstruktor verlangt die genauen Zahlen und wird von KeyGenerator aufgerufen
PrivateKey::PrivateKey(mpz_class e, mpz_class n, mpz_class d_privat, mpz_class fp,
mpz_class fq, mpz_class u){
		p=new PublicKey(e, n);
		d=d_privat;
```

```
                faktor_p=fp;
                faktor_q=fq;
                modinv_u=u;
                mpz_class dt1=d%(faktor_p-1); // d kann bei der Verwendung als
                // Exponent mod p in dieser Art verkleinert werden
                mpz_class dt2=d%(faktor_q-1); // weil x^(dp+φ(p) mod p = x^dp · x^φ(p)
                                              // mod p = x^dp · 1 mod p
                mpz_init(d1); // mpz_t-Typen müssen initialisiert werden
                mpz_init(d2);
                mpz_set(d1, dt1.get_mpz_t()); // Somit werden die Zahlen direkt als
                mpz_set(d2, dt2.get_mpz_t()); // Typ und nicht als Klasse gespeichert
                                              // → eine spätere Umrechnung fällt weg
        }
// Destruktor
PrivateKey::~PrivateKey(){
        delete p; // Public-Key-Objekt im Speicher löschen.
}
void PrivateKey::printInfo(){
        p->printInfo();
        cout << endl << "Privater Schlüssel: Geheimer Exponent d: " << d << endl
         << "Faktor p: " << faktor_p << endl << "Faktor q: " << faktor_q << endl
         << "Multiplikativ Inverses u von p mod q: " << modinv_u << endl;
}
mpz_class PrivateKey::decrypt(mpz_class c){
        if(!(c<p->getN())){
                cout << "Geheimtext zu groß!"; // c muss kleiner als n sein
                return NULL;
        }
        mpz_t m1;
        mpz_init(m1);
        mpz_t m2;
        mpz_init(m2);
        // Lösung der Kongruenzen mit dem Chinesischen Restsatz
        // Es sind kleinere Berechnungen durchzuführen, was die
        // Geschwindigkeit erhöht
        mpz_powm(m1, c.get_mpz_t(), d1, faktor_p.get_mpz_t());
        // m1 = c^d1 mod p
        mpz_powm(m2, c.get_mpz_t(), d2, faktor_q.get_mpz_t());
        // m2  = c^d2 mod q
        mpz_class h=mpz_class(m2)-mpz_class(m1);
        if(h<0){
        h+=faktor_q; }
        h=(modinv_u * h) % faktor_q;
        mpz_class m=mpz_class(m1)+(h*faktor_p);
        return m;
}
// Für die Schlüsselpaarerzeugung zuständig
class KeyGenerator{
        public:
        PrivateKey generateKeyPair(unsigned int); // generiert ein Schlüsselpaar
        KeyGenerator();
        private:
        char *getRandom(int); // liefert eine Folge von Zufallsbytes
};
// der Pseudozufallsgenerator wird durch die aktuelle Zeit in ms initialisiert
KeyGenerator::KeyGenerator(){
        srand(time(NULL));
}
// Liefert ein Schlüsselpaar mit einem Modulus der Länge 'bit' Bit
PrivateKey KeyGenerator::generateKeyPair(unsigned int bit){
        if(bit%16!=0 && bit!=0){
                cout << "Die Bitlänge soll ein Vielfaches von 16 sein!";
                return PrivateKey(NULL, NULL, NULL, NULL, NULL, NULL);
        }
        mpz_class e("65537"); // Standardwert für den öffentlichen Exponenten
        // weil er nur zwei binäre Einsen enthält und somit schnell mit dem, in der
        // GMP-Lib eingebauten, Square-And-Multiply-Algorithmus exponentieren kann,
        // aber trotzdem groß ist.
```

```
        mpz_class q; // der größere Primfaktor q und
        mpz_class p; // der kleinere Primfaktor p
        mpz_class n; // der Zahl n
        unsigned int nbitsize=0;
        mpz_t vergleich;
        mpz_init(vergleich); // Für einen großen Abstand von p und q benötigt
do{
        mpz_t rand1; // Zufallszahl
        mpz_init2(rand1, bit/2); // der halben Länge des Modulus
        char *zufall1=getRandom(bit/16);
        // Füllen mit Zufallsbits
        for(unsigned int i=0; i<bit/16; i++){
               for(int n=7; n>=0; n--){
               if((zufall1[i] & (1 << n)) != 0 ){
                      mpz_setbit(rand1, i*8+n);
               }
               }
               zufall1[i]=i*7%256; // in Speicher überschreiben
        }
        mpz_setbit(rand1, bit/2-1); // erstes
        mpz_setbit(rand1, 0); // und letztes Bit auf 1 setzten → groß genug und
                              // ungerade
        delete(zufall1); // Speicher freigeben, delete überschreibt den Speicher
                         // nicht
        mpz_t rand2; // die zweite Zufallszahl
        mpz_init2(rand2, bit/2);
        char *zufall2=getRandom(bit/16);
        // wird mit Zufallsbits gefüllt
        for(unsigned int i=0; i<bit/16; i++){
               for(int n=7; n>=0; n--){
               if((zufall2[i] & (1 << n)) != 0 ){
                      mpz_setbit(rand2, i*8+n);
               }
               }
               zufall2[i]=i*7%256; // in Speicher überschreiben
        }
        mpz_setbit(rand2, bit/2-1); // erstes und
        mpz_setbit(rand2, 0); // letztes Bit auf 1 setzen
        delete(zufall2); // Speicher freigeben
do{
        while (!mpz_probab_prime_p(rand1, 50)){ // Miller-Rabin und andere
               mpz_add_ui(rand1, rand1, 2); // Verfahren testen, ob es eine Primzahl
                                            // ist
        }                                   // falls nicht, wird sie um 2 erhöht (bleibt
                                            // ungerade)
        p=mpz_class (rand1); // Die erste Primzahl p wurde gefunden

        while (!mpz_probab_prime_p(rand2, 50)){ // 50 bedeutet die Anzahl der Tests
               mpz_add_ui(rand2, rand2, 2);
        }
        q=mpz_class (rand2); // Primzahl q wurde gefunden
        if(p>q){ // p soll kleiner sein als q → Tausch falls p>q
               mpz_class t=p;
               p=q;
               q=t;
        }
        n=p*q; // der Modulus n wird berechnet
        nbitsize=mpz_sizeinbase(n.get_mpz_t(),2); // die Länge wird abgefragt
        if(nbitsize<bit){ // und auf die richtige Länge überprüft
               cout << nbitsize << " ist die falsche Länge, erneuter Durchlauf" <<
               endl;
               // im nächsten Durchlauf werden die Zufallszahlen erhöht
               // indem die eins der ersten Bits auf 1 gesetzt wird
               if(rand()%2==0) // für eine der beiden Primzahlen
               mpz_setbit(rand1, bit/2-2-rand()%5);
               else
               mpz_setbit(rand2, bit/2-2-rand()%5);
        }
```

```
}while(nbitsize<bit); // Es wird weiter nach geeigneten Primfaktoren für n gesucht

	cout << endl << "Größe von N: " << nbitsize << " Bit" << endl;
	// wenn p-q kleiner ist als 2·n^¼, ist es einfach p und q zu bestimmen
	mpz_class temporaer((q-p)/2);
	mpz_pow_ui(vergleich, temporaer.get_mpz_t(), 4);
	if(mpz_class(vergleich)<n){
		cout << "p und q zu nah aneinander";
	}
//wenn p und q zu nah aneinander liegen weiter suchen
}while(mpz_class(vergleich)<n);
	// p und q sind gefunden, φ(n) wird bestimmt
	mpz_class phi=(p-1)*(q-1);
	// falls e und phi nicht teilerfremd sind, wird e erhöht
	while(erwEuklid(e, phi)[2]!=1){
		e+=2;
		cout << endl << "e nicht teilerfremd zu phi!" << endl;
	}
	mpz_class d=erwEuklid(e, phi)[0] % phi; // Addieren und Subtrahieren ergibt
	                                        // gültiges d:
	if(d<0){
	d+=phi; } // Alle Zahlen sollen positiv sein.
	// Erhöhen um φ(n) bedeutet dennoch gültiges d, da nur das unwichtige k sich
	// in der Gleichung ändern würde:
	// e·(d+φ(n))=k·φ(n)+1
	// <=> e·d+e·φ(n)=k·φ(n)+1 |-e·φ(n)
	// <=> e·d=k·φ(n)-e·φ(n)+1
	// <=> e·d=(k-e)·φ(n)+1
	mpz_class u=erwEuklid(p, q)[0] % q; // u wird als mult. Inverses zu p modulo
	                                    // q gesucht
	if(u<0){
	u+=q; } // Alle Zahlen sollen positiv sein
	// das Schlüsselpaar wurde erzeugt
	return *new PrivateKey(e, n, d, p, q, u);
}
// eine Zufallsfolge von Bytes der Länge 'len' wird erzeugt
char *KeyGenerator::getRandom(int len){
	char *zufallsfolge=new char[len];
	ifstream guterzufall("/dev/random"); // das Unix-Gerät für hardwareabhängigen
	                                     // guten Zufall
	ifstream nichtblockenderzufall("/dev/urandom"); // und das Gerät für
	                                  // schnellen Zufall wird angefordert
	if(guterzufall.is_open()){ // Geraete ist verfügbar
	 cout << endl << "Generiere Zufall: Bewegen Sie die Maus, tippen Sie,"
"arbeiten Sie mit dem PC und benutzen Sie die Festplatte." << endl;
	 if(!nichtblockenderzufall.is_open()){
		cout << "/dev/urandom ist nicht verfügbar, Zeitabhängiger Pseudo-"
"Zufall wird teilweise hinzugezogen." << endl;
		}
		for(int i=0; i<len; i++){
			if(i%8==0){ // Jedes 8. Byte ist echter, blockender Zufall
			 cout << "*" << flush;
			 guterzufall.get(zufallsfolge[i]);
			}else{ // der Rest ist Pseudozufall
				if(nichtblockenderzufall.is_open()){ // entweder gut
				nichtblockenderzufall.get(zufallsfolge[i]);
				}else{ // oder weniger gut, wie dieser zeitabhängige
					// Zufall
				zufallsfolge[i]=char(256.0*rand()/(RAND_MAX + 1.0));
				}
			}
		}
		guterzufall.close();
		nichtblockenderzufall.close();
		cout << endl;
	}else{ // es sind keine Zufallsgeräte verfügbar, es wird ein zeitabhängiger
		// Pseudozufall verwendet
		cout << endl << "/dev/random ist nicht verfügbar, Zeitabhängiger"
```

```
"Pseudo-Zufall muss benutzt werden." << endl;
            for(int i=0; i<len; i++){
                  zufallsfolge[i]=char(256.0*rand()/(RAND_MAX + 1.0));
            }
      }
      return zufallsfolge;
}

int main(int argc, char** argv)
{
      KeyGenerator k;
      PrivateKey geheimschluessel=k.generateKeyPair(1024);
      PublicKey oeffentlich=geheimschluessel.getPK();
      geheimschluessel.printInfo();
      mpz_class nachricht("98765432100123456789");
      cout << endl << "Klartext: " << nachricht << endl;
      mpz_class geheimtext=oeffentlich.encrypt(nachricht);
      cout << "Geheimtext: " << geheimtext << endl;
      cout << endl << "Entschlüsselter Klartext: " <<
geheimschluessel.decrypt(geheimtext) << endl;
      return 0;
}
```

Erklärung zur selbstständigen Anfertigung der Facharbeit

Ich versichere, dass ich die vorliegende Facharbeit selbstständig verfasst, alle aus anderen Werken wörtlich oder sinngemäß entnommenen Stellen und Abbildungen unter Angabe der Quelle als Entlehnung kenntlich gemacht und andere als die angegebenen Hilfsmittel nicht benutzt habe.

_____, den _____ _____

(Ort) (Datum) (Unterschrift)