Gymnasium Ramstein-Miesenbach **2005/2006**

Facharbeit

aus dem Fach

Mathematik

Thema: Eine mathematische Betrachtung des RSA-Verfahrens

Verfasser:	Christopher Tim Althoff
Leistungskurs:	Mathematik
Kursleiter:	Herr Leyes
Abgabetermin:	15. Mai 2006

Erzielte Note: _____ In Worten: _____

Erzielte Punkte: _____ In Worten: _____
(einfache Wertung)

Abgabe im Sekreteriat: _____

Unterschrift des Kursleiters: _____

Inhaltsverzeichnis

1 Einleitung

Seit über zehn Jahren gibt es mittlerweile das sogenannte World-Wide-Web (WWW), ein riesiges multimediales Informationssytem, das sich auf das sogenannte Internet stützt, in dem Hunderte von Millionen von Rechnern vernetzt sind. Immer mehr Menschen nutzen die Dienstleistungen, die über das WWW angeboten werden. Viele dieser Dienstleistungen müssen bezahlt werden oder erfordern die Übermittlung sonstiger vertraulicher Informationen. Um den erforderlichen Datenschutz zu gewährleisten bedient man sich sehr leistungsfähiger Verschlüsselungsverfahren, die vor 50 Jahren nur in staatlichen Militärapperaten zu finden waren. Neben dem immer größer werdenden Einfluss im täglichen Leben wird regelmäßig sehr viel Geld investiert, um noch größere Primzahlen zu generieren oder noch größere Zahlen faktorisieren zu können. So mag man sich fragen, was denn dahintersteckt.
Diese Facharbeit beschäftigt sich mit einem solchen Verfahren, dem sogenannten RSA-Verfahren, welches aus einer mathematischen Perspektive betrachtet wird, um damit eine Antwort auf die oben gestellte Frage zu geben.

Das RSA-Verfahren wurde 1977 entwickelt und ist nach seinen geistigen Vätern Rivest, Shamir und Adleman benannt. RSA spielt ihn der Krypographie eine ganz besondere Rolle, da der Algorithmus als erster eine praktische Umsetzung der Public-Key-Kryptographie ermöglichte. Die Theorie der asymmetrischen Verschlüsselung (Public Key Kryptographie) war zuvor von Whitfield Diffie und Martin Hellman veröffentlicht worden (vgl. [5]). Die besondere Stellung von RSA resultiert dabei aus der Lösung des Schlüsselverteilungsproblems (notwendige Übertragung eines Sitzungsschlüssels auf sicherem Weg), welches man bis in die 70er Jahre für prinzipiell unlösbar hielt.

In dieser Arbeit möchte ich den Algorithmus vorstellen und mathematisch betrachten. Um RSA nachvollziehen zu können, gibt das zweite Kapitel einen Überblick über die prinzipiellen Grundlagen der asymmetrischen Verschlüsselung. Außerdem wird auf digitale Signaturen und deren Anwendung in der Praxis eingegangen.
Das zentrale dritte Kapitel behandelt die zahlentheoretischen Grundlagen, die für das Verständnis von RSA notwendig sind. Dazu werden zunächst Sätze zur Teilbarkeit formuliert und die Kongruenzrechnung vorgestellt. Des Weiteren werden modulare Inverse und der Satz von Euler-Fermat betrachtet, welche den Schlüssel zum Beweis des Verfahrens liefern.
Im vierten Kapitel wird der RSA-Algorithmus erläutert und an einem Beispiel verdeutlicht. Danach wird die Gültigkeit des Verfahrens auf Grundlage des dritten Kapitels bewiesen und mein Programm vorgestellt, welches den RSA-Algorithmus

praktisch umsetzt.
Im fünften Kapitel wird ein Fazit gezogen und speziell die Aussichten von RSA in der Zukunft erörtert.

Eine genauere Betrachtung anderer asymmetrischer Verfahren, Hashfunktionen, Faktorisierung oder diskreten Logarithmen würde den Rahmen dieser Arbeit sprengen. Ebenso können bekannte Attacken auf RSA nicht näher analysiert oder die immense wirtschaftliche und politische Bedeutung von RSA diskutiert werden. Der interessierte Leser sei hier auf weiterführende Literatur (siehe Quellen) verwiesen. [1], [2] und [3] behandeln die Kryptogryphie aus informationstechnischer Sicht. [4] und [6] vermitteln die benötigten mathematischen Grundlagen. Der geschichtliche Hintergrund wird in [5] dargestellt.

2 Asymmetrische Verschlüsselung

Bei symmetrischen Verfahren wie AES, DES oder IDEA (vgl. [1]) verwendet man einen Schlüssel, mit dem die Nachricht sowohl chiffriert, als auch dechiffriert werden kann. Die Verfahren zur Ver- und Entschlüsselung können jedoch unterschiedlich sein, d.h. die Symmetrie bezieht sich hier auf die Schlüssel, nicht auf das Verfahren. Für symmetrische Verfahren gilt jedoch das Problem der Schlüsselverteilung, welches man bis in die 70er Jahre für prinzipiell unlösbar hielt. Das Problem liegt darin, dass vor dem eigentlichen Nachrichtentransfer erst ein geheimer Sitzungsschlüssel auf sicherem (!) Weg übermittelt werden muss. Andernfalls hätte praktisch jeder die Möglichkeit die Nachrichten zu entschlüsseln. Dieses Problem wird durch die asymmetrischen Verfahren elegant gelöst.

2.1 Prinzip der Asymmetrischen Verschlüsselung

Asymmetrische Verschlüsselungsverfahren (z.B. RSA, Rabin, Elgamal) verwenden zwei unterschiedliche Schlüssel, von denen einer öffentlich („public key“) und einer geheim („private key“) ist (vgl. [1]). Der öffentliche Schlüssel e („encrypt“) wird dazu benutzt, eine verschlüsselte Nachricht an seinen Eigentümer zu senden. Der private Schlüssel d („decrypt“) ermöglicht es dem Eigentümer nun die erhaltene Nachricht wieder zu entschlüsseln.
Will z.B. Alice eine Nachricht m („message“) an Bob senden, ermittelt sie zuerst Bobs öffentlichen Schlüssel e_{Bob}, chiffriert damit ihre Nachricht und sendet sie an Bob. Sie muss also keinen Sitzungsschlüssel auf sicherem Weg übertragen, sondern kann den für jedermann zugänglichen öffentlichen Schlüssel verwenden. Nun kann Bob ihre verschlüsselte Nachricht c („ciphertext“) mit seinem privaten Schlüssel d_{Bob} dechiffrieren (und sonst niemand).
Es darf hierbei nicht möglich sein, den privaten aus dem öffentlichen Schlüssel zu gewinnen oder die Nachricht aus dem Geheimtext. „Nicht“ ist hier im kryptologischen Sinne zu verstehen, d.h. nicht mit verfügbaren Algorithmen und vertretbarem Aufwand an Geld und Zeit (vgl. [2]).

Die asymmetrischen Verfahren basieren auf sogenannten Einwegfunktionen („one-way function“), welche leicht zu berechnen aber sehr schwer umzukehren sind. Falltürfunktionen („trapdoor function“) sind spezielle Einwegfunktionen, die sich mit Hilfe von Zusatzinformationen leicht umkehren lassen. Allgemein beruhen diese Funktionen auf mathematisch harten Problemen, wie z.B. der Primfaktorzerlegung oder diskreten Logarithmen (Umkehrung der modularen Exponentation). Von diesen Problemen glaubt man, dass sie nur mit sehr großem Aufwand gelöst werden können, was bisher jedoch nicht bewiesen werden konnte (vgl. [1]).

Ein weiterer Vorteil asymmetrischer Verfahren ist die geringere Schlüsselanzahl. Jeder Teilnehmer braucht nur noch ein Schlüsselpaar, anstatt einen Sitzungsschlüssel für den Kontakt mit jedem weiteren Teilnehmer. Bei n Teilnehmern benötigt man also nur $2n$ Schlüssel für asymmetrische Verfahren, wohingegen man $\frac{n(n-1)}{2}$ Schlüssel für symmetrische Varfahren bräuchte (vgl. [3]).
Ein wesentlicher Nachteil liegt in der Notwendigkeit des Schlüsselmanagments. Gibt sich eine Person für eine andere aus, so wird es ihr ermöglicht alle Nachrichten für diese Person abzufangen, da man nicht überprüfen kann, ob es sicher bei dieser Person um die gewünschte handelt. Es existieren zwei populäre Lösungskonzepte. Beim ersten verwaltet ein Zentralrechner die öffentlichen Schlüssel vertrauenswürdiger Personen. Beim zweiten Konzept, dem sogenannten „web of trust“ zertifizieren sich die Anwender gegenseitig nach dem Motto „der Freund meines Freundes ist auch mein Freund“ (vgl. [2]).

Die größten Kritikpunkte sind die geringe Geschwindigkeit bei großen Datenmengen und die Anfälligkeit gegenüber Attacken (v.a. „chosen ciphertext attack“; vgl. [1] und [2]). Eine Lösung bilden die so genannten hybriden Systeme, welche die Vorteile von symmetrischen und asymmetrischen Verfahren kombinieren, aber nicht alle Schwächen aufweisen. Bei hybriden Systemen wird ein zufalliger Sitzungsschlüssel erzeugt und mit dem öffentlichen Schlüssel des Empfängers per asymmetrischem Verfahren verschlüsselt. Die eigentliche Nachricht wird mit dem Sitzungsschlüssel per symmetrischem Verfahren chiffriert. Beides senden wir dem Empfänger, der nun als einziger mit seinem privaten Schlüssel den Sitzungsschlüssel ermitteln und die Nachricht entziffern kann (vgl. [2]).

2.2 Digitale Signaturen

Viele asymmetrische Algorithmen (z.B. RSA) bieten die Möglichkeit digitale Signaturen zu erzeugen. Sie ist praktisch das elektronische Äquivalent zur konventionellen Unterschrift. Es muss sichergestellt sein, dass nur eine einzige Person diese Unterschrift produzieren, aber jeder andere verifizieren kann, dass diese Unterschrift von der besagten Person stammt (vgl. [3]).

Das Prinzip ist simpel: Alice chiffriert das Dokument mit ihrem privaten Schlüssel und sendet das somit unterzeichnete Dokument an Bob. Dieser dechiffriert das Dokument mit Alices öffentlichem Schlüssel. Ergibt sich so ein sinnvolles Dokument, ist die Echtheit der Unterschrift überprüft, da nur Alice den benötigten privaten Schlüssel besitzt.

Diese Möglichkeit digitaler Signaturen weist jedoch einige Nachteile auf: Durch die Verwendung eines asymmetrischen Verfahrens (nur umgekehrt) ist diese Signatur vor möglichen Attacken nicht immer ausreichend geschützt und zudem noch sehr aufwendig zu produzieren. Eine bekannte Attacke ist die „chosen-ciphertext attack“, bei der man durch unachtsames Signieren von Texten einer bestimmten Struktur dem Angreifer ermöglicht andere Nachrichten zu entschlüsseln (vgl. [1] und [2]). Des Weiteren ist das ursprüngliche Dokument nicht lesbar, was dieses Verfahren unpraktikabel erscheinen lässt.

Dieses Problem kann mit den sogenannten Hashfunktionen (vgl. [1] und [2]), die auch tatsächlich im Alltag verwendet werden, gelöst werden. Hashfunktionen berechnen z.B. aus einem Dokument eine (deutlich kleinere) Prüfsumme, mit welcher man ein Dokument identifizieren kann.
Der Trick liegt nun darin, dass man nicht das Dokument selbst, sondern nur dessen Hashwert signiert. Dies ist mit wesentlich weniger Aufwand zu realisieren, da Hashwerte deutlich kleiner sind als die Dokument selbst. Außerdem kann das eigentliche Dokument so für den Empfänger direkt lesbar mitgesendet werden.
Will man nun die Echtheit von Alices Unterschrift überprüfen, verschlüsselt man den signierten Hashwert mit ihrem öffentlichen Schlüssel. Dann bildet man selbst den Hashwert des mitgesendeten Dokuments und vergleicht beide Werte. Sind sie gleich ist die Unterschrift gültig.

3 Zahlentheoretische Grundlagen

Im Folgenden werden die Zahlentheoretischen Grundlagen vorgestellt, welche für das mathematische Verständnis des RSA-Algorithmus unabdingbar sind.
Zunächst werden wesentliche Sätze zur Teilbarkeit betrachtet, die später bei der Kongruenzrechnung wichtig sind. Außerdem werden Möglichkeiten aufgezeigt den größten gemeinsamen Teiler oder das multiplikative Inverse modulo m zu berechnen, welche wir später zur Schlüsselerzeugung nutzen werden. Weiter wird der Satz von Euler-Fermat behandelt, der den wesentlichen Beitrag zur Gültigkeit des RSA-Verfahrens liefert.

3.1 Teilbarkeit und Primzahlen

Definition: Teilbarkeit
Seien a und b ganze Zahlen. Die Zahl a teilt b genau dann, wenn es eine ganze Zahl k gibt, sodass gilt:

$$b = k \cdot a$$

Hinweis: Man schreib $a|b$.

Es gelten folgende Sätze zur Teilbarkeit (vgl. [7])

Satz 1
Seien a, b, c ganze Zahlen, so gilt:

1. $a|0$
2. $a|b \Leftrightarrow (a \cdot c)|(b \cdot c)$ mit $c \neq 0$
3. $a|b \Rightarrow a|(c \cdot b)$
4. $(a \cdot b)|c \Rightarrow a|c \wedge b|c$
5. $a|b \wedge a|c \Rightarrow a|(b \pm c)$
6. $a|(b \pm c) \wedge a|b \Rightarrow a|c$
7. $a|b \wedge b \neq 0 \Rightarrow |a| \leq |b|$
8. $a|b \wedge b|a \Rightarrow |a| = |b|$
9. $c|b \wedge b|a \Rightarrow c|a$
10. $b_1|a_1 \wedge b_2|a_2 \Rightarrow (b_1 \cdot b_2)|(a_1 \cdot a_2)$

Beweis:

1. $0 = k \cdot a$ mit $k = 0 \Rightarrow a|0$, da $k \in \mathbb{Z}$
2. $a|b \Leftrightarrow b = a \cdot k \Leftrightarrow b \cdot c = (a \cdot c) \cdot k$ mit $k \in \mathbb{Z}$
3. $b = a \cdot k \Leftrightarrow b \cdot c = (c \cdot k) \cdot a$ mit $(c \cdot k) \in \mathbb{Z}$
4. $c = a \cdot b \cdot k = a \cdot (b \cdot k) = b \cdot (a \cdot k)$ mit $(a \cdot k), (b \cdot k) \in \mathbb{Z}$
5. $b = a \cdot k_1$, $c = a \cdot k_2 \Longrightarrow b \pm c = (a \cdot k_1 \pm a \cdot k_2) = a \cdot (k_1 \pm k_2)$ mit $(k_1 \pm k_2) \in \mathbb{Z}$
6. $b \pm c = a \cdot k_1$, $b = a \cdot k_2 \Longrightarrow a \cdot k_2 \pm c = a \cdot k_1$
   $\Leftrightarrow c = \pm a(k_1 - k_2)$ mit $k_1, k_2 \in \mathbb{Z}$
7. $b = a \cdot k \Rightarrow |b| = |a| \cdot |k|$
   $b \neq 0 \Rightarrow a \neq 0 \wedge k \neq 0$
   Aus $k \in \mathbb{Z}$ und $k \neq 0$ folgt $|k| \geq 1 \Rightarrow |a| \leq |b|$
8. $b|a \Rightarrow |b| \leq |a|$, $a|b \Rightarrow |a| \leq |b| \Rightarrow |a| = |b|$ (vgl. Satz 1.7)
9. $a = k_1 \cdot b$, $b = k_2 \cdot c \Longrightarrow a = (k_1 \cdot k_2)c$ mit $(k_1 \cdot k_2) \in \mathbb{Z}$
10. $a_1 = k_1 \cdot b_1$, $a_2 = k_2 \cdot b_2 \Longrightarrow a_1 \cdot a_2 = (k_1 \cdot k_2)(b_1 \cdot b_2)$ mit $(k_1 \cdot k_2) \in \mathbb{Z}$

Definition: Primzahl
Sei $p \in \mathbb{N}$. Hat p genau zwei verschiedene positive Teiler, nämlich 1 und p, so nennt man p eine Primzahl.

3.2 Größter gemeinsamer Teiler und (erweiterter) Euklidischer Algorithmus

Aus Satz 1.7 folgt, dass jede Zahl nur endlich viele Teiler besitzt. Folglich gibt es auch nur endlich viele gemeinsame Teiler zweier Zahlen. Es muss also auch einen größten gemeinsamen Teiler geben (vgl. [4]). Dieser wird später von Bedeutung sein. Er ist wie folgt definiert:

Definition: größter gemeinsamer Teiler
Seien x, y, d ganze Zahlen. d heißt genau dann größter gemeinsamer Teiler von x und y, wenn folgende zwei Bedingeungen erfüllt sind:

1. d ist gemeinsamer Teiler von x und y, d.h. $d|x$ und $d|y$
2. Für jeden weiteren Teiler d' von x und y gilt $d'|d$

Man schreibt: $d = ggT(x, y)$

Alternative Definition:
Sei T die Menge aller Teiler von a und b, so gilt: T ist nichtleer ($1 \in T$) und nach oben durch $max(|a|, |b|)$ beschränkt (vgl. Satz 1.7).
Man definiert: $ggT(a, b) := max(T)$

Anmerkung:
Gilt $ggT(x, y) = 1$, so sagt man auch x und y sind teilerfremd oder zueinander relativ prim.

Um den größten gemeinsamen Teiler zweier Zahlen zu bestimmen, muss man nicht zwingend beide in ihre Primfaktoren zerlegen (Faktorisierung sehr zeitaufwändig!). Bereits vor über 2000 Jahren gab Euklid einen Algorithmus an, mit dessen Hilfe man den größten gemeinsamen Teiler zweier Zahlen rekursuv berechnen kann.
Anmerkung: Der Euklidische Algorithmus ist gleichzeitig ein (konstruktiver) Beweis für die Existenz des größten gemeinsamen Teilers (vgl. [3] und [4]).

Schema zur Berechnung des $ggT(a, b)$:
Wir setzen $r_0 = a$ und $r_1 = b$ und führen nach folgendem Schema (eindeutige) Teilungen mit Rest (vgl. [4]) sukzessiv durch, bis sich der Rest 0 ergibt.

$$\begin{array}{ll} r_0 = q_1 \cdot r_1 + r_2 & 0 < r_2 < r_1 \\ r_1 = q_2 \cdot r_2 + r_3 & 0 < r_3 < r_2 \\ \vdots & \vdots \\ r_{n-2} = q_{n-1} \cdot r_{n-1} + r_n & 0 < r_n < r_{n-1} \\ r_{n-1} = q_n \cdot r_n + 0 & \end{array}$$

Da die Reihe der Zahlen $r_1, r_2, \ldots$ streng monoton abnimmt, endet der Algorithmus nach endlich vielen Schritten.

Behauptung: $r_n = ggT(a, b)$
Nach der Defintion des größten gemeinsamen Teilers ist zweilerlei zu zeigen:

1. r_n ist gemeinsamer Teiler von a und b
2. Jeder weitere gemeinsame Teiler d von a und b ist auch ein Teiler von r_n

Beweis:
zu 1.
Wir müssen das Gleichungssystem von unten nach oben durchlaufen.
Aus der letzen Gleichung folgt r_n teilt r_{n-1}. Also teilt r_n auch $q_{n-1} \cdot r_{n-1} + r_n$ und damit r_{n-2} (vgl. Satz 1.5). So fortfahrend erhält man, dass r_n alle r_i $(i = n, n-1, \ldots, 1, 0)$

teilt. Insbesondere ist r_n ein Teiler von $r_0 = a$ und $r_1 = b$.

zu 2.
Hierzu betrachten wir die Gleichungen von der ersten bis zur letzten. Aus der ersten Gleichung folgt, dass der gemeinsame Teiler d auch r_2 teilt (vgl. Satz 1.6). Da d somit ein gemeinsamer Teiler von r_1 und r_2 ist, folgt aus der zweiten Gleichung, dass d auch ein Teiler von r_3 ist. Schließlich erhält man, dass d ein Teiler von r_n ist.

Beispiel 1:
Wir wollen den ggT(293,84) berechnen.
Wir setzen $r_0 = 293$ und $r_1 = 84$.

$$\begin{aligned} 293 &= 3 \cdot 84 + 41 \\ 84 &= 2 \cdot 41 + 2 \\ 41 &= 20 \cdot 2 + 1 \\ 2 &= 2 \cdot 1 + 0 \end{aligned}$$

Der größte gemeinsame Teiler von 283 und 84 ist also 1.

Lemma von Bézout
Seien a, b ganze Zahlen und $d = ggT(a, b)$ ihr größter gemeinsamer Teiler. Dann gibt es ganze Zahlen x, y, sodass gilt:

$$d = x \cdot a + y \cdot b$$

Hinweis: Eine solche Darstellung nennt man Vielfachsummendarstellung des $ggT(a, b)$.

Um dies zu beweisen und gleichzeitig eine Möglichkeit anzugeben die Zahlen x und y zu berechnen (konstruktiver Beweis), müssen wir die Euklidische Gleichungskette, wenn auch leicht umgeformt, ein weiteres Mal betrachten. Man nennt dies den erweiterten Euklidischen Algorithmus (vgl. [3]).

$$\begin{aligned} r_n &= r_{n-2} - q_{n-1} \cdot r_{n-1} \\ r_{n-1} &= r_{n-3} - q_{n-2} \cdot r_{n-2} \\ r_{n-2} &= r_{n-4} - q_{n-3} \cdot r_{n-3} \\ &\vdots \end{aligned}$$

Durch sukzessive Elimination von r_k $(k = n-1, n-2, \ldots, 2)$ ergibt sich:

$$\begin{aligned}
r_n &= r_{n-2} - q_{n-1} \cdot r_{n-1} \\
&= r_{n-2} + (-q_{n-1}) \cdot r_{n-1} \\
&= r_{n-2} - q_{n-1}(r_{n-3} - q_{n-2} \cdot r_{n-2}) \\
&= (-q_{n-1}) \cdot r_{n-3} + (1 + q_{n-2} \cdot q_{n-1}) \cdot r_{n-2} \\
&= (1 + q_{n-2} \cdot q_{n-1})(r_{n-4} - q_{n-3} \cdot r_{n-3}) - q_{n-1} \cdot r_{n-3} \\
&= [\ldots] \cdot r_{n-4} + [\ldots] \cdot r_{n-3} \\
&= \ldots \\
&= x \cdot r_0 + y \cdot r_1 \\
&= x \cdot a + y \cdot b
\end{aligned}$$

Die Ausdrücke werden immer komplexer, jedoch handelt es sich bei jeder Klammer um eine Zahl.

Beispiel 2:
Wir wollen Zahlen x und y berechnen, sodass gilt:
$ggT(293, 84) = 1 = x \cdot 293 + y \cdot 84$ (vgl. Beispiel 1)

$$\begin{aligned}
1 &= 41 - 20 \cdot 2 \\
&= 41 - 20(84 - 2 \cdot 41) = 41 \cdot 41 - 20 \cdot 84 \\
&= 41(293 - 3 \cdot 84) - 20 \cdot 84 \\
&= 41 \cdot 293 - 143 \cdot 84
\end{aligned}$$

Mit dem erweiterten Euklidischen Algorithmus berechnen wir $x = 41$ und $y = -143$.

3.3 Kongruenzen und Restklassen

Definition: Restklassen modulo m (vgl. [7])
Sei $m \in \mathbb{N}$. Die Mengen $[0]_m, [1]_m, \ldots, [m-1]_m$ seien folgendermaßen definiert:

$[0]_m = \{x | x = k \cdot m + 0, \forall k \in \mathbb{Z}\}$
$[1]_m = \{x | x = k \cdot m + 1, \forall k \in \mathbb{Z}\}$
$\vdots$
$[m-1]_m = \{x | x = k \cdot m + (m-1), \forall k \in \mathbb{Z}\}$

also allgemein:
$[i]_m = \{x | x = k \cdot m + i, \forall k \in \mathbb{Z}\}$

Man nennt diese Mengen die **Restklassen modulo** m.

Beispiel 3 für $m = 4$:
$[0]_4 = \{\ldots, -12, -8, -4, 0, 4, 8, 12, \ldots\}$
$[1]_4 = \{\ldots, -11, -7, -3, 1, 5, 9, 13, \ldots\}$
$[2]_4 = \{\ldots, -10, -6, -2, 2, 6, 10, 14, \ldots\}$
$[3]_4 = \{\ldots, -9, -5, -1, 3, 7, 11, 15, \ldots\}$

Man bezeichnet diese Mengen als Restklassen modulo m, weil der Rest i, den x bei ganzzahliger Division (vgl. [4]) durch den Modul m lässt, für die Zugehörigkeit zur jeweiligen Restklasse entscheident ist. Offensichtlich kann man (laut Definition) den Modul m beliebig oft addieren oder subtrahieren ohne die Restklasse modulo m zu verlassen. Man nennt i den **Repräsentanten** der jeweiligen Restklasse.

Definition: Kongruenz
Seien $a, b \in \mathbb{Z}$ und $m \in \mathbb{N}$. Die ganzen Zahlen a und b sind genau dann kongruent modulo m, wenn sie Elemente derselben Restklasse modulo m sind.
Man schreibt:

$$a \equiv b \bmod m$$

Anmerkung: Auch folgende Definition von Kongruenz gilt:
Seien $a, b \in \mathbb{Z}$ und $m \in \mathbb{N}$. a und b sind genau dann kongruent modulo m, wenn $m|(a-b)$ gilt.
Es wird die Äquivalenz (Schluss in beide Richtungen) der beiden Definitionen gezeigt:

Beweis:
$(a \equiv b \bmod m) \Leftrightarrow [(a = k \cdot m + i) \wedge (b = l \cdot m + i)]$ mit $k, l \in \mathbb{Z}$
$a - b = k \cdot m - l \cdot m = (k - l) \cdot m \Longrightarrow m|(a - b)$, da $(k - l) \in \mathbb{Z}$
Gilt also $a \equiv b \bmod m$, dann gilt auch $m|(a - b)$.

$m|(a - b) \Leftrightarrow a - b = k \cdot m$. Sei nun $a = l \cdot m + i$
$a - b = (l \cdot m + i) - b = k \cdot m \Leftrightarrow b = (l - k) \cdot m + i$
Gilt also $m|(a-b)$, dann sind a und b Elemente derselben Restklasse $[i]_m$ und somit kongruent modulo m.

Es gelten folgende Regeln (vgl. [7]):

Satz 2
Seien $a, b, c, d \in \mathbb{Z}$ und $m, n, k \in \mathbb{N}$, so gilt:

1. $a \equiv b \bmod m \ \wedge \ c \equiv d \bmod m$
   $\Rightarrow (a + c) \equiv (b + d) \bmod m \ \wedge \ (a - c) \equiv (b - d) \bmod m$
2. $a \equiv b \bmod m \ \wedge \ c \equiv d \bmod m$
   $\Rightarrow (a \cdot c) \equiv (b \cdot d) \bmod m$
3. $(k \cdot a) \equiv (k \cdot b) \bmod m \ \wedge \ ggT(k, m) = 1$
   $\Rightarrow a \equiv b \bmod m$
4. $a \equiv b \bmod m \ \wedge \ a \equiv b \bmod n \ \wedge \ ggT(m, n) = 1$
   $\Rightarrow a \equiv b \bmod (m \cdot n)$
5. $a \equiv b \bmod m \Rightarrow a^k \equiv b^k \bmod m$

Beweis:

1. $a \equiv b \bmod m \Leftrightarrow m|(a - b)$
   $c \equiv d \bmod m \Leftrightarrow m|(c - d)$
   Aus Satz 1.5 folgt $m|[(a - b) \pm (c - d)] \Leftrightarrow m|[(a \pm c) - (b \pm d)]$
   $\Longrightarrow (a + c) \equiv (b + d) \bmod m \ \wedge (a - c) \equiv (b - d) \bmod m$
2. Nach Satz 1.3 folgt aus $m|(a-b) \ \wedge \ m|(c-d)$, dass $m|[c(a-b)] \ \wedge \ m|[b(c-d)]$
   Nach Satz 1.5 folgt: $m|[c(a-b)+b(c-d)] \Rightarrow m|(ac-bc+bc-bd) \Rightarrow m|(ac-bd)$
   $\Leftrightarrow (a \cdot c) \equiv (b \cdot d) \bmod m$
3. Nach dem Lemma von Bézout gibt es ganze Zahlen x und y, sodass gilt:
   $ggT(k, m) = 1 = x \cdot k + y \cdot m \Rightarrow (a - b) = (a - b)(x \cdot k) + (a - b)(y \cdot m)$
   Ebenfalls gilt: $m|[k(a - b)] \Leftrightarrow k(a - b) = l \cdot m$

$\Rightarrow a-b = k(a-b)x+(a-b)ym = lmx+(a-b)ym = m(lx+(a-b)y) \Rightarrow m|(a-b)$,
da $(lx + (a - b)y) \in \mathbb{Z}$
$\Leftrightarrow a \equiv b \bmod m$

4. Nach dem Lemma von Bézout gibt es ganze Zahlen x und y, sodass gilt:
   $ggT(m, n) = 1 = x \cdot m + y \cdot n \Rightarrow (a - b) = (a - b)(x \cdot m) + (a - b)(y \cdot n)$
   Ebenfalls gilt: $m|(a - b) \Leftrightarrow (a - b) = k \cdot m \ \wedge \ n|(a - b) \Leftrightarrow (a - b) = l \cdot n$
   $\Rightarrow (a - b) = (a - b)xm + (a - b)yn = lnmx + kmyn = mn(lx + ky)$
   $\Rightarrow mn|(a - b)$, da $(lx + ky) \in \mathbb{Z}$
   $\Leftrightarrow a \equiv b \bmod (m \cdot n)$

5. Aus $a \equiv b \bmod m$ folgt nach Satz 2.2:
   $a \cdot a \equiv b \cdot b \bmod m$ sowie
   $a \cdot a \cdot a \equiv b \cdot b \cdot b \bmod m$
   $\vdots$
   $\underbrace{a \cdot a \cdot \ldots \cdot a}_{a^k} \equiv \underbrace{b \cdot b \cdot \ldots \cdot b}_{b^k} \bmod m$
   Der Spezialfall $k = 0 \Rightarrow 1 \equiv 1 \bmod m$ gilt selbstverständlich ebenfalls.

3.4 Multiplikatives Inverses

Rufen wir uns zunächst das Lemma von Bézout in Erinnerung. Seien a und b ganze Zahlen, so gibt es ganze Zahlen x und y, sodass gilt: $ggT(a, b) = x \cdot a + y \cdot b$

Sei nun $m \in \mathbb{N}^+$ und a eine zu m teilerfremde Zahl ($ggT(a, m) = 1$). Nach dem Lemma von Bézout gibt es ganze Zahlen y und y, sodass gilt:

$$1 = x \cdot a + y \cdot m$$

Betrachten wir diese Gleichung modulo m. Da $y \cdot m \equiv 0 \bmod m$ gilt:

$$1 = x \cdot a + y \cdot m \equiv x \cdot a \bmod m$$

Es gilt $x \equiv i \bmod m$ ($x \in [i]_m$), da x Element irgendeiner Restklasse modulo m sein muss.
Somit gilt: $i \cdot a \equiv 1 \bmod m$

Die Existenz der Zahl i folgt aus dem Lemma von Bézout. Des Weiteren kann i mit dem erweiterten Euklidischen Algorithmus berechnet werden.

Nun müssen wir zeigen, dass nur genau eine solche Zahl i existiert.

Satz 3
Sei m eine positive ganze Zahl und a eine zu m teilerfremde ganze Zahl, so gibt es genau eine Zahl i mit $0 \leq i \leq m-1$, sodass gilt:

$$i \cdot a \equiv 1 \bmod m$$

Man bezeichnet i als das multiplikative Inverse zu a modulo m.

Beweis durch Widerspruch:
Seien $0 \leq i \neq j \leq m-1$ ganze Zahlen, sodass $i \cdot a \equiv 1 \equiv j \cdot a \bmod m$.
Da $ggT(a, m) = 1$ gilt, folgt nach Satz 2.3 $i \equiv j \bmod m$.
Wegen $0 \leq i, j \leq m-1$ muss $i = j$ gelten. Dies ist ein Widerspruch zur Annahme $i \neq j$. Somit gibt es nur genau eine solche Zahl i.

Beispiel 4:
Wir wollen das multiplikative Inverse zu 84 modulo 293 finden.
Wir wissen: $1 = 41 \cdot 293 - 143 \cdot 84$ (vgl. Beispiel 2)
Betrachten wir diese Gleichung modulo 293 erhalten wir:
$1 \equiv -143 \cdot 84 \bmod 293$

Mit -143 haben wir bereits ein multiplikatives Inverse zu 84 modulo 293 gefunden. Normalerweise gibt man jedoch den Repräsentanten der inversen Restklasse an. Dazu müssen wir nur die kleinstmögliche positive Zahl der Restklasse finden.
$-143 + 1 \cdot 293 = 150$
$[150]_{293}$ ist somit die zu $[84]_{293}$ inverse Restklasse modulo 293.

3.5 Satz von Euler-Fermat

Der Satz von Euler-Fermat beinhaltet die Eulersche φ-Funktion. Diese ist wie folgt definiert (vgl. [6]):

Definition: Eulersche φ-Funktion
Sei $n \in \mathbb{N}$. Dann ist die Eulersche φ-Funktion $\varphi(n)$ gegeben durch die Anzahl der Zahlen $1 \leq a < n$ mit $ggT(a, n) = 1$, also der zu n teilerfremden Zahlen.

$$\varphi(n) = |\{a|\ ggT(a, n) = 1, 1 \leq a < n\}|$$

Man bezeichnet die Zahlen $1 \leq a_1 < \ldots < a_{\varphi(n)} < n$ für die $ggT(a_i, n) = 1$

$(i = 1, 2, \ldots, \varphi(n))$ gilt, als prime Reste modulo n und $[a_1]_n, [a_2]_n, \ldots, [a_{\varphi(n)}]_n$ als prime Restklassen modulo n.

Beispiel 5:
1. $\varphi(10) = |\{1, 3, 7, 9\}| = 4$
2. $\varphi(5) = |\{1, 2, 3, 4\}| = 4$

Am zweiten Beispiel erkennt man bereits:
Ist p prim, so gilt: $\varphi(p) = |\{1, 2, \ldots, p-1\}| = p - 1$
Für die nachfolgende Anwendung des Satzes von Euler-Fermat ist $\varphi(p \cdot q)$ mit p und q prim $(p \neq q)$ von Bedeutung. Betrachten wir hierzu alle Zahlen kleiner $p \cdot q$, die zu $p \cdot q$ nicht teilerfremd sind. Dies sind alle Vielfache von p und q.
$|\{p, 2p, 3p, \ldots, (q-1) \cdot p\}| = q - 1$
$|\{q, 2q, 3q, \ldots, (p-1) \cdot q\}| = p - 1$
Ziehen wir nun von der Anzahl aller ganzen Zahlen kleiner $p \cdot q$ die Anzahl der nicht teilerfremden Zahlen ab, so erhält man:
$\varphi(p \cdot q) = (p \cdot q - 1) - (p - 1) - (q - 1) = p \cdot q - p - q + 1 = (p - 1)(q - 1)$

Die Eulersche Verallgemeinerung des kleinen Satzes von Fermat lautet wie folgt (vgl. [6]):

Satz von Euler-Fermat
Seien a und $m \geq 2$ teilerfremde ganze Zahlen. Dann gilt:

$$a^{\varphi(m)} \equiv 1 \bmod m$$

Vorüberlegung:
Seien $1 \leq x_1 < \ldots < x_{\varphi(m)} < m$ die $\varphi(m)$ primen Restklassen modulo m und $y_i := a \cdot x_i$ mit $1 \leq i \leq \varphi(m)$.
Es gilt: Die y_i sind paarweise inkongruent modulo m.

Beweis durch Widerspruch:
Annahme: Zwei y_i und y_j mit $1 \leq i \neq j \leq \varphi(m)$ sind kongruent modulo m.

$$\begin{aligned} y_i &\equiv y_j \bmod m \\ a \cdot x_i &\equiv a \cdot x_j \bmod m \end{aligned}$$

Da $ggT(a, m) = 1$ folgt nach Satz 2.3

$$x_i \equiv x_j \bmod m$$

was jedoch einen Widerspruch zu $i \neq j$ bedeutet, da $1 \leq i \neq j \leq \varphi(m)$.

Beweis:
Laut der Vorüberlegung sind die y_i paarweise inkongruent. Außerdem sind alle y_i teilerfremd zu m, da aus $ggT(x_i, m) = 1$ (laut Definition der φ-Funktion) und $ggT(a, m) = 1$ (laut Voraussetzung) $ggT(a \cdot x_i, m) = ggT(y_i, m) = 1$ folgt. Dies lässt sich leicht über die (eindeutige) Primfaktorzerlegung nachvollziehen.

Es liegen unter den y_i also $\varphi(m)$ zu m teilerfremde Restklassen vor. Da keine Restklasse doppelt vorkommt (y_i paarweise inkongruent), müssen unter ihnen alle primen Restklassen $[x_1]_m, \ldots, [x_{\varphi(m)}]_m$ vorkommen.

Nach Satz 2.2 gilt:

$$\begin{aligned} y_1 \cdot \ldots \cdot y_{\varphi(m)} &\equiv x_1 \cdot \ldots \cdot x_{\varphi(m)} \bmod m \\ a \cdot x_1 \cdot \ldots \cdot a \cdot x_{\varphi(m)} &\equiv x_1 \cdot \ldots \cdot x_{\varphi(m)} \bmod m \\ a^{\varphi(m)} \cdot x_1 \cdot \ldots \cdot x_{\varphi(m)} &\equiv x_1 \cdot \ldots \cdot x_{\varphi(m)} \bmod m \end{aligned}$$

Da $ggT(x_1 \cdot \ldots \cdot x_{\varphi(m)}, m) = 1$ gilt, folgt nach Satz 2.3

$$a^{\varphi(m)} \equiv 1 \bmod m$$

q.e.d.

4 Der RSA-Algorithmus

Dieses Kapitel widmet sich dem RSA-Algorithmus, welcher erklärt und an Beispielen verdeutlicht wird. Außerdem wird ein Beweis zur Gültigkeit des Verfahrens vorgestellt. Zusätzlich wird ein Programm beschrieben, welches den RSA-Algorithmus in die Praxis umsetzt und wesentliche Probleme diskutiert.

4.1 Das Verfahren

Schlüsselerzeugung

1. Man benötigt zwei möglichst große Primzahlen $p \neq q$ mit $p, q > 2$
   Man berechnet $n = p \cdot q$ und $\varphi(n) = (p-1)(q-1)$
2. Man wählt ein $e > 1$ mit $ggT(e, \varphi(n)) = 1$
3. Da e modulo $\varphi(n)$ invertierbar ist (vgl. Satz 3) können wir ein $d > 1$ mit $e \cdot d \equiv 1 \bmod \varphi(n)$ berechnen.
4. e, n bilden den öffentlichlichen und d, n den privaten Schüssel. Alle anderen Daten sollten aus Sicherheitsgründen gelöscht werden.

Verschlüsselung

Um die Nachricht m („message“) mit $m < n$ zu verschlüsseln berechnen wir:

$$c = m^e \bmod n$$

Um die verschlüsselte Nachricht c („ciphertext“) mit $m < n$ zu entschlüsseln berechnen wir:

$$m' = c^d \bmod n$$

Sicherheit:
Jeder Angreifer kennt e und n, da diese den öffenlichen Schlüssel bilden. Um eine Nachricht zu entschlüsseln benötigt er d. Wenn er $\varphi(n)$ kennen würde, könnte er d leicht berechnen. Dazu benötigt er p und q, welche er durch Faktorisierung von n erhält. Man vermutet (!), dass die Sicherheit von RSA auf diesem Problem der Faktorisierung größer Zahlen basiert, es also keinen anderen Weg gibt RSA zu knacken (vgl. [1]).

Beispiel 6:
Zur Verdeutlichung soll das Wort „Facharbeit“ ver- sowie entschlüsselt werden. Um die Buchstaben in Zahlen umzuformen verwenden wir den ASCII-Code.
Somit ergibt sich für die jeweiligen m: 070 097 099 104 097 114 098 101 105 116
Als (gültigen) Schlüssel wählen wir $e = 23$, $d = 155$ und $n = 437$.
Es ist hierbei wichtig, dass $n > 255$ gilt, da man sonst nicht alle Zeichen des ASCII-Codes darstellen könnte (weniger Restklassen als Zeichen).
Wir wenden den Algorithmus exemplarisch an, indem wir uns praktisch selbst die Nachricht schicken. Dazu verschlüsseln wir sie mit unserem öffentlichen Schlüssel e, n.
$c = m^e = 70^{23} \equiv 185 \mod 437$
Fährt man so fort, erhält man: 185 051 283 035 051 114 167 423 174 070
Dies wäre die Nachricht, welche wir erhalten würden. Da nur wir im Besitz des privaten Schlüssels d, n sind, sind folglich auch nur wir in der Lage die Nachricht zu entschlüsseln.
Dazu berechnen wir:
$m' = c^d = 185^{155} \equiv 70 \mod 437$
Wir erhalten weiter: 070 097 099 104 097 114 098 101 105 116
Somit erhalten wir genau die Zeichenkette, die wir uns selbst übermitteln wollten und können diese nach dem ASCII-Code wieder zu „Facharbeit“ dekodieren.

4.2 Beweis des Verfahrens

Wir wollen nun zeigen, dass $m' = c^d \mod n$ wieder die ursprüngliche Nachricht m ergibt. Da m und c beide ganze Zahlen mit $0 \leq m, c < n$ sind, genügt es zu zeigen, dass sie Elemente derselben Restklasse modulo n sind, also:

$$\begin{aligned} m' &\equiv m \mod n \\ c^d &\equiv m \mod n \\ (m^e)^d &\equiv m \mod n \\ m^{ed} &\equiv m \mod n \end{aligned}$$

Der RSA-Algorithmus arbeitet also korrekt, wenn für alle m mit $0 \leq m < n$ gilt:

$$m^{ed} \equiv m \mod n$$

Beweis:
Der Beweis wird mit Hilfe einer Fallunterscheidung durchgeführt. Im ersten Schritt bestätigen wir die zu beweisende Gleichung modulo p, im zweiten modulo q mit p und q prim (Beachte: $n := p \cdot q$). Wir beweisen die Gleichung jeweils für alle

möglichen m, indem wir $ggT(m, p \text{ bzw. } q) = 1$ und $ggT(m, p \text{ bzw. } q) \neq 1$ getrennt betrachten.

Fall 1.1: $ggT(m, p) = 1$
Es gilt: $e \cdot d \equiv 1 \bmod (p-1)(q-1)$ (siehe Schlüsselerzeugung)
Dies ist äquivalent zu: $e \cdot d = k \cdot (p-1)(q-1) + 1$ für ein $k \in \mathbb{Z}$

$$\begin{aligned} m^{ed} &\equiv m^{k\cdot(p-1)(q-1)+1} \bmod p \\ &\equiv m \cdot (m^{(p-1)})^{k\cdot(q-1)} \bmod p \\ &\equiv m \cdot 1^{k\cdot(q-1)} \equiv m \bmod p \end{aligned}$$

Fall 1.2: $ggT(m, p) \neq 1$
Da p prim muss m ein Vielfaches von p sein.
Deswegen gilt: $m \equiv 0 \bmod p$
Nach Satz 2.5 folgt: $m^{ed} \equiv 0^{ed} \equiv 0 \bmod p$
Somit gilt: $m^{ed} \equiv m \bmod p$

Fall 2.1: $ggT(m, q) = 1$
analog zu Fall 1.1

Fall 2.2: $ggT(m, q) \neq 1$
analog zu Fall 1.2

Wir wissen nun:
aus Fall 1.1 und 1.2: $m^{ed} \equiv m \bmod p$
aus Fall 2.1 und 2.2: $m^{ed} \equiv m \bmod q$
Nach Satz 2.4 folgt daraus:

$$\begin{aligned} m^{ed} &\equiv m \bmod (p \cdot q) \\ m^{ed} &\equiv m \bmod n \end{aligned}$$

q.e.d.

4.3 Mein Programm

Um das RSA Verfahren praktisch anzuwenden wurde ein Programm in Delphi geschrieben. Im Besonderen soll an dieser Stelle auf wesentliche Punkte und Probleme eingegangen werden, die bei der praktischen Umsetzung des Verfahrens eine Rolle spielen.

Problempunkte:

1. RSA gilt als sicher, weil es sehr aufwendig ist große Zahlen zu faktorisieren. In der Praxis verwendet man Schlüssel die ungefähr 200 Dezimalstellen besitzen. Doch trifft man auch bei deutlich kleineren Zahlen auf das Problem des Umgangs mit „großen“ Zahlen. Schon bei der Berechnung von $10^{10} \bmod n$ wird der Zahlenbereich einer Integervariable überschritten. Die Lösung dieses Problems lautet „square and multiply“ (vgl. [1]).
   Beispiel: $10^7 \bmod n = 10 \cdot (10 \cdot (10^2 \bmod n) \bmod n)^2 \bmod n$
   Durch diese modularen Reduktionen erreichen wir, dass die Zahlenwerte deutlich geringer sind. Trotzdem ist uns durch die Integervariable ein maximaler Wert vorgegeben ($max_{int} = 2147483647 \Rightarrow max = 46340 = round(\sqrt{max_{int}})$).

2. Um Schlüssel zu generieren benötigt man Primzahlen p und q. Diese müssen wir in unserem Fall erst generieren, doch wie erzeugt man Primzahlen?
   Erfreulicherweise haben wir bereits eine Möglichkeit kennengelernt. Aus dem kleinen Satz von Fermat (Satz von Euler Fermat mit m=prim) folgt:
   Gibt es eine Zahl a mit $ggT(a, p) = 1$ für die gilt: $a^{p-1} \not\equiv 1 \bmod p$, dann ist p definitiv nicht prim.
   Es ist uns möglich daraus einen Monte-Carlo-Algorithmus (randomisierter Algorithmus, der mit einer nach oben beschränkten Wahrscheinlichkeit ein falsches Ergebnis liefert) zu formulieren.
   Gilt $a^{p-1} \equiv 1 \bmod p$ für alle zufällig gewählten a, so ist p mit einer bestimmten Wahrscheinlichkeit prim. Die Wahrscheinlichkeit, dass der Satz gilt, obwohl p nicht prim ist liegt bei 50% (vgl. [1]). Wiederholt man diesen Test n-mal, so halbiert sich die Fehlerwahrscheinlichkeit mit jedem Durchlauf ($\frac{1}{2^n}$).
   Aber Achtung: Es gibt Zahlen, die den Test für alle Basen bestehen, obwohl sie zusammengesetzt sind, die sogenannten Carmichael-Zahlen (z.B. 561).
   Um die Sicherheit des Verfahrens zu gewährleisten, wäre es deshalb geschickt einen effektiveren Test (z.B. Miller-Rabin-Test) zu verwenden, aber im Sinne der Verdeutlichung werden im Programm alle Primzahlen über den kleinen Satz von Fermat generiert.

3. Um einen Klartext zu verschlüsseln muss man erst die Zeichen in Zahlen umwandeln, mit denen wir rechnen können. Hierzu wird der ASCII-Code verwendet.

4. Der größte gemeinsame Teiler und das Inverse modulo m werden über den erweiterten Euklidischen Algorithmus berechnet, der sich besonders gut rekursiv programmieren lässt.

Bewertung:
Der vorgestellte RSA-Algorithmus wurde in ein Programm implementiert, welches in Delphi geschrieben wurde. Es bietet die Möglichkeit Schlüssel nach eigenen Vorgaben zu generieren oder zu überprüfen. Man kann damit Texte ver- und entschlüsseln, sowie Dokumente signieren oder bestehende Signaturen überprüfen. Außerdem ist es möglich Texte jederzeit abzuspeichern und zu laden. Des Weiteren bietet das Programm die Möglichkeit jeden Schritt nachzurechnen, um ihn so besser nachvollziehen zu können.

An dieser Stelle soll nun der Sicherheitsaspekt betrachtet werden. Um in der Realität professionell zur Verschlüsselung genutzt zu werden, müssten die verwendeten Schlüssel deutlich länger sein um eine Faktorisierung von n zu verhindern. Viele Angriffe richten sich nicht gegen den Grundalgorithmus, sondern gegen die Implementierung von RSA (vgl. [1]). Beispielsweise sollte man sich mit dem Interlockprotokoll gegen eine „man-in-the-middle attack“ schützen, bei der ein Angreifer vorgibt der Empfänger zu sein und die Nachricht so abfangen kann (vgl. [2]). Dabei müsste der chiffrierte Text in Blöcke unterteilt werden, die einzeln geschickt werden. Dies hat außerdem zur Folge, dass man mit einer simplen Häufigkeitsanalyse (vgl. [2]) keine Chance hätte die Nachricht zu entschlüsseln. Wählt man die Schlüssel nach besonderen Kriterien aus (z.B. e und d möglichst groß) erhöht sich die Sicherheit der Verschlüsselung. Für e wählt man z.B. oft eine Zahl, deren Binärdarstellung viele Nullen enthält (z.B. $2^{16} + 1 = 65537$). Für diese Exponenten erweist sich der „square and multiply“ Algorithmus als außerordentlich effektiv. Es wäre auch geschickt RSA nur für die sichere Übermittlung eines Sitzungsschlüssels zu verwenden und die eigentliche Nachricht per symmetrischen Algorithmus zu verschlüsseln. Diese hybriden Systeme sind in der Praxis deutlich effektiver. Ein Beispiel für ein ausgereiftes Programm, welches dies alles schon leistet wäre PGP (Pretty Good Privacy).

5 Schlussbemerkung

In dieser Arbeit wurde das Prinzip der asymmetrischen Verschlüsselung erläutert und im Besonderen auf das RSA-Verfahren eingegangen. Dazu wurden die zahlentheoretischen Grundlagen behandelt und an Beispielen verdeutlicht. Außerdem wurde die Gültigkeit des Verfahrens bewiesen und ein Programm vorgestellt, welches den Algorithmus nachvollziehbar umsetzt.

Zum Schluss möchte ich auf die Zukunftsaussichten des behandelten RSA-Verfahrens eingehen. Der RSA-Algorithmus ist seit seiner Veröffentlichung der unangefochtene Standard im Bereich der asymmetrischen Verschlüsselung, obwohl man bisher nur vermutet, dass es sich bei der Faktorisierung um ein mathematisch hartes Problem handelt. Zudem sind seit Entwicklung des Zahlkörpersiebs durch J-M. Pollard (1991) keine prinzipiell neuen Ansätze zur Faktorisierung publiziert worden.
2005 gelang es Mathematikern der Universität Bonn eine 200-stellige Dezimalzahl zu faktorisieren. Die Faktorisierung begann Ende 2003 und dauerte bis Mai 2005.
Trotz dieses Fortschrittes bei Faktorisierungsalgorithmen ist RSA nicht gefährdet, da der Modul n nur hinreichend groß gewählt werden muss um eine hohe Sicherheit zu gewährleisten. Allerdings könnte jederzeit ein neuer Algorithmus gefunden werden, der auf einen Schlag RSA unbrauchbar machen würde. Dies wäre verheerend, da Schlüssel asymmetrischer Verfahren praktisch als Generalschlüssel über lange Zeit verwendet werden. Einen solchen Algorithmus gibt es bereits für Quantencomputer mit dem Shor-Algorithmus zur Faktorisierung großer natürlicher Zahlen.
Reinhard Wobst bringt dies wie folgt auf den Punkt (siehe [2], S.174):

> Die gegenwärtigen asymmetrischen Verfahren bieten großen Komfort bei sehr hoher Sicherheit mit extrem großen Schaden bei eventueller Kompromitierung.

6 Quellenangaben

Literatur

[1] Schneier, Bruce
Angewandte Kryptographie
Addison-Wesley, 1996.

[2] Wobst, Reinhard
Abenteuer Kryptologie
Addison-Wesley, 1998.

[3] Beutelspacher, Albrecht
Kryptologie
Vieweg Verlag, 1987.

[4] Forster, Otto
Algorithmische Zahlentheorie
Vieweg Verlag, 1996.

[5] Singh, Simon
Geheime Botschaften
Carl Hanser Verlag, 2000.

[6] Heuberger, Clemens
Zahlentheorie
http://www.oemo.at/intern/formel/zahlentheorie.pdf
Stand: 8. Mai 2006.

[7] Ge, Yimin
Die Mathematik von RSA
http://yimin.sinuslab.net/download.php?&id=4, 2005
Stand: 8. Mai 2006.

[8] Kurtz, Andreas
Erstellen einer Facharbeit mit LaTeX
http://www.rzuser.uni-heidelberg.de/~akurtz/vortraege/latex-einfuehrung.pdf, 2004
Stand: 8. Mai 2006.

7 Anhang

Quelltext des Programms (Unit1.pas)

```
1   unit Unit1;

    interface

5   uses
      Windows, Messages, SysUtils, Classes, Graphics,
      Controls, Forms, Dialogs, Menus, ComCtrls, StdCtrls;

    type
10    TRSA = class(TForm)
        OpenDialog1: TOpenDialog;
        SaveDialog1: TSaveDialog;
        MainMenu1: TMainMenu;
        Datei1: TMenuItem;
15      Bearbeiten1: TMenuItem;
        Quelltextladen1: TMenuItem;
        Quelltextspeichern1: TMenuItem;
        Codetextladen1: TMenuItem;
        Codetextspeichern1: TMenuItem;
20      Schlsselberpfen1: TMenuItem;
        Schlsselgenerieren1: TMenuItem;
        Verschlsseln1: TMenuItem;
        Entschlsseln1: TMenuItem;
        Signieren1: TMenuItem;
25      Signaturberprfen1: TMenuItem;
        PageControl1: TPageControl;
        Schluessel: TTabSheet;
        Label1: TLabel;
        Label2: TLabel;
30      Label3: TLabel;
        Label4: TLabel;
        Label5: TLabel;
        e_edit: TEdit;
        d_edit: TEdit;
35      p_edit: TEdit;
        Verschluesseln: TTabSheet;
        Label6: TLabel;
        Label7: TLabel;
        quelltext: TMemo;
40      codetext: TMemo;
        Signieren: TTabSheet;
        Label10: TLabel;
        Label11: TLabel;
```

```
        dokument: TMemo;
45      signatur: TMemo;
        Dokumentladen1: TMenuItem;
        Dokumentspeichern1: TMenuItem;
        Signaturladen1: TMenuItem;
        Signaturspeichern1: TMenuItem;
50      Label8: TLabel;
        q_edit: TEdit;
        Label9: TLabel;
        Label12: TLabel;
        Label13: TLabel;
55      Label14: TLabel;
        Label15: TLabel;
        Label16: TLabel;
        Label17: TLabel;
        unten_edit: TEdit;
60      oben_edit: TEdit;
        anzahl_edit: TEdit;
        durchl_p: TLabel;
        zeit_p: TLabel;
        Label18: TLabel;
65      Label19: TLabel;
        e_label: TLabel;
        d_label: TLabel;
        Label20: TLabel;
        Label21: TLabel;
70      n_label: TLabel;
        Button1: TButton;
        Label22: TLabel;
        Label23: TLabel;
        durchl_q: TLabel;
75      zeit_q: TLabel;
        Label24: TLabel;
        gueltig_label: TLabel;
        Label25: TLabel;
        Beenden1: TMenuItem;
80      Label26: TLabel;
        Label27: TLabel;
        basis: TEdit;
        exponent: TEdit;
        modulus: TEdit;
85      Label28: TLabel;
        Label29: TLabel;
        Label30: TLabel;
        ergebnis: TLabel;
        Label32: TLabel;
90      Button2: TButton;
```

```
    Label31: TLabel;
    Label33: TLabel;
    Label34: TLabel;
    a: TEdit;
    b: TEdit;
    Label35: TLabel;
    eea_erg: TLabel;
    Button3: TButton;
    Label36: TLabel;
    Label37: TLabel;
    primzahl: TEdit;
    Label38: TLabel;
    testbasen: TEdit;
    Label39: TLabel;
    gueltig: TLabel;
    Button4: TButton;
    n_edit: TEdit;
    procedure PageControl1Change(Sender: TObject);
    procedure Schlsselgenerieren1Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Schlsselberpfen1Click(Sender: TObject);
    procedure Verschlsseln1Click(Sender: TObject);
    procedure Entschlsseln1Click(Sender: TObject);
    procedure Quelltextladen1Click(Sender: TObject);
    procedure Codetextladen1Click(Sender: TObject);
    procedure Quelltextspeichern1Click(Sender: TObject);
    procedure Codetextspeichern1Click(Sender: TObject);
    procedure Beenden1Click(Sender: TObject);
    procedure Signieren1Click(Sender: TObject);
    procedure Signaturberprfen1Click(Sender: TObject);
    procedure Dokumentladen1Click(Sender: TObject);
    procedure Signaturladen1Click(Sender: TObject);
    procedure Dokumentspeichern1Click(Sender: TObject);
    procedure Signaturspeichern1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private { Private-Deklarationen }
  public { Public-Deklarationen }
  end;

var
  RSA: TRSA;
  e,d,p,q,n,phin,m,c,start,ende,anzahl:integer;
  //Schluessel als oeffentliche Variablen
  t:string;
```

```
implementation

{$R *.DFM}


function modexp(b,e,m:Integer):Integer;
//square and multiply
//berechnet schnell x=b^e mod m
begin
result:=1;
while e<>0 do
      begin
      if e mod 2=0 then
         begin
         b:=(b*b) mod m;
         e:=e div 2;
         end
      else
         begin
         result:=(result*b) mod m;
         e:=e-1;
         end;
      end;
end;


function ggt(a,b:integer):integer;
//Euklidischer Algorithmus (rekursiv)
//gibt den ggT(a,b) direkt zurueck
var     r:integer;
begin
a:=abs(a);
b:=abs(b);
r:=b;
while r>0 do
        begin
        r:=a mod b;
        a:=b;
        b:=r;
        end;
result:=a;
end;
```

```
180
     procedure eea(a,b:integer; var m,s,u:integer);
     //http://de.wikipedia.org/wiki/
     //Erweiterter_Euklidischer_Algorithmus
     //Forster Zahlentheorie
185  //benoetigt zwei Zahlen a,b
     //gibt m=ggT(a,b) und deren Vorfaktoren s und u
     //fuer die Vielfachsummendarstellung des ggT zurueck
     var n,t,v,tmp,q,r:integer;
     begin
190  m:=a; n:=b; //ggT
     s:=1; t:=0; //Vorfaktor "a"
     u:=0; v:=1; //Vorfaktor "b"

     while n>0 do //rekursive Berechnung
195          begin
             q:=m div n;
             r:=m-q*n;
             m:=n; n:=r;
             tmp:=t; t:=s-q*t; s:=tmp;
200          tmp:=v; v:=u-q*v; u:=tmp;
             end;
     end;


205  function prim(p,anzahl:integer):boolean;
     //kleiner Satz von Fermat
     //Primzahltest
     //p: zu testende Zahl; anzahl: Anzahl der Testbasen
     //gibt true/false zurueck
210  var a,b:integer;
     begin
     randomize;
     result:=true;
     a:=0;
215          repeat
             b:=random(p-2)+2; //b \in [2..p-1]; Testbase
             if modexp(b,p-1,p)=1 then a:=a+1
             else begin result:=false; a:=anzahl; end;
             until a=anzahl; //gewuenschte Anzahl an Durchlaeufen
220  end;
```

```
procedure primgent(var p,d:integer;
var t:string; start,ende,anzahl:integer);
//kleiner Satz von Fermat
//http://www.dsdt.info/tipps/?id=53; Zeitmessung
//generiert Primzahl p mit start <= p < ende
//anzahl: Anzahl der Testbasen des kleinen Satzes von Fermat
//d: Anzahl der Durchlaeufe
//   die benoetigt wurden um die Primzahl zu finden
//t: benoetigte Zeit zum generieren in ms
var a,b:integer;
    x,y,z: TLargeInteger;
begin

QueryPerformanceFrequency(x);
QueryPerformanceCounter(y); //Zeitmessung Start

randomize;
p:=random(ende-start)+start;
a:=0;
d:=1;

repeat
      b:=random(p-2)+2; //b \in [2..p-1]
      if modexp(b,p-1,p)=1 then a:=a+1
         else
             begin
             p:=random(ende-start)+start;
             a:=0;
             d:=d+1;
             end;
until a=anzahl;

QueryPerformanceCounter(z); //Zeitmessung Stop
t:=Format('%.3f ms',[(z.QuadPart-y.QuadPart)*1000/x.QuadPart]);
//Ausgabe in ms
end;


procedure TRSA.Schlsselgenerieren1Click(Sender: TObject);
//alle benoetigten Schluessel werden generiert
```

```
var m,s,u: integer;
begin
start:=strtoint(unten_edit.text);
ende:=strtoint(oben_edit.text);
anzahl:=strtoint(anzahl_edit.text);

repeat
primgent(p,d,t,start,ende,anzahl); //Primzahl p
p_edit.text:=inttostr(p);
durchl_p.caption:=inttostr(d);
zeit_p.caption:=t;

primgent(q,d,t,start,ende,anzahl); //Primzahl q
q_edit.text:=inttostr(q);
durchl_q.caption:=inttostr(d);
zeit_q.caption:=t;
until p<>q;
//notwendige Bedingung, da sonst Faktorisierung zu leicht

n:=p*q; //Schluessel werden in oeffentlicher Variable abgelegt
phin:=(p-1)*(q-1);
n_edit.text:=inttostr(n);

repeat  //man waehlt ein e mit ggt(e,phi(n))=1
e:=random(phin-3)+3; //e \in [3..phi(n)-1]
eea(e,phin,m,s,u);
until m=1;

d:=s; //d ist multiplikatives zu e mod phi(n)
while d<0 do d:=d+phin; //d = Repraesentant der Restklasse

e_edit.text:=inttostr(e);
d_edit.text:=inttostr(d);
end;


procedure TRSA.Schlsselberpfen1Click(Sender: TObject);
//Schluessel werden auf Gueltigkeit ueberprueft
var gueltig:boolean;
    tmp:integer;
begin
p:=strtoint(p_edit.text);
```

```
305   q:=strtoint(q_edit.text);
      e:=strtoint(e_edit.text);
      d:=strtoint(d_edit.text);
      n:=p*q;
      n_edit.text:=inttostr(n);
310   phin:=(p-1)*(q-1);
      anzahl:=strtoint(anzahl_edit.text);

      gueltig:=true;
      IF NOT prim(p,anzahl) THEN gueltig:=false; //p prim
315   IF NOT prim(q,anzahl) THEN gueltig:=false; //q prim
      IF p=q THEN gueltig:=false; //p <> q

      tmp:=ggT(e,phin);
      IF tmp<>1 THEN gueltig:=false; //ggT(e,phi(n))=1
320   tmp:=(e*d) mod phin;
      IF tmp<>1 THEN gueltig:=false; //(e*d)=1 mod phi(n)

      IF gueltig=true THEN
      gueltig_label.caption:='ja'
325   else gueltig_label.caption:='nein';
      end;



      procedure TRSA.Verschlsseln1Click(Sender: TObject);
330   //Text aus dem Quelltextmemo wird verschluesselt
      //und im Codetextmemo abgelegt
      var l,i:integer;
          tmp,qzeile,czeile:string;
      begin
335   e:=strtoint(e_label.caption);
      d:=strtoint(d_label.caption);
      n:=strtoint(n_label.caption);

      For l:=0 to (codetext.lines.count-1) do codetext.lines[l]:='';
340   //Codetextmemo wird geloescht
      while codetext.lines.count<100 do codetext.Lines.Add('');
      //Codetextmemo wird auf 100 Zeilen aufgefuellt
      //um Speicher fuer die Zeilen freizugegeben

345   For l:=0 to (quelltext.lines.count-1) do
      //zeilenweise Verschluesselung
```

```
      begin
      qzeile:=quelltext.lines[l];
      czeile:='';
350           For i:=1 to length(qzeile) do
              //innerhalb der Zeile zeichenweise Verschluesselung
              begin
              m:=ord(qzeile[i]); //ASCII-Code
              c:=modexp(m,e,n); //Verschluesselung
355           tmp:=inttostr(c);
              While length(tmp)<length(inttostr(n)) DO tmp:='0'+tmp;
              //Auffuellen um Blocklaenge beizubehalten
              czeile:=czeile+tmp;
              end;
360   codetext.lines[l]:=czeile;
      end;
      end;


365   procedure TRSA.Entschlsseln1Click(Sender: TObject);
      //Zeichenkette aus dem Codetextmemo wird entschluesselt
      //und im Quelltextmemo abgelegt
      var l,i,j:integer;
          tmp,qzeile,czeile:string;
370   begin
      e:=strtoint(e_label.caption);
      d:=strtoint(d_label.caption);
      n:=strtoint(n_label.caption);

375   For l:=0 to (quelltext.lines.count-1) do quelltext.lines[l]:='';
      //Quelltextmemo wird geloescht
      while quelltext.lines.count<100 do quelltext.Lines.Add('');
      //Quelltextmemo wird auf 100 Zeilen aufgefuellt
      //um Speicher fuer die Zeilen freizugegeben
380
      For l:=0 to (codetext.lines.count-1) do
      //zeilenweise Entschluesselung
      begin
      czeile:=codetext.lines[l];
385   qzeile:='';
              For i:=1 to (length(czeile) div length(inttostr(n))) do
              //Anzahl der Bloecke; Blocklaenge=length(n)
              begin
```

```
            tmp:='';
390                 for j:=1 to length(inttostr(n)) do
                    //jeweils 1 Block in tmp (zeichenweise)
                    begin
                    tmp:=tmp+czeile[((i-1)*length(inttostr(n)))+j]
                    end;
395         c:=strtoint(tmp); //Block --> Zahl
            m:=modexp(c,d,n); //Entschluesselung
            tmp:=chr(m); //ASCII-Code
            qzeile:=qzeile+tmp;
            end;
400 quelltext.lines[l]:=qzeile;
    end;
    end;


405 procedure TRSA.Signieren1Click(Sender: TObject);
    //signieren = verschluesseln mit d (priv. Schluessel)
    //Text aus dem Dokumentmemo wird mit d verschluesselt
    //und im Signaturmemo abgelegt
    var l,i:integer;
410     tmp,qzeile,czeile:string;
    begin
    e:=strtoint(e_label.caption);
    d:=strtoint(d_label.caption);
    n:=strtoint(n_label.caption);
415
    For l:=0 to (signatur.lines.count-1) do signatur.lines[l]:='';
    //Signaturmemo wird geloescht
    while signatur.lines.count<100 do signatur.Lines.Add('');
    //Signaturmemo wird auf 100 Zeilen aufgefuellt
420 //um Speicher fuer die Zeilen freizugegeben

    For l:=0 to (dokument.lines.count-1) do
    //zeilenweise Verschluesselung
    begin
425 qzeile:=dokument.lines[l];
    czeile:='';
            For i:=1 to length(qzeile) do
            //innerhalb der Zeile zeichenweise Verschluesselung
            begin
430         m:=ord(qzeile[i]); //ASCII-Code
```

```
            c:=modexp(m,d,n); //Verschluesselung mit d, da Signatur
            tmp:=inttostr(c);
            While length(tmp)<length(inttostr(n)) DO tmp:='0'+tmp;
            //Auffuellen um Blocklaenge beizubehalten
            czeile:=czeile+tmp;
            end;
    signatur.lines[l]:=czeile;
    end;
    end;


    procedure TRSA.Signaturberprfen1Click(Sender: TObject);
    //Signatur ueberpruefen = entschluesseln mit e
    //Zeichenkette aus dem Signaturmemo wird mit e entschluesselt
    //und im Dokumentmemo abgelegt
    var l,i,j:integer;
        tmp,qzeile,czeile:string;
    begin
    e:=strtoint(e_label.caption);
    d:=strtoint(d_label.caption);
    n:=strtoint(n_label.caption);

    For l:=0 to (dokument.lines.count-1) do dokument.lines[l]:='';
    //Dokumentmemo wird geloescht
    while dokument.lines.count<100 do dokument.Lines.Add('');
    //Dokumentmemo wird auf 100 Zeilen aufgefuellt
    //um Speicher fuer die Zeilen freizugegeben

    For l:=0 to (signatur.lines.count-1) do
    //zeilenweise Entschluesselung
    begin
    czeile:=signatur.lines[l];
    qzeile:='';
            For i:=1 to (length(czeile) div length(inttostr(n))) do
            //Anzahl der Bloecke; Blocklaenge=length(n)
            begin
            tmp:='';
                    For j:=1 to length(inttostr(n)) do
                    //jeweils 1 Block in tmp (zeichenweise)
                    begin
                    tmp:=tmp+czeile[((i-1)*length(inttostr(n)))+j]
                    end;
```

```
            c:=strtoint(tmp); //Block --> Zahl
            m:=modexp(c,e,n); //Entschluesselung
            tmp:=chr(m); //ASCII-Code
            qzeile:=qzeile+tmp;
            end;
      dokument.lines[l]:=qzeile;
      end;
      end;


      procedure TRSA.PageControl1Change(Sender: TObject);
      //Steuerung des Menues
      //Moeglichkeiten werden an- bzw. ausgeschaltet
      begin

      IF pagecontrol1.activepage=Schluessel THEN
      begin
      Quelltextladen1.enabled:=false;
      Quelltextspeichern1.enabled:=false;
      Codetextladen1.enabled:=false;
      Codetextspeichern1.enabled:=false;
      Dokumentladen1.enabled:=false;
      Dokumentspeichern1.enabled:=false;
      Signaturladen1.enabled:=false;
      Signaturspeichern1.enabled:=false;

      Schlsselberpfen1.enabled:=true;
      Schlsselgenerieren1.enabled:=true;
      Verschlsseln1.enabled:=false;
      Entschlsseln1.enabled:=false;
      Signieren1.enabled:=false;
      Signaturberprfen1.enabled:=false;
      end;

      IF pagecontrol1.activepage=Verschluesseln THEN
      begin
      Quelltextladen1.enabled:=true;
      Quelltextspeichern1.enabled:=true;
      Codetextladen1.enabled:=true;
      Codetextspeichern1.enabled:=true;
      Dokumentladen1.enabled:=false;
      Dokumentspeichern1.enabled:=false;
      Signaturladen1.enabled:=false;
      Signaturspeichern1.enabled:=false;

      Schlsselberpfen1.enabled:=false;
```

```
      Schlsselgenerieren1.enabled:=false;
      Verschlsseln1.enabled:=true;
520   Entschlsseln1.enabled:=true;
      Signieren1.enabled:=false;
      Signaturberprfen1.enabled:=false;
      end;

525   IF pagecontrol1.activepage=Signieren THEN
      begin
      Quelltextladen1.enabled:=false;
      Quelltextspeichern1.enabled:=false;
      Codetextladen1.enabled:=false;
530   Codetextspeichern1.enabled:=false;
      Dokumentladen1.enabled:=true;
      Dokumentspeichern1.enabled:=true;
      Signaturladen1.enabled:=true;
      Signaturspeichern1.enabled:=true;
535
      Schlsselberpfen1.enabled:=false;
      Schlsselgenerieren1.enabled:=false;
      Verschlsseln1.enabled:=false;
      Entschlsseln1.enabled:=false;
540   Signieren1.enabled:=true;
      Signaturberprfen1.enabled:=true;
      end;
      end;

545
      procedure TRSA.Button1Click(Sender: TObject);
      //Schluessel werden uebernommen
      begin
      e_label.caption:=e_edit.text;
550   d_label.caption:=d_edit.text;
      n_label.caption:=n_edit.text;
      end;


555   procedure TRSA.Quelltextladen1Click(Sender: TObject);
      begin
      opendialog1.execute;
      quelltext.lines.loadfromfile(opendialog1.FileName);
      end;
560

      procedure TRSA.Codetextladen1Click(Sender: TObject);
      begin
      opendialog1.execute;
```

```
565   codetext.lines.loadfromfile(opendialog1.FileName);
      end;


      procedure TRSA.Quelltextspeichern1Click(Sender: TObject);
570   begin
      savedialog1.Execute;
      quelltext.lines.savetofile(savedialog1.FileName);
      end;

575
      procedure TRSA.Codetextspeichern1Click(Sender: TObject);
      begin
      savedialog1.Execute;
      codetext.lines.savetofile(savedialog1.FileName);
580   end;


      procedure TRSA.Beenden1Click(Sender: TObject);
      begin
585   close;
      end;


      procedure TRSA.Dokumentladen1Click(Sender: TObject);
590   begin
      opendialog1.execute;
      dokument.lines.loadfromfile(opendialog1.FileName);
      end;

595
      procedure TRSA.Signaturladen1Click(Sender: TObject);
      begin
      opendialog1.execute;
      signatur.lines.loadfromfile(opendialog1.FileName);
600   end;


      procedure TRSA.Dokumentspeichern1Click(Sender: TObject);
      begin
605   savedialog1.Execute;
      dokument.lines.savetofile(savedialog1.FileName);
      end;


610   procedure TRSA.Signaturspeichern1Click(Sender: TObject);
      begin
```

```
savedialog1.Execute;
signatur.lines.savetofile(savedialog1.FileName);
end;


procedure TRSA.Button2Click(Sender: TObject);
//RSA Taschenrechner
//Button: square and multiply
begin
ergebnis.caption:=inttostr(modexp(strtoint(basis.text),
strtoint(exponent.text),strtoint(modulus.text)));
end;


procedure TRSA.Button3Click(Sender: TObject);
//RSA Taschenrechner
//Button: erweiterter Euklidischer Algorithmus
var m,s,u:integer;
begin
eea(strtoint(a.text),strtoint(b.text),m,s,u);
eea_erg.caption:=inttostr(m)+'='+inttostr(s)+'*'
+a.text+'+'+inttostr(u)+'*'+b.text;
end;


procedure TRSA.Button4Click(Sender: TObject);
//RSA Taschenrechner
//Button: Fermatscher Primzahltest
begin
IF prim(strtoint(primzahl.text),strtoint(testbasen.text))
THEN gueltig.caption:='ja'
else gueltig.caption:='nein'
end;


procedure TRSA.FormCreate(Sender: TObject);
//Memos werden auf 100 Zeilen aufgefuellt
//um Speicher fuer die Zeilen freizugegeben
begin
while quelltext.lines.count<100 do quelltext.Lines.Add('');
while codetext.lines.count<100 do codetext.Lines.Add('');
while dokument.lines.count<100 do dokument.Lines.Add('');
while signatur.lines.count<100 do signatur.Lines.Add('');
end;


end.
```

Ich erkläre hiermit, dass ich die Facharbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

_____, den	_____	_____
Ort	Datum	Unterschrift