

FACHARBEIT

Realisierung und Anwendung des RSA- Algorithmus in der Informatik

Daniel Dieckelmann

Inda-Gymnasium Aachen-Kornelimünster

Februar / März 2007

Unterrichtsfach: Informatik Grundkurs

Fachlehrer: Konrad Ewers

Inhaltsverzeichnis

Einleitung	2
1. Einführung	2
2. Das Verfahren	
2.1 Der Algorithmus	3
2.2 Ver- und Entschlüsselung	5
2.3 Informationstechnologische Umsetzung mit JAVA	6
3. Anwendung des Algorithmus	
3.1 Sicherheitsaspekte	10
3.2 Tatsächliche Anwendungsgebiete von RSA	12
4. Diskussion	13
Literaturverzeichnis	14

Anlagen:

- I. Erklärung
- II. Abstract
- III. Internetquellen

Einleitung

Diese Facharbeit befasst sich mit dem RSA-Algorithmus. Dabei wird besonderer Wert auf die informationstechnologische Realisierung des Algorithmus in JAVA und die Anwendung dieses Verfahrens gelegt. In diesen Bereichen wird das eigentliche Verfahren genau erklärt und generelle Funktionsweise, Sicherheitsaspekte und tatsächliche Anwendungsgebiete des Systems thematisiert. Eine Pro- und Contradiskussion, in der Stärken und Schwächen des Verfahrens übersichtlich aufgelistet werden, bildet den Abschluss dieser Arbeit. Auf eine umfassende mathematische Beweisführung und Erklärung des Verfahrens muss aus Gründen der inhaltlichen Fülle verzichtet werden. Da es sich hierbei um eine Facharbeit im Fach Informatik handelt, wird stattdessen die Entwicklung eines JAVA-Programms innerhalb der Arbeit hervorgehoben. Das Quellmaterial besteht zu einem großen Teil aus zuverlässigen Internetquellen, sowie einiger themenspezifischer Literatur.

1. Einführung

Der RSA-Algorithmus ist ein so genanntes asymmetrisches Kryptosystem. Entwickelt wurde er im Jahr 1977 von den drei Mathematikern Ron Rivest, Adi Shamir und Len Adleman, deren Anfangsbuchstaben ihrer Nachnamen dem Verfahren seinen Namen gaben. Bei einem asymmetrischen Verschlüsselungssystem (oft auch als Public-Key-Verfahren bezeichnet) muss im Gegensatz zu herkömmlichen Verfahren kein geheimer Schlüssel zwischen Absender und Empfänger ausgetauscht werden, mit dem man Nachrichten ver- und wieder entschlüsseln kann. Bei RSA gibt es einen geheimen Schlüssel, den sog. *Private Key*, den nur der Empfänger kennt und mit dem er eine an ihn gesendete verschlüsselte Nachricht entschlüsseln kann. Einem Absender wird der *Public Key* übergeben, mit dem er seine Nachricht verschlüsseln kann. Bei einem asymmetrischen Verfahren gibt es also nur einen Empfänger, der verschlüsselte Nachrichten entschlüsseln kann, aber im Prinzip beliebig viele Absender, die mit dem der Öffentlichkeit bekanntem Schlüssel ihre Nachrichten verschlüsseln können, so dass nur der Empfänger diese entschlüsseln und folglich auch lesen kann. Der *Private Key* wird aus dem *Public Key* errechnet. Die ganze Sicherheit in diesem System besteht darin, dass eine Zurückrechnung von einem *Public Key* auf einen *Private Key* äußerst schwierig, wenn nicht sogar unmöglich ist.

Rivest, Shamir und Adleman entwickelten auf der Grundlage einer von Whitfield Diffie und Martin Hellman verfassten Theorie zur Public-Key-Kryptografie ein Kryptosystem, welches diesen Anforderungen entsprach.¹ Die Kernidee und damit verbunden die Sicherheit ihres Algorithmus hängt dabei sehr stark vom dem mathematischen Problem ab, (sehr) große Zahlen in ihre Primfaktoren zu zerlegen (Primfaktorzerlegung). Die drei Männer leisteten einen großen Beitrag zur Entwicklung von Public-Key-Verfahren, indem sie ein für damalige Verhältnisse revolutionär einfaches mathematisches Verfahren dazu benutzen, dieses neue Problem anzugehen und zu lösen. Bis heute ist der RSA-Algorithmus das am häufigsten benutzte aber auch das am hitzigsten diskutierte Public-Key-Verfahren.

2. Das Verfahren

2.1 Der Algorithmus

Wie sieht nun der Algorithmus, der eine asymmetrische Verschlüsselung möglich macht, genau aus? Wir wollen uns nun anschauen, was nötig ist, um den *Private Key* und den *Public Key* herzustellen.

Zunächst wenden wir uns der Herstellung des öffentlichen Schlüssels zu. Dieser besteht aus den Zahlen n und e . Die Erzeugung von n ist sehr einfach. Man multipliziert zwei Primzahlen p und q miteinander. Im Kapitel 3.1 werden wir uns genauer mit der Wahl dieser Primfaktoren befassen.

$$n = p \cdot q$$

Dann berechnen wir die so genannte Eulersche Phi-Funktion $\varphi(n)$.

$$\varphi(n) = (p - 1) \cdot (q - 1)$$

Nun wählen wir eine Zahl e , die teilerfremd zu $\varphi(n)$ ist und die größer als 1 aber nicht größer als $\varphi(n)$ ist.

¹ Quelle [5], Abschnitt *Verfahren*

$$1 < e < \varphi(n) \text{ mit } \text{ggT}(e, \varphi(n)) = 1$$

Zwei Zahlen sind teilerfremd, wenn es keine natürliche Zahl außer 1 gibt, die sowohl die eine, als auch die andere der beiden Zahlen teilt. Das heißt, dass der größte gemeinsame Teiler dieser beiden Zahlen 1 beträgt.

Jetzt berechnen wir den privaten Schlüssel, der aus den Zahlen n und d besteht. n kennen wir bereits, wir brauchen also nur noch d :

$$\begin{aligned} e \cdot d \bmod \varphi(n) &= 1 \\ \Leftrightarrow (e \cdot d) \cdot \varphi(n) \cdot k &= 1 \\ \Leftrightarrow e \cdot d &= k(p-1)(q-1) + 1 \end{aligned}$$

Den Term in der ersten Zeile formen wir um. $a \bmod b$ (Sprich: a modulo b) ergibt den Rest der Division von a und b (man spricht von der Modulo-Division). Das heißt, dass a mal b mal einer Zahl k gleich 1 ist. Nun wählen wir k so, dass es uns am Besten für d passt. Das heißt, dass k eine beliebige natürliche Zahl sein kann. Durch Einsetzen erhalten wir einen Wert für d .

Nun haben wir den *Public Key*, bestehend aus n und e , und den *Private Key* d und n ermittelt. Die Zahlen p , q , $\varphi(n)$ und k werden nicht mehr benötigt.

Hier ein Beispiel, das den Herstellungsprozess der Schlüssel veranschaulichen soll:

Beispiel: Der Einfachheit halber wählen wir nur zwei kleine Primzahlen $p = 11$ und $q = 17$. Wir berechnen $n = p \cdot q = 187$. Daraus ergibt sich für $\varphi(n) = (11-1) \cdot (17-1) = 160$. Jetzt wählen wir die Zahl e , die größer als 1 aber nicht größer als $\varphi(n)$ ist und deren größter Gemeinsamer Teiler mit $\varphi(n)$ gleich 1 ist. Dies gilt z.B. für $e = 7$. Nun widmen wir uns der Berechnung von d . Wir setzen bereits uns bekannte Größen in die Gleichung ein: $7d = k \cdot 160 + 1$. Durch ausprobieren erhalten wir relativ zügig, dass für $k = 1$, $d = 23$ ist. Schon haben wir alle Schlüssel ermittelt.

2.2 Ver- und Entschlüsselung

Nachdem wir nun die nötigen Schlüssel erzeugt haben, widmen wir uns der eigentlichen Arbeit des Algorithmus: Dem Ver- und Entschlüsseln. Der Einfachheit halber nehmen wir eine beliebige Zahl als zu übermittelnden Geheimtext. Im Kapitel 2.3 werden wir uns dann mit der Übermittlung von Texten befassen.

m sei der Klartext, den wir verschlüsseln wollen, und c der chiffrierte Geheimtext.

Das Verschlüsselungsverfahren funktioniert nun wie folgt:

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

Analog dazu lautet das Entschlüsselungsverfahren:

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

Wir können aus den Formeln ganz einfach ableiten, wie das RSA-Verfahren in der Praxis funktionieren wird. Ein Absender, der an den Empfänger, welcher als einziger im Besitz des *Public Keys*, d.h. im Besitz von d und n ist, eine Nachricht senden möchte, verschlüsselt diese mit dem öffentlichen Schlüssel und übermittelt sie. Auch wenn die Nachricht abgefangen würde und dem Angreifer der öffentliche Schlüssel, d.h. n und e bekannt wären, könnte dieser die Nachricht nicht entschlüsseln, weil er d nicht kennt.

Wir wissen aber, dass d aus n (genauer gesagt aus $\varphi(n)$) und e errechnet wird. Die Frage ist nun, ob der Angreifer diesen Vorgang nicht umkehren kann, um d zu errechnen. Und genau hier liegt bei RSA „der Hund begraben“: Die Sicherheit hängt damit zusammen, dass eine solche Rückrechnung eine erhebliche mathematische Schwierigkeit mit sich bringt. Das Problem liegt im immensen Rechenaufwand, sehr große Primzahlen zu faktorisieren. Mehr zu diesem Thema findet sich im Kapitel 3.1.

Wiederum wollen wir den Ver- und Entschlüsselungsprozess durch ein Beispiel veranschaulichen:

Beispiel: Im Beispiel des letzten Kapitels haben wir bereits die Schlüssel berechnet, welche wir jetzt verwenden wollen: $n = 187$, $e = 7$ und $d = 23$. Um den Vorgang nicht unnötig zu verkomplizieren wählen wir eine Zahlenfolge als Klartextnachricht aus: $m = 12\ 13\ 37\ 04$. Wir verschlüsseln jede Zahl mit der uns bekannte Formel: $c_1 = 12^7 \bmod 187 = 177$, $c_2 = 13^7 \bmod 187 = 106$, $c_3 = 37^7 \bmod 187 = 181$, $c_4 = 04^7 \bmod 187 = 115$. Jetzt setzen wir unsere Ergebnisse zur verschlüsselten Nachricht c zusammen: $c = 177\ 106\ 181\ 115$. Dann wollen wir unsere Geheimbotschaft wieder entschlüsseln: $m_1 = 177^{23} \bmod 187 = 12$, $m_2 = 106^{23} \bmod 187 = 13$, $m_3 = 181^{23} \bmod 187 = 37$, $m_4 = 115^{23} \bmod 187 = 4$. Und schon haben wir unseren originalen Klartext wieder ermittelt: $m = 12\ 13\ 37\ 04$.

2.2 Informationstechnologische Umsetzung mit JAVA

In diesem Abschnitt werden wir den Algorithmus in JAVA-Syntax bringen und ein funktionsfähiges RSA-Programm entwickeln.

Dem Programm soll der Benutzer den Klartext m und den öffentlichen Schlüssel n und e übergeben können, um den Text chiffrieren zu können. Natürlich soll auch eine Dechiffrierfunktion vorhanden sein, die durch Übergabe des Geheimtextes c und des privaten Schlüssels d und n funktioniert.

Wir definieren die folgenden Funktionen:

- `public String verschluesseln(String m, int N, int E)`
- `public String entschluesseln(String c, int D, int N)`
- `private void textInZahlUmwandeln(String text)`
- `private void zahlInTextUmwandeln(BigInteger[] zahlenArray)`

```
import java.lang.Math;
import java.math.BigInteger;

/**
 * Implementierung des RSA-Algorithmus
 * Bestandteil der Facharbeit von Daniel Dieckelmann
 *
 * Alle Rechte vorbehalten
 */
public class Rsa
```

```

{
    BigInteger n;
    BigInteger d;
    BigInteger e;

    BigInteger[] decMsgOriginal;
    BigInteger[] decMsgCodiert;

```

Die Tatsache, dass wir als Datentyp *BigInteger* und nicht etwa die bekannteren und einfacher zu handhabenden Typen *Integer* oder *Double* für die Schlüssel verwenden, hat mit einem internen Fehler in JAVA zu tun. Bei der Modulo-Division werden ab einer bestimmten Grenze hohe Zahlen (z.B. 40^{50}) falsch gerundet, so dass die Ergebnisse der Entschlüsselung nicht mehr korrekt sind. Deswegen muss der Umweg über *BigInteger* erfolgen.

Da es sich bei unseren „Geheimbotschaften“ um Strings, d.h. um echte Zeichenketten, und nicht nur um der Vereinfachung dienende Zahlen handeln soll, müssen wir eine Funktion programmieren, die aus dem Klartext *m* eine Zahlenfolge erstellt, die weiterhin aus der dann verschlüsselten Zahlenfolge den chiffrierten Geheimtext *c* macht, die diesen dann in wieder in Zahlen umwandeln kann und die aus den entschlüsselten Zahlen wieder den Klartext *m* erzeugt. Für die Umwandlung des Textes in Zeichen bedienen wir uns einfach der Tatsache, dass Buchstaben bzw. Textzeichen in der Informatik im Prinzip nichts anderes als Dezimalzahlen sind (stark vereinfacht erklärt). Wir nutzen den ASCII-Zeichensatz, um einzelne Zeichen (Datentyp *Char*) in Dezimalzahlen und umgekehrt umzuwandeln. Die umgekehrte Umwandlung funktioniert aber leider auf Grund einer weiteren JAVA-Eigenart nicht. Eigentlich benutzt JAVA zwar von sich aus den Unicode-Satz, der auch Zahlen größer als 128 (so viele Zahlen sind im ASCII-Satz enthalten) unterstützt, aber die BlueJ-Konsole zeigt bei solchen Zahlen anstatt der entsprechenden Symbole nur ein bestimmtes Zeichen an, dass beim Einlesen leider als die Zahl 0 interpretiert wird. Vermutlich kann BlueJ nur ASCII-Zeichen ausgeben. Auf Grund dieses Ausgabe- und Einleseproblems zwischen *ASCII* und *Unicode* ist der übergebene Wert für *C* bei *entschluesseln* nicht von Bedeutung.

```

// Macht aus einem Text die Zahlenfolge, die zum verschluesseln benoetigt
// wird.
private BigInteger[] textInZahlUmwandeln(String text)
{
    int i;
    char[] zeichenArray;
    BigInteger[] zahlenArray;

    // Kopiert den String in Zeichen zerlegt in ein Array
    zeichenArray = text.toCharArray();

```

```

    zahlenArray = new BigInteger[text.length()];
    for(i=0;i<text.length();i++)
    {
        // Umwandlung von Zeichen in Zahlen gemäß ASCII-Code
        zahlenArray[i] = new
BigInteger(String.valueOf((int)zeichenArray[i]));
    }

    return zahlenArray;
}

// Macht aus einer Zahlenfolge in einem Array einen Text mittels ASCII-
Codierung
private String zahlInTextUmwandeln(BigInteger[] zahlenArray)
{
    int i;
    char[] zeichenArray;
    String text;

    text = "";

    // Kopieren der Zahlen in ein Zeichen-Array und umwandeln der Zahlen in
ihr entsprechendes Zeichen
    for(i=0;i<zahlenArray.length;i++)
    {
        text = text + (char)zahlenArray[i].intValue();
    }

    return text;
}

```

Jetzt widmen wir uns der eigentlichen Arbeit unseres RSA-Programms: Dem Ver- und Entschlüsseln. Für die Funktion *verschluesseln* muss der Benutzer bereits zwei beliebige Primzahlen miteinander multipliziert haben (n) und auch die Zahl e ausgewählt haben. Selbstverständlich braucht er auch einen Klartext, der verschlüsselt werden soll. Analog dazu braucht der Benutzer für das *entschluesseln* den Geheimtext, welcher vom Programm ausgegeben wird und die Schlüssel n und d , welchen er bereits errechnet haben muss. Auf Grund des oben genannten Problems werden anstatt des Strings die Daten aus dem Array *decMsgCodiert* als Geheimnachricht benutzt.

```

public void verschluesseln(String m, int N, int E)
{
    int i;
    String text;

    // Umwandlung der Integer in BigInteger
    n = new BigInteger(String.valueOf(N));
    e = new BigInteger(String.valueOf(E));

    System.out.println("Demonstration des RSA-Verfahrens\nvon Daniel
Dieckelmann\n\nDies ist der Klartext: " + m + "\n");
}

```

```

// Umwandeln in Dezimalzahlen
decMsgOriginal = textInZahlUmwandeln(m);

text = "";
decMsgCodiert = new BigInteger[decMsgOriginal.length];

System.out.println("Klartext in Zahlen / Codierte Nachricht in
Zahlen");

// Jedes Zeichen einzeln verschlüsseln
for(i=0;i<decMsgOriginal.length;i++)
{
    System.out.print(decMsgOriginal[i] + " / ");
    // Aus JAVADocs: modPow: Returns a BigInteger whose value is
(this^exponent mod m)
    decMsgCodiert[i] = decMsgOriginal[i].modPow(e,n);
    System.out.print(decMsgCodiert[i] + "\n");
}

System.out.println("\nDies wäre der Geheimtext: \n" +
zahlInTextUmwandeln(decMsgCodiert));
}

public void entschluesseln(String c, int D, int N)
{
    int i;
    String text;

    /*
    * Funktioniert nicht wegen Unicode/ASCII-Umwandlung
    * decMsgCodiert = textInZahlUmwandeln(c);
    * Deswegen ist der Wert von c momentan irrelevant
    */

    // Umwandlung der Integer in BigInteger
    d = new BigInteger(String.valueOf(D));
    n = new BigInteger(String.valueOf(N));

    text = "";
    decMsgOriginal = new BigInteger[decMsgCodiert.length];

    System.out.println("\nCodierte Nachricht in Zahlen / Encodierte
Nachricht in Zahlen");

    // Jedes Zeichen entschlüsseln
    for(i=0;i<decMsgCodiert.length;i++)
    {
        System.out.print(decMsgCodiert[i] + " / ");
        decMsgOriginal[i] = decMsgCodiert[i].modPow(d,n);
        System.out.print(decMsgOriginal[i] + "\n");
    }

    System.out.println("\nEncodierter Klartext: " +
zahlInTextUmwandeln(decMsgOriginal));
}
}

```

Das Programm funktioniert nun so, dass zunächst der Benutzer die Funktion *verschluesseln* aufruft und dort den Klartext und den *Public Key* eingibt. Dann ruft er die Funktion *entschluesseln* auf, gibt dann n und d ein und gäbe dann den Geheimtext ein. Letzteres braucht er nicht, das erledigt aus bekanntem Grund das Programm für ihn. Dann kann der Benutzer z.B. im BlueJ-Terminal nachvollziehen, wie die Verschlüsselung abläuft. Am Ende wird er wieder seinen eingegebenen Klartext erhalten.

3. Anwendung des Algorithmus

3.1 Sicherheitsaspekte

In den obigen Kapiteln wurde bereits erwähnt, dass die Sicherheit von RSA stark mit den Größen der verwendeten Primzahlen p und q und der mathematischen Schwierigkeit der Zerlegung von n in seine beiden Faktoren abhängt. Ein Hacker, welcher die öffentlichen Schlüssel kennt, braucht bekanntlich noch den Schlüssel d um eine abgefangene Nachricht zu entschlüsseln. Für dessen Berechnung braucht er aber $\varphi(n)$ oder den Wert für $(p-1) \cdot (q-1)$. Und deswegen muss er n in seine beiden Primfaktoren zerlegen, um die Nachricht lesen zu können. Bei kleinen Zahlen, wie in unseren Beispielen scheint dies ja noch sehr einfach zu sein, da aber im realen Gebrauch mindestens Zahlen mit einer Stärke von 512 Bit (das entspricht einer 128-stelligen Zahl (!)) empfohlen werden, wird der Rechenaufwand bei einer herkömmlichen Faktorisierung so immens, dass ein Knacken der Verschlüsselung auf diesem Weg prinzipiell unmöglich ist. Das Faktorisieren funktioniert im Prinzip so, dass man alle Zahlen, die kleiner als n sind, daraufhin überprüft, ob sie Teiler von n sind oder nicht. Man könnte sich zwar in unserem Falle auf Primzahlen und auch nur auf alle Zahlen die kleiner als \sqrt{n} (die Wurzel aus n würde schon alle Möglichkeiten abdecken, da bereits $\sqrt{n} \cdot \sqrt{n} = n$ ist. \sqrt{n} ist der größtmögliche Wert für einen der Faktoren.) sind beschränken, aber das hieße immer noch, dass man bei einer 128-stelligen Zahl (maximaler Wert: 10^{128}) im schlimmsten Falle alle Primzahlen zwischen 2 und 10^{64} (64-stellige Zahl) durchgehen müsste.² Angesichts solcher Zahlen können wir uns ein ungefähres Bild von der Unmöglichkeit dieses Unterfanges machen.

² Beispiel entliehen aus [2], Zahlenwerte abgewandelt

Allerdings ist bis heute nicht geklärt, ob es nicht einen weniger aufwendigeren Weg zur Faktorzerlegung gibt. Man kann also nicht mit Sicherheit sagen, dass der Algorithmus 100%ig sicher ist, denn theoretisch könnte jederzeit ein Mathematiker ein neues revolutionäres Verfahren entdecken, das den RSA-Algorithmus sofort unbrauchbar machen würde. Es ist in keiner Weise abzusehen, ob ein solches Verfahren überhaupt existieren könnte oder ob es in nächster Zeit entdeckt werden könnte.

Der große Vorteil des RSA-Algorithmus besteht aber nicht nur in seiner praktischen „Unbesiegbarkeit“ sondern viel eher noch in der Tatsache, dass für seinen Gebrauch kein Schlüsselaustausch, wie bei herkömmlichen symmetrischen Verfahren, nötig ist. Die Übertragung des Geheimschlüssels an den Empfänger einer Nachricht stellte schon immer das größte Problem bei klassischen Verschlüsselungssystemen dar.

Allerdings muss hier auch auf die potenziellen Risiken bei der Verwendung von RSA hingewiesen werden. So stellt zum Beispiel die Faktorisierung von n für einen Quantencomputer kein Problem dar. Allerdings gibt es bis heute weder einen schnellen Faktorisierungsalgorithmus noch einen ausreichend effizienten Quantencomputer. Doch wann der Zeitpunkt kommen wird, ab dem RSA unbrauchbar wird, ist nicht abzuschätzen. Das könnte morgen früh oder in 50 Jahren oder überhaupt nicht der Fall sein. Deswegen arbeiten einige Wissenschaftler bereits an neuen Public-Key-Verfahren, die gegen die Vorteile eines Quantencomputers (Er kann mit Werten, die gleichzeitig 1 und 0 sind rechnen - das ermöglicht mehrere Reoperationen gleichzeitig) gewappnet sind. Eine genauere Erklärung zu der Funktionalität von Quantencomputern und den angesprochenen Verfahren sind in [6] zu finden.

3.2 Tatsächliche Anwendungsgebiete

Aus verschiedenen Gründen wird der RSA-Algorithmus in der Praxis aber nicht zum Verschlüsseln größerer Datenmengen, Texte, etc. verwendet. Vor allem weil der Ver- und Entschlüsselungsprozess bei den großen Zahlen, die Sicherheit gewährleisten, sehr rechen- und zeitaufwendig ist. Daher verwendet man RSA in der Praxis oft dazu, einen Schlüssel selbst zu verschlüsseln, um diesen sicher an einen Empfänger zu übergeben und diesen Dann für ein wesentlich schnelleres symmetrisches Verfahren zu verwenden.

Ein typisches Anwendungsgebiet für diese Praxis ist die Verschlüsselung der PIN bei Bankkonten. Die eingegebene PIN wird via RSA am Bankautomaten verschlüsselt und an die Rechenzentrale der Bank gesendet, wo man mit Hilfe des *Private Keys* die PIN entschlüsseln und überprüfen kann (Man spricht von einer Signatur-Überprüfung). Wenn die PIN korrekt ist kann jetzt z.B. ein symmetrisches Verfahren angewendet werden, um den Geld- und Datentransfer abzusichern.

Ein anderes Beispiel für die Anwendung des RSA-Kryptosystems sind verschlüsselte Internetverbindungen via SSL. Bei Webseiten, deren URL mit einem `https://` beginnt, besteht eine verschlüsselte Verbindung zwischen Browser und Server. Auch hier wird RSA dazu verwendet den Schlüssel für die symmetrische Datenverschlüsselung sicher zu übertragen. Die ganze Webseite oder Daten, die z.B. über ein Formular verschickt werden, werden mit einer anderen Technik kodiert. Oft verwendet man den *Advanced Encryption Standard* (AES). Der nötige AES-Schlüssel wird vom Browser erzeugt und mittels RSA-Kodierung an den Server gesendet. Den *Public Key* erhält der Browser in einem so genannten Zertifikat vom Server. Nur der Server „kennt“ den *Private Key* und kann den zu übertragenen AES-Schlüssel ermitteln. Jetzt kann die gesicherte Datenübertragung beginnen. Wenn ein Benutzer eine so gesicherte Internetseite besucht (beispielsweise `https://www.ccc.de/`), wird er von den meisten Browsers darüber informiert, dass er eine gesicherte Verbindung aufbauen möchte und ob er das Zertifikat anerkennt. Hier kann er sich allerhand Informationen über das Zertifikat einholen, unter anderem auch den *Public Key* einsehen.

Bekannte Anwendungsgebiete für das RSA-Verfahren sind:[5]

- Internet- und Telefonie-Infrastruktur: X.509-Zertifikate
- Übertragungs-Protokolle: IPsec, TLS, SSL, SSH, WASTE
- E-Mail-Verschlüsselung: PGP, S/MIME
- Authentifizierung französischer Telefonkarten
- Kartenzahlung: EMV

4. Diskussion

In diesem Kapitel seien noch die stichhaltigsten Pro- und Contraargumente für bzw. gegen das RSA-Kryptosystem aufgeführt. Denn der RSA-Algorithmus wurde und wird auch immer noch in der Wissenschaft kontrovers diskutiert.

Pro

- Public Key Verfahren: Es muss kein Schlüssel übertragen werden, was die Sicherheit des Verfahrens gegenüber herkömmlichen symmetrischen Algorithmen deutlich erhöht.
- Praktische Unmöglichkeit der Fremdenschlüsselung: Durch die sehr großen verwendeten Zahlen, ist kein Computerverbund der Welt in der Lage, auf herkömmlichen Weg den *Private Key* anhand des *Public Keys* zu ermitteln.
- Einfacher Algorithmus: Der Rechenweg zu Erstellung der Schlüssel und zum Ver- und Entschlüsseln von Nachrichten ist an sich sehr simpel und unkompliziert. Das vereinfacht eine Implementierung erheblich.

Contra

- Keine 100%ige Sicherheit: Es ist ungewiss, ob das RSA-Verfahren auch in Zukunft sicher bleibt. Sollte dies nicht der Fall sein, bedeutete dies, dass unsere ganzen Internet- und Bankensicherheitswesen umgestellt werden müsste.
- Rechenintensives Verfahren: Der Ver- und Entschlüsselungsprozess nimmt bei in der Praxis gebräuchlichen Datenmengen auf Grund der sehr großen Zahlen und vielen Einzeloperationen sehr viel Zeit und Rechenaufwand in Anspruch.
- Anfällig: Es gibt verschiedene Angriffstechniken, die Schwachstellen in RSA ausnutzen, um die Ermittlung von d effizienter zu gestalten. Z.B. kann man eine Bank-PIN mittels Durchprobieren (sog. Brute-Force-Hacking) ermitteln. Wenn bei einer PIN der Chiffretext mit m übereinstimmt, weiß der Angreifer, dass er die richtige PIN gefunden hat.³

³ Beispiel stammt aus [6] Absatz *Wieso ist RSA sicher?*

Literaturverzeichnis

- [1] Beutelspacher, Albrecht; *Kryptologie - Eine Einführung in die Wissenschaft vom Verschlüsseln, Verbergen und Verheimlichen*; 6., überarbeitete Auflage; Vieweg; Braunschweig/Wiesbaden 2002
- [2] Buchmann, Johannes; *Einführung in die Kryptographie*; 2., erweiterte Auflage; Springer; Berlin, Heidelberg, New York 2001
- [3] Ley, Reinhold und Verlagsredaktion; *Praktische Informatik mit JAVA*; 1. Auflage; Cornelsen; Berlin 2002
- [4] Ge, Yimin; *Die Mathematik von RSA*; Seiten 15-20; August 2005; http://yimin.sinuslab.net/doc/mathematik_von_rsa.pdf (abgerufen 29.12.06)
- [5] Verschiedene; *RSA-Kryptosystem*; Wikipedia Foundation; 2007; <http://de.wikipedia.org/wiki/RSA-Kryptosystem> (zuletzt abgerufen 02.03.07)
- [6] Buchmann, Johannes und Takagi, Tsuyoshi; *Kryptographie - Chancen und Risiken*; November 2003; http://www.cdc.informatik.zu-darmstadt.de/reports/TR/TI-03-06.thema_Forschung.pdf (abgerufen 17.02.07)
- [7] Janssen, Lukas; *Anwendung und Hintergründe des RSA-Verfahrens*; Januar 2004; <http://www.oberstufen-informatik.de/krypto/rsajanssen.pdf> (abgerufen 29.12.06)