FACHHOCHSCHULE WEDEL

Seminar IT-Sicherheit

Sommersemester 2016

CRYPTANALYTIC ATTACKS ON RSA

Integer Factorization Attacks

Referent:

Alina Claussen

minf100579@fh-wedel.de

Betreuer:

Prof. Dr. Gerd Beuster

gb@fh-wedel.de

eingereicht am: 05.06.2016

Inhaltsverzeichnis

1. Einleitung

Die Sicherheit von dem Verschlüsselungsverfahren RSA basiert auf der Schwierigkeit das Faktorisierungsproblem in angemessener Zeit zu lösen. Diese Ausarbeitung geht auf zwei grundlegende Algorithmen RSA durch Faktorisierung zu entschlüsseln ein. Es ist nicht das Ziel effiziente und erfolgreiche Methoden zu finden, sondern die grundlegenden Ansätze und Methoden zu erläutern, die zu komplexeren führen.

1.1. Mathematische Grundlagen

Zunächst müssen erstmal ein paar mathematische Grundlagen erläutert werden, damit das weitere Verständnis gewährleistet ist.

1.1.1. Kongruenz [2]

Die Kongruenz ist elementar für das Verschlüsselungsverfahren RSA. Sie wird im Folgenden definiert.

$a\ und\ b\ \in\ \mathbb{Z}$ sind kongruent zueinander, wenn sie modulo $m\ \in\ \mathbb{N}$ das gleiche Ergebnis haben. Die Schreibweise ist wie folgt:

$$a\ \equiv b\ (mod\ m),$$

Aus dieser Kongruenz folgt, dass m die Differenz $(a - b)$ teilt, sowie, dass a und b bei einer Division durch m den gleichen Rest lassen.

1.1.2. Euler'sche φ-Funktion [2][5]

Die Euler'sche φ-Funktion ordnet der Zahl m die Anzahl aller Zahlen zwischen 1 und m zu, die zu m teilerfremd sind. Teilerfremd sind zwei Zahlen, wenn ihr größter gemeinsamer Teil 1 ist.

Die Euler'sche φ-Funktion definiert den Rückgabewert für $m\ \in\ \mathbb{N}$ wie folgt:

$$\varphi(m) = \#\{a\ \in\ \mathbb{Z}\ |\ 1\ \leq a\ \leq m, ggT(a,m) = 1\}$$

Wenn m aus dem Produkt der beiden Faktoren $m_1, m_2\ \in\ \mathbb{N}$ besteht und diese beiden Faktoren teilerfremd sind, gilt:

$$\varphi(m_1 m_2) =\ \varphi(m_1)\varphi(m_2)$$

Für Primzahlen gilt:

$$\varphi(p) = p - 1$$

1.1.3. RSA allgemein [1][3]

RSA, benannt nach seinen Erfindern Rivest, Shamir und Adleman, ist eins der meist genutzten Verschlüsselungsverfahren. Es ist das erste funktionsfähige kryptographische Public-Key-System. Es wurde im Jahr 1977 erfunden und 1978 veröffentlicht.

Die grundlegende Idee der Verschlüsselungstechnik ist die, dass es zeitlich viel zu lange dauern würde den chiffrierten Text zu entschlüsseln. Für das Verfahren werden zwei Schlüssel erzeugt. Möchte Teilnehmer A Teilnehmer B eine verschlüsselte Nachricht schicken, muss er Teilnehmer B nach einem öffentlichen Schlüssel fragen mit dem er dann die Nachricht verschlüsseln kann. Der öffentliche Schlüssel ist öffentlich, daher muss dieser nicht verschlüsselt werden. Die Sicherheit von RSA ist dadurch nicht gefährdet. Der öffentliche Schlüssel besteht aus dem Zahlenpaar (e, N). Teilnehmer A verschlüsselt seine Nachricht M nach der Vorschrift $C \equiv M^e \ (mod\ N)$. Dabei ist C der Chiffretext, e der Verschlüsselungsexponent und $N = pq$. p und q sind zwei möglichst große Primzahlen um die Sicherheit zu erhöhen. Allerdings ist Teilnehmer A nur N bekannt und nicht p und q.

Die verschlüsselte Nachricht C ist ohne den privaten Schlüssel nicht in angemessener Zeit zu entschlüsseln. Der private Schlüssel besteht aus dem Zahlenpaar (d, N). Dieser Schlüssel behält Teilnehmer B für sich. Die Formel für die Entschlüsselung sieht wie folgt aus: $M \equiv C^d \ (mod\ N)$. Wobei d der Entschlüsselungsexponent ist.

Nun stellt sich die Frage, wie diese Schlüssel erzeugt werden. Zunächst werden zwei ungleiche Primzahlen gewählt. Diese Primzahlen werden im Folgenden p und q genannt. Mit diesen Primzahlen wird die Zahl N definiert: $N = pq$. N wird auch das RSA-Modul genannt. Danach wird die Euler'sche φ-Funktion von N wie folgt bestimmt:

$$\varphi(N) = \varphi(pq) = \varphi(p)\varphi(q) = (p-1)(q-1)$$

Der Verschlüsselungsexponent e aus dem öffentlichen Schlüssel wird so gewählt, dass er teilerfremd zu $\varphi(N)$ ist und $1 < e < \varphi(N)$ gilt. Damit wäre der öffentliche Schlüssel definiert.

Für den privaten Schlüssel wird nun der Entschlüsselungsexponent d als multiplikatives Inverses von e bezüglich des Moduls $\varphi(N)$ festgelegt, das heißt folgende Kongruenz soll gelten:

$$ed \equiv 1 \ (mod\ \varphi(N))$$

Damit wäre auch der private Schlüssel definiert.

1.1.4. Faktorisierungsproblem

Das Faktorisierungsproblem beschreibt das Problem, dass es bei einer gegebenen Zahl kein effizienter Algorithmus bekannt ist die Faktoren zu ermitteln. Dabei ist aber auch noch nicht bewiesen, dass es keinen Algorithmus mit polynomialer Laufzeit gibt. Es ist sogar noch nicht einmal bekannt welcher Komplexitätsklasse das Faktorisierungsproblem zugeordnet werden kann.

Wäre ein Algorithmus mit polynomialer Laufzeit bekannt, der das Faktorisierungsproblem lösen würde, könnte RSA auch in polynomialer Laufzeit entschlüsselt werden, das heißt RSA wäre nicht mehr sicher genug.

1.2. Warum Integer Factorization Attacks? [1]

Einer der intuitivsten Ansätze RSA zu entschlüsseln wäre den privaten Schlüssel zu ermitteln. Dieser besteht aus dem Zahlenpaar (d, N). Da N in dem öffentlichen Schlüssel, bestehend aus dem Zahlenpaar (e, N), ebenfalls enthalten ist, muss nur noch der Entschlüsselungsexponent d ermittelt werden. d ist laut seiner Definition das multiplikative Inverse von dem Verschlüsselungsexponenten e bezüglich des Moduls $\varphi(N)$. Dabei sollte folgende Kongruenz gelten: $ed \equiv 1\ (mod\ \varphi(N))$.

Da e Teil des öffentlichen Schlüssels ist, fehlt zur Berechnung der Gleichung noch $\varphi(N)$. Wie bereits gezeigt ist $\varphi(N) = (p - 1)(q - 1)$. Zur Berechnung von $\varphi(N)$ werden p und q benötigt, die ebenfalls nicht bekannt sind. Da bekannt ist, dass $N = pq$ ist, wäre die direkteste Methode zu entschlüsseln N zu faktorisieren.

Anmerkung: Neuere Recherchen vermuten, dass RSA zu entschlüsseln leichter ist als das Faktorisierungsproblem zu lösen.

1.3. Zwei Hauptkategorien nach Laufzeiten [1]

Viele Algorithmen zur Faktorisierung von N können in zwei Hauptkategorien eingeteilt werden. Die Laufzeit ist entweder von der Größe von N oder von p abhängig. In der Praxis sind Algorithmen aus beiden Kategorien nützlich. Allerdings sind auch andere Kategorien denkbar, zum Beispiel deterministisch und nicht deterministisch oder konditional und nicht konditional.

2. Fermat Factoring Attack [1][4][7]

Fermat Factoring Attack wurde das erste Mal von Fermat 1643 beschrieben. Der Algorithmus ist Grundlage für weiterer Algorithmen. Der Algorithmus ist nur effizient, wenn die Differenz von p und q kleiner ist als $N^{1/4}$.

2.1. Herleitung

Bei dem Fermat Factoring Attack wird zunächst eine Annahme gemacht.

[1] $N = x^2 - y^2$

Durch die Anwendung der dritten Binomischen Formel ergibt sich:

[2] $N = (x + y)(x - y)$

Da bekannt ist, dass N das Produkt zweier Primzahlen p und q ist, kann hier gleich gesetzt werden.

[3] $q = x + y$

[4] $p = x - y$

Dabei gilt $x > y$, da ansonsten q negative Werte annehmen könnte und $p \leq q$.

Um Gleichungen für x und y zu erhalten werden folgende Umformungen gemacht. Die Gleichung [3] wird nach x aufgelöst:

[5] $q = x + y \Leftrightarrow x = q - y$

Nun kann Gleichung [5] in Gleichung [4] eingesetzt werden:

[6] $p = (q - y) - y \Leftrightarrow p = q - 2y \Leftrightarrow 2y = q - p \Leftrightarrow y = \frac{q-p}{2}$

Durch das Einsetzen von Gleichung [6] in Gleichung [5] wird x definiert:

[7] $x = q - \frac{q-p}{2} \Leftrightarrow x = \frac{2q-(q-p)}{2} \Leftrightarrow x = \frac{2q-q+p}{2} \Leftrightarrow x = \frac{q+p}{2}$

Aus Gleichung [1] geht außerdem hervor, wenn eine Quadratzahl y gefunden wird, sodass $y^2 = x^2 - N$ gilt, könnte aus den ermittelten Werten für x und y p und q errechnet werden. Das heißt, die naivste Variante wäre für x alle möglichen Werte einzusetzen und zu testen, ob y eine Quadratzahl ist.

Das Intervall, das die zu testenden Werte enthält, sollte dennoch nicht einfach $[1, \dots, N]$ sein, eine Einschränkung wäre sinnvoll. Der Wert für x ist am kleinsten, wenn die Differenz der Primzahlen p und q möglichst klein ist. Dies ist der Fall, wenn $p = q = \sqrt{N}$ ist. Durch Einsetzen in Gleichung [7] erhält man:

$$[8]\ x_{min} = \frac{\sqrt{N}+\sqrt{N}}{2} = \frac{2\sqrt{N}}{2} = \sqrt{N}$$

Für den größtmöglichen Wert für x gibt es verschiedene Ansätze. Der erste Ansatz bestimmt x_{max} auf die Hälfte von N.

$$[9]\ x_{max} = \frac{N}{2}$$

Ein anderer Ansatz besagt, dass die kleinstmögliche Primzahl für p 3 ist. 2 wird nicht berücksichtigt, da es eine gerade Zahl handelt. Daraus ergibt sich für q:

$$[10]\ N = 3q \iff q = \frac{N}{3}$$

p und q eingesetzt in Gleichung [7] ergibt:

$$[11]\ x_{max} = \frac{\frac{N}{3}+3}{2} = \frac{N+9}{6}$$

Im Weiteren dieser Ausarbeitung wird der erste Ansatz verfolgt.

2.2. Algorithmus

Der Algorithmus hat als einzigen Input das RSA-Modul N. Das Ziel ist es p und q zu ermitteln. Zunächst wird eine Variable x mit $\lfloor\sqrt{N}\rfloor + 1$ initialisiert. Gesucht wird nun eine Quadratzahl, deren Wurzel y wäre.

Nun wird getestet, ob $x^2 - N$ eine Quadratzahl ist.

Wenn ja, ist der Algorithmus beendet und die Ergebnisse wären:

$$y = \sqrt{x^2 - N}$$

$$p = x - y$$

$$q = x + y$$

Wenn $x^2 - N$ keine Quadratzahl ist, wird x um 1 erhöht und erneut getestet, ob eine Quadratzahl vorliegt. Das geht solange bis $x < \frac{N}{2}$ oder p und q gefunden wurden.

2.3. Alternativer Algorithmus

In dem alternativen Algorithmus ist vieles identisch zu dem vorherigen Algorithmus. Der Ansatz, dass x in jedem Schleifendurchgang um 1 inkrementiert wird, bleibt ebenfalls gleich. Es werden die Werte für die Quadratzahl (im Folgenden $y2$ genannt) in jedem Schleifendurchgang analysiert. x wird wie bereits im vorherigen Algorithmus mit $\lfloor\sqrt{N}\rfloor + 1$ initialisiert.

In der folgenden Tabelle wird $y2$ in dem jeweiligen Schleifendurchlauf i berechnet.

$\boldsymbol{i}$	$\boldsymbol{x_i}$	$\boldsymbol{y2_i}$
0	x	$\boldsymbol{x^2 - N}$
1	$x+1$	${x_1}^2 - N = (x+1)^2 - N = \boldsymbol{x^2 + 2x + 1 - N}$
2	$x_1 + 1$	${x_2}^2 - N = (x_1+1)^2 - N = {x_1}^2 + 2x_1 + 1 - N = {x_1}^2 + 2(x+1) + 1 - N$ $= \boldsymbol{{x_1}^2 + 2x + 3 - N}$
3	$x_2 + 1$	${x_3}^2 - N = (x_2+1)^2 - N = {x_2}^2 + 2x_2 + 1 - N$ $= {x_2}^2 + 2(x_1+1) + 1 - N = {x_2}^2 + 2x_1 + 3 - N$ $= {x_2}^2 + 2(x+1) + 3 - N = \boldsymbol{{x_2}^2 + 2x + 5 - N}$

Aus dieser Tabelle wird deutlich, dass ein Teil jeder Gleichung jeweils dem vorherigem x entspricht. Bei jedem Schleifendurchlauf wird zudem zweimal dem ursprünglichem x addiert. Übrig bleibt die Zahlenfolge $1, 3, 5, \ldots$. Dafür wird eine zusätzliche Variable d deklariert, die mit 1 initialisiert wird und bei jedem Schleifendurchlauf um 2 erhöht wird.

2.4. Beweis auf Gleichheit der Algorithmen

Die beiden aufgeführten Varianten des Algorithmus sind optisch zwar sehr unterschiedlich, aber bewirken doch das Gleiche. Um zu beweisen, dass die beiden Algorithmen das gleiche Ergebnis liefern, sind vor allem die Initialisierung und der Code in der Schleife wichtig.

	Algorithmus 1	**Algorithmus 2**
Initialisierung	$x = \lfloor\sqrt{N}\rfloor + 1$	$x = \lfloor\sqrt{N}\rfloor + 1$ $d = 1$
Schleifencode	$x = x + 1$ $y2 = x^2 - N$	$y2 = y2 + 2x + d$ $d = d + 2$

Zunächst werden für die zwei Algorithmen jeweils eine allgemeine Formel gesucht, die das Ergebnis jedes Schleifendurchlaufs beschreibt. Durch Zeigen, dass die Formeln gleich sind, wird die Gleichheit der Algorithmen bewiesen.

Für Algorithmus 1, i ist die Anzahl der Schleifendurchläufe:

$\boldsymbol{i}$	$\boldsymbol{x_i}$	$\boldsymbol{y2_i}$
0	$\lfloor\sqrt{N}\rfloor + 1$	$(\lfloor\sqrt{N}\rfloor + 1)^2 - N$
1	$\lfloor\sqrt{N}\rfloor + 2$	$(\lfloor\sqrt{N}\rfloor + 2)^2 - N$
2	$\lfloor\sqrt{N}\rfloor + 3$	$(\lfloor\sqrt{N}\rfloor + 3)^2 - N$
3	$\lfloor\sqrt{N}\rfloor + 4$	$(\lfloor\sqrt{N}\rfloor + 4)^2 - N$
i	$\lfloor\sqrt{N}\rfloor + 1 + i$	$(\lfloor\sqrt{N}\rfloor + (i+1))^2 - N$

Daraus ergibt sich, dass folgende Formel für $y2$ für den i.ten Schleifendurchlauf gilt:

$$y2_i = \left(\lfloor\sqrt{N}\rfloor + (i+1)\right)^2 - N = \lfloor\sqrt{N}\rfloor^2 + 2\lfloor\sqrt{N}\rfloor(i+1) + (i+1)^2 - N$$
$$= \lfloor\sqrt{N}\rfloor^2 + 2i\lfloor\sqrt{N}\rfloor + 2\lfloor\sqrt{N}\rfloor + i^2 + 2i + 1 - N$$

Das gleiche wird für den zweiten Algorithmus gemacht.

$\boldsymbol{i}$	$\boldsymbol{y2_i}$	$\boldsymbol{d_i}$
0	$(\lfloor\sqrt{N}\rfloor + 1)^2 - N$	1
1	$(\lfloor\sqrt{N}\rfloor + 1)^2 - N + 2(\lfloor\sqrt{N}\rfloor + 1) + 1$	3
2	$(\lfloor\sqrt{N}\rfloor + 1)^2 - N + 2(\lfloor\sqrt{N}\rfloor + 1) + 1 + 2(\lfloor\sqrt{N}\rfloor + 1) + 3$	5
3	$(\lfloor\sqrt{N}\rfloor + 1)^2 - N + 2(\lfloor\sqrt{N}\rfloor + 1) + 1 + 2(\lfloor\sqrt{N}\rfloor + 1) + 3$ $+2(\lfloor\sqrt{N}\rfloor + 1) + 5$	7
i	$(\lfloor\sqrt{N}\rfloor + 1)^2 - N + 2i(\lfloor\sqrt{N}\rfloor + 1) + \sum_{k=1}^{i}(1 + 2(k-1))$	$1 + 2i$

Um auf die gleiche Gleichung des ersten Algorithmus zu kommen, muss die Gleichung umgeformt werden.

$$(\lfloor\sqrt{N}\rfloor + 1)^2 - N + 2i(\lfloor\sqrt{N}\rfloor + 1) + \sum_{k=1}^{i}(1 + 2(k-1))$$

$$= (\lfloor\sqrt{N}\rfloor + 1)^2 - N + 2i(\lfloor\sqrt{N}\rfloor + 1) + \sum_{k=1}^{i} 1 + \sum_{k=1}^{i} 2(k-1)$$

$$= (\lfloor\sqrt{N}\rfloor + 1)^2 - N + 2i(\lfloor\sqrt{N}\rfloor + 1) + i + 2\sum_{k=1}^{i} k - 1$$

$$= (\lfloor\sqrt{N}\rfloor + 1)^2 - N + 2i(\lfloor\sqrt{N}\rfloor + 1) + i + 2\sum_{k=1}^{i} k - 2\sum_{k=1}^{i} 1$$

$$= (\lfloor\sqrt{N}\rfloor + 1)^2 - N + 2i(\lfloor\sqrt{N}\rfloor + 1) + i + 2\sum_{k=1}^{i} (k) - 2i$$

Für die Summe kann nun die Gaußesche Summenformel eingesetzt werden.

$$= (\lfloor\sqrt{N}\rfloor + 1)^2 - N + 2i(\lfloor\sqrt{N}\rfloor + 1) + i + 2\frac{i^2 + i}{2} - 2i$$

$$= (\lfloor\sqrt{N}\rfloor + 1)^2 - N + 2i(\lfloor\sqrt{N}\rfloor + 1) + i^2$$

$$= \lfloor\sqrt{N}\rfloor^2 + 2\lfloor\sqrt{N}\rfloor + 1 - N + 2i\lfloor\sqrt{N}\rfloor + 2i + i^2$$

Somit würde die Gleichheit der Algorithmen bewiesen.

2.5. Pseudocode

$x \leftarrow \lfloor\sqrt{N}\rfloor + 1$

$y2 \leftarrow x * x - N$

$d \leftarrow 1$

do {

if $\lfloor\sqrt{y2}\rfloor = \sqrt{y2}$ {

$X \leftarrow \sqrt{N + y2}$

$Y \leftarrow \sqrt{y2}$

Print $X - Y$ **und** $X + Y$

Beende Algorithmus

}

$y2 \leftarrow y2 + 2x + d$

$d \leftarrow d + 2$

} **while** $\lfloor\sqrt{y2}\rfloor < \frac{N}{2}$

2.6. Beispielprogramm

Im Folgenden wurde der Fermat Factoring Attack in der zweiten Variante in der Programmiersprache C implementiert.

```
/** Fermat Factoring Attack */
#include <stdio.h>
#include <math.h>

void step3( int N, double * x, double * y2, double * d );
void step4( int N, double * x, double * y2, double * d );

/**
 * Fuehrt die hauptsaechliche Rechnung aus
 * Berechnet neue Quadratzahl
 *
 * @param N RSA-Modul
 * @param *x Startwert des Algorithmus
 * @param *y2 aktueller Wert der Quadratzahl
 * @param *d Laufvariable
 */
void step2( int N, double * x, double * y2, double * d ) {
      if ( ceil( sqrt( (*y2) ) ) == sqrt( (*y2) ) )
            step4( N, x, y2, d );
      else {
            (*y2) = (*y2) + 2 * (*x) + (*d);
            (*d) = (*d) + 2;
            step3( N, x, y2, d );
      }
}

/**
 * Prueft, ob der Algorithmus beendet werden soll
 *
 * @param N RSA-Modul
 * @param *x Startwert des Algorithmus
 * @param *y2 aktueller Wert der Quadratzahl
 * @param *d Laufvariable
 */
void step3( int N, double * x, double * y2, double * d ) {
      if ( ceil( sqrt( (*y2) ) ) < N / 2 )
            step2( N, x, y2, d );
      else
            printf("No Factor Found\n");
}

/**
 * Berechnet die Faktoren und gibt diese aus
 *
 * @param N RSA-Modul
 * @param *x Startwert des Algorithmus
 * @param *y2 aktueller Wert der Quadratzahl
 * @param *d Laufvariable
 */
void step4( int N, double * x, double * y2, double * d ) {
      int X = sqrt( N + (*y2) );
      int Y = sqrt( (*y2) );
      printf("Factors:  %i  %i\n", X - Y, X + Y );
}
```

```
/**
 * Hauptprogramm
 * Initialisiert und leitet den Fermat Factoring Attack ein
 */
int main ( int argc, char * argv [] )
{
	int N;
	double x, y2, d;

	N = atoi( argv[1] );

	x = floor( sqrt( (double) N ) ) + 1.0;
	y2 = x * x - (double) N;
	d = 1;

	step2( N, &x, &y2, &d );

	return 0;
}
```

2.7. Beispiel

Im Weiteren wird der Fermat Factoring Attack an einem Beispiel gezeigt.

Gegeben ist ein $N = 377$.

Initialisierung:

$x = \lfloor \sqrt{N} \rfloor + 1 = 20$

$y2 = x * x - N = 23$

$d = 1$

Erster Schleifendurchlauf:

Da 23 keine Quadratzahl ist, werden $y2$ und d neu gesetzt.

$y2 = y2 + 2x + d = 64$

$d = d + 2 = 3$

Zweiter Schleifendurchlauf:

64 ist eine Quadratzahl, daher können die Faktoren berechnet werden.

$X = \sqrt{N + y2} = 21$

$Y = \sqrt{y2} = 8$

Damit wäre $p = X - Y = 13$ und $q = X + Y = 29$

3. Pollard's „p-1" Factoring Algorithm [1][6]

Pollard's „p-1" Factoring Algorithm kann p und q effizient ermitteln, wenn p und q beide kleiner als 10^{20} sind.

3.1. Satz von Euler/Fermat

Pollard's „p-1" Factoring Algorithm basiert auf dem kleinen Satz von Fermat.

Der kleine Satz von Fermat stellt eine Eigenschaft einer Primzahl da. Er hat zwei Formen:

[1] $a^p \equiv a \ (mod\ p)$

[2] $a^{p-1} \equiv 1 \ (mod\ p)$

Dabei gilt $p \in primes, a \in \mathbb{Z}$ mit $p \nmid a$

Die für den „p-1" Algorithmus relevante Variante ist die [2]. Sie ist aus dem Satz von Euler herzuleiten.

Der Satz von Euler besagt:

[3] $a^{\varphi(m)} \equiv 1 \ (mod\ m), m \in \mathbb{N}, a \in \mathbb{Z}\ mit\ ggT(a,m) = 1$

Da in diesem Fall m eine Primzahl ist, ist $\varphi(m) = (m-1)$. Wenn $\varphi(m)$ mit $m-1$ ersetzt wird, ergibt sich der kleinen Satz von Fermat.

3.2. Herleitung

Für Pollard's „p-1" Factoring Algorithm wird der kleine Satz von Fermat verallgemeinert.

$a^m \equiv 1 \ (mod\ p)$

Dabei ist m ein Vielfaches von $p-1$. Aus dieser Kongruenz wird deutlich, dass p $(a^m - 1)$ teilt. Das heißt, dass p auch den $ggT(a^m - 1,\ N)$ teilt.

3.3. Algorithmus

Der Algorithmus nutzt diese Eigenschaften aus und versucht eine Zahl f zu ermitteln, die ein nicht trivialer Faktor von N ist. Die Eingabe des Algorithmus ist das RSA-Modul N.

Zunächst wird eine zufällige Zahl $a \in \mathbb{Z}_N$ gewählt, und eine positive Zahl k gesucht, die durch möglichst viele Primzahlpotenzen teilbar ist. Im einfachsten Fall ist das das kleinste gemeinsame Vielfache $kgV(1, 2, \dots, B)$, wobei B eine geeignete Grenze sein sollte. Nun wird f berechnet.

$$f = ggT(a^k \; mod \; N - 1, \; N)$$

Wenn f nach der Berechnung größer als 1 ist und kleiner als N, dann ist f ein nicht trivialer Faktor von N.

Wenn kein Faktor gefunden wurde, kann der Algorithmus mit einem neuen a und/oder einem neuen k erneut ausgeführt werden.

Pollard's „p-1" Factoring Algorithm hat eine Laufzeit von $\mathcal{O}(B \log B \; (\log N)^2)$ und ist der Kategorie der Algorithmen zuzuordnen, die von der Größe von p abhängig sind.

3.4. Pseudocode

do {

$\mathrm{a} \leftarrow \mathrm{random}$

$\mathrm{B} \leftarrow \mathrm{random}$

$\mathrm{k} \leftarrow \mathrm{lcm}(1, 2, \dots, \mathrm{B})$

$\mathrm{f} = \mathrm{gcd}(\mathrm{a}^{\mathrm{k}} \bmod \mathrm{N} - 1, \; \mathrm{N})$

} **while** $\mathrm{f} \leq 1$ or $\mathrm{f} \geq \mathrm{N}$

3.5. Beispielprogramm

Es folgt ein Beispielprogramm in der Programmiersprache C, das Pollard's „p-1“ Factoring Algorithm implementiert.

```c
/** Pollard's p - 1 Factoring Algorithm */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/**
 * Sucht für die gegebenen Zahlen i1 und i2
 * den groessten gemeinsamen Teiler
 *
 * @param i1 erste Zahl
 * @param i2 zweite Zahl
 * @return groesster gemeinsamer Teiler von i1 und i2
 */
int gcd( int i1, int i2 ) {
      if ( i1 == 1 || i2 == 1 ) return 1;
    if ( i1 == i2 ) return i1;
    if ( i1 > i2 ) return gcd( i1 - i2, i2 );
    if ( i1 < i2 ) return gcd( i1, i2 - i1 );

    return 0;
}

/**
 * Sucht für die gegebenen Zahlen i1 und i2
 * das kleinste gemeinsame Vielfache
 *
 * @param i1 erste Zahl
 * @param i2 zweite Zahl
 * @return kleinstes gemeinsames Vielfaches von i1 und i2
 */
int lcm( int i1, int i2 ) {
    if ( i1 == 0 || i2 == 0 ) return 0;

    return ( i1 * i2 ) / gcd( i1, i2 );
}
```

```
/**
 * Initialisiert alle Variablen für den "p-1" Algorithmus
 *
 * @param N RSA-Modul
 * @param *a zufällige Zahl
 * @param *k lcm(1, ..., B)
 * @param *B Grenze für *k
 */
void initialization( int N, int * a, int * k, int * B ) {
    int i = 2;

    (*a) = rand() % N;
    (*B) = rand() % 5; /* Limitierung, da in diesem Programm
       mit Integer-Werten gearbeitet wird */

    (*k) = 1;
    while ( i <= (*B) ) {
        (*k) = lcm( (*k), i );
        i++;
    }
}

/**
 * Berechnet den groessten gemeinsamen Teiler von a^k-1 und N
 *
 * @param N RSA-Modul
 * @param a zufällige Zahl
 * @param k lcm(1, ..., B)
 */
int computeGCD( int N, int a, int k ) {
    return gcd( ( ( int ) pow( ( double ) a, ( double ) k ) - 1 ) % N, N );
}

/**
 * Prueft, ob ein nicht trivialer Faktor gefunden wurde
 *
 * @param N RSA-Modul
 * @param f potentieller Faktor
 * @return Wahrheitswert, ob ein Faktor gefunden wurde
 */
int factorFound( int N, int f ) {
    if ( 1 < f && f < N ) {
        printf("Factor: %i\n", f);
        return 1;
    }

    return 0;
}
```

```
/**
 * Hauptprogramm
 * Leitet Pollards "p-1" Factoring Algorithm ein
 */
int main( int argc, char **argv ) {
    int N, a, k, B;

    if ( argc < 2 ) exit( 1 );

    N = atoi( argv[1] );
    do {
        initialization( N, &a, &k, &B );
    } while ( !factorFound( N, computeGCD( N, a, k ) ) );

      return 0;
}
```

3.6. Beispiel

Im Weiteren wird Pollard's „p-1" Factoring Algorithm an einem Beispiel gezeigt.

Gegeben ist ein $N = 540143$.

Gewählt wird $a = 2$ und $B = 8$. Daraus folgt, dass $k = lcm(1,2,3,4,5,6,7,8) = 840$ ist.

Die Berechnung für f lautet:

$$f = \gcd\left(a^k \bmod N - 1,\ N\right) = 421$$

Damit wäre ein nicht trivialer Faktor gefunden.

$$p = 421$$

$$q = \frac{540143}{421} = 1283$$

4. Quellen

4.1. Literatur

[1] Song Y. Yan: Cryptanalytic Attacks on RSA, 2008 Springer Science + Business Media LLC

[2] Rainer Schulze-Pillot: Elementare Algebra und Zahlentheorie, Springer-Verlag 2007

[3] Albrecht Beutelspacher, Jörg Schwenk, Klaus-Dieter Wolfenstetter: Moderne Verfahren der Kryptographie, Friedr. Vieweg & Sohn Verlag / GWV Fachverlage GmbH, Wiesbaden 2006

4.2. Internet

[4] http://mathworld.wolfram.com/FermatsFactorizationMethod.html (Stand 05.06.2016)

[5] http://www.mathepedia.de/Eulersche_Phi-Funktion.aspx (Stand 05.06.2016)

[6] http://www.mathepedia.de/Satz_von_Fermat.aspx (Stand 05.06.2016)

[7] https://www.wi1.uni-muenster.de/pi/lehre/ws0708/seminar/Abgaben/Faktorisierung.pdf (Stand 05.06.2016)