Universität Paderborn
Fakultät für Elektrotechnik, Informatik und Mathematik

Implementierung der Coppersmith-Methode zur Untersuchung von Schwachstellen in RSA

Diplomarbeit vorgelegt von: Christoph Spanke
Matrikelnummer: 6055379

Betreut von: Alexander May

Vorgelegt bei:
Prof. Dr. Johannes Blömer, Universität Paderborn

Zweitgutachter:
Prof. Dr. Friedhelm Meyer auf der Heide, Universität Paderborn

E-Mail: spanke@uni-paderborn.de

Paderborn, August 2005

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß aus einer anderen Arbeit übernommen wurden, sind als solche gekennzeichnet.

Paderborn, den 31. August 2005

Christoph Spanke

Vorwort

Sichere Übertragungen werden in Zeiten zunehmenden Datenverkehrs ein immer wichtigeres Thema. Beispielsweise sollen Nachrichten, mit denen Informationen wie Geschäfts- oder Finanzdaten über das Internet ausgetauscht werden, so gut wie möglich vor nicht authorisiertem Zugriff geschützt werden. Zu diesem Zweck herangezogene Verfahren müssen hohe Sicherheitsanforderungen erfüllen. Eines der verbreitetsten Verfahren zur Verschlüsselung von E-Mails ist RSA, benannt nach seinen Entwicklern Rivest, Shamir und Adleman, entwickelt im Jahr 1977. Es handelt sich um ein sogenanntes „Public Key“-Verschlüsselungsverfahren und gibt als solches den Benutzern den enormen Vorteil gegenüber anderen Verfahren, verschlüsste Nachrichten austauschen zu können, ohne vorher sicherheitskritische Informationen wie Codeschlüssel austauschen zu müssen. Diese müssten ebenfalls sicher übermittelt werden, z.B. durch ein Treffen der beteiligten Personen. Ein solcher Austausch ist nicht immer leicht zu gewährleisten.
RSA wird auch zum Austausch von Codeschlüsseln für andere, beispielsweise sogenannte symmetrische Verschlüsselungsverfahren benutzt, um den eigentlichen Nachrichtenverkehr mit diesem anderen Verfahren zu chiffrieren. Ein Verfahren wird symmetrisch genannt, wenn sowohl zum Chiffrieren als auch zum Dechiffrieren der gleiche Schlüssel zugänglich sein muss. Man kann diese Verfahren also als Gegenstück zum „Public Key“-Konzept ansehen.
Die Aufgabenstellung dieser Diplomarbeit bestand darin, verschiedene Algorithmen in *C/C++* umzusetzen, deren theoretische Grundlagen schon seit einigen Jahren bekannt sind. Sie alle basieren auf einem von Coppersmith[2] vorgestellten Konzept und können zu Angriffen auf das RSA-Verfahren benutzt werden. Unter einem Angriff auf ein Verschlüsselungsverfahren verstehen wir in dieser Arbeit einen algorithmischen Versuch, die Sicherheit des entsprechenden Verfahrens zu brechen, so dass der Angreifer Nachrichten entschlüsseln und lesen kann, die vorher mit dem Verfahren verschlüsselt worden sind, ohne dass er im Vorfeld über die zur Entschlüsselung benötigten Informationen verfügt.
Im Folgenden werden wir ebenfalls beschreiben, wie die vorgestellten Algorithmen mit RSA verwendet werden können, um den privaten Schlüssel aus dem öffentlichen Schlüssel zu berechnen. Die Sicherheit von RSA basiert auf der Tatsache, dass kein Algorithmus bekannt ist, mit dem eine Zahl $N \in \mathbb{Z}$ effizient in ihre Primfaktoren zerlegt werden kann. Ist allerdings eine hinreichend gute Approximation

eines der Primfaktoren verfügbar, ist es möglich, N effizient zu faktorisieren und die Sicherheit des RSA-Verfahrens zu brechen. Einige Algorithmen, die das leisten können, werden in dieser Arbeit vorgestellt und mit verschiedenen Testergebnissen, Beispielen und Anwendungen beschrieben.

Die Arbeit beginnt mit einer Einführung in das RSA-Verfahren. Im ersten Abschnitt des ersten Kapitels wird das Verfahren selbst ausführlich erklärt und eine Angriffsvariante motiviert. Der zweite Abschnitt beschäftigt sich mit dem nach seinen Erfindern Lenstra, Lenstra und Lovasz benannten L^3-Algorithmus, der in allen später vorgestellten Algorithmen eine entscheidende Rolle spielen wird. Wir werden in dieser Arbeit nicht näher auf den Algorithmus selbst eingehen, sondern nur die für diese Arbeit wichtigen Resultate aufführen. Die Implementierungen für diese Arbeit greifen auf verschiedene Methoden aus der *C/C++*-Bibliothek *NTL* zurück. Die Bibliothek selbst und die benutzten Methoden werden dann im dritten Abschnitt des ersten Kapitels genauer beschrieben.

Jedem der thematisierten Algorithmen ist je ein Kapitel gewidmet. Im ersten Abschnitt des jeweiligen Kapitels sind die theoretischen Grundlagen zusammengefasst. Die Theorie wurde zu einem großen Teil jeweils der Arbeit entnommen, in welcher das entsprechende Konzept zum ersten Mal vorgestellt wurde. Allerdings haben wir im Rahmen dieser Arbeit viel Wert darauf gelegt, die Theorie kleinschrittig aufzubereiten, so dass sie für den mit der Materie nicht vertrauten Leser leichter nachzuvollziehen ist. Dann folgt jeweils als zweiter Abschnitt eine Erläuterung über die eigentliche Implementierung, die im Rahmen dieser Arbeit entstanden ist. Zu jedem Algorithmus werden im abschließenden, dritten Abschnitt Anwendungen vorgestellt, die sich alle auf die Thematik RSA beziehen. Die Algorithmen lassen sich auf verschiedene Arten zu Angriffen auf RSA einsetzen, und einige der Möglichkeiten werden in diesen Abschnitten erläutert.

Die Arbeit endet mit einer Zusammenfassung und Interpretation der wichtigsten Ergebnisse. Das Ziel ist es, einen Überblick über die Möglichkeiten zu geben, die die vorgestellten Verfahren bieten und Methoden mit ähnlicher Zielsetzung zu vergleichen. Außerdem wollen wir eine Aussage darüber treffen, welche Angriffsvariante für einen bestimmten Zweck am nützlichsten erscheint. Weiter soll klar werden, in welchen Größenordnungen sich die Laufzeiten der Programme bewegen, wenn man ein bestimmtes Problem lösen will, so dass man bei Verwendung des Codes abschätzen kann, wie lange auf ein Resultat gewartet werden muss. Zu diesem Zweck werden wir sowohl theoretisch errechnete Schranken als auch Laufzeittabellen angeben.

Inhaltsverzeichnis

Kapitel 1

Grundlagen zu den vorgestellten Algorithmen

1.1 Das RSA-Verfahren

Das RSA-Verfahren ist ein „Public Key“-Verschlüsselungsverfahren, mit dem es einem beliebigen Absender ermöglicht werden soll, Nachrichten an einen Adressaten zu verschlüsseln, so dass nur dieser die Nachricht auch wieder entschlüsseln kann, ohne dass Absender und Adressat vorher sicherheitskritische Informationen austauschen müssen. Die zur Verschlüsselung einer Nachricht nötigen Informationen bilden den öffentlichen Teil des Schlüssels, den „Public Key“. Dieser muss vom Adressaten erzeugt und öffentlich zugänglich gemacht werden. Mit diesen Informationen ist es jeder Person möglich, Nachrichten so zu verschlüsseln, dass nur der Adressat sie entschlüsseln kann, solange das Verfahren sicher ist.
Die zur Entschlüsselung nötigen Informationen hält der Adressat geheim, denn nur er soll in der Lage sein, die an ihn gerichteten Nachrichten zu lesen. Analog wird dieser Teil des Schlüssels als „Private Key“ bezeichnet. Das Verfahren funktioniert im Detail folgendermaßen:

- Der Adressat berechnet zwei Primzahlen $p, q \in \mathbb{N}$ von ungefähr gleicher Größe und berechnet dann $N := p \cdot q$
- Der Adressat berechnet $\phi(N)$, die Euler'sche *Phi*-Funktion an der Stelle N, dazu nutzt er die Berechnungsvorschrift $\phi(N) = (p-1) \cdot (q-1)$
- Der Adressat wählt eine zufällige Zahl e, $1 < e < \phi(N)$, so dass die Bedingung $ggT(e, \phi(N)) = 1$ erfüllt ist
- Der Adressat berechnet mit dem Erweiterten Euklid'schen Algorithmus die Zahl d, $1 < d < \phi(N)$, so dass $e \cdot d = 1 \ mod \ \phi(N)$ erfüllt ist; eine solche Zahl d existiert, weil $ggT(e, \phi(N)) = 1$ gilt

- Die Verschlüsselungsfunktion $enc(x)$ zu einer Nachricht $x \in \mathbb{Z}_N$, wird definiert als $enc(x) := x^e \; mod \; N$
- Auch die Entschlüsselungsfunktion kann nun definiert werden; zu einem Chiffretext $y \in \mathbb{Z}_N$ definiert man $dec(y) := y^d \; mod \; N$

Ein Theorem von Leonhard Euler zu der nach ihm benannten ϕ-Funktion besagt, dass Exponenten zu Basen aus $\mathbb{Z}_N^*$ *modulo* $\phi(N)$ reduziert werden können. Also gilt, da $ed = 1 \; mod \; \phi(N)$, für eine korrekt verschlüsselte Nachricht $x \in \mathbb{Z}_N^*$:

$$dec(enc(x)) = (x^e)^d = x^{ed} = x \; mod \; N$$

Auch für Nachrichten $x \in \mathbb{Z}_N \setminus \mathbb{Z}_N^*$ kann man zeigen, dass diese Gleichung gilt. Bei Betrachtung der Verschlüsselungsfunktion $enc(x)$ wird deutlich, dass der Absender die Zahlen e und N benötigt, um zu einer Nachricht x die verschlüsselte Nachricht $enc(x)$ zu berechnen. Folglich müssen e und N den öffentlichen Teil des Schlüssels bilden. Um zu gewährleisten, dass nur der Adressat in der Lage ist, die Nachrichten zu entschlüsseln, muss der Exponent d geheim gehalten werden. Und da es ohne großen Aufwand möglich ist, d zu berechnen, wie es auch der Erzeuger des Schlüssels getan hat, wenn $\phi(N)$ oder p und q bekannt sind, müssen folglich auch $\phi(N)$, p und q geheim gehalten werden. $\phi(N)$, p, q und d bilden also den privaten Schlüssel.
Eine Möglichkeit, die Sicherheit des Verfahrens zu brechen, ist es folglich, einen der beiden Primfaktoren von N zu berechnen; den anderen erhält man dann direkt, indem man N durch den gefundenen Primfaktor dividiert. Mit den beiden Primfaktoren p, q lässt sich, wie es auch der Adressat getan hat, der Exponent d berechnen und Nachrichten können, wie in der Beschreibung des Verfahrens, dechiffriert werden. Die Sicherheit von RSA wäre gebrochen. Das wichtigste Thema in den folgenden Kapiteln ist, wie die Primfaktoren von N bestimmt werden können, falls einige bestimmte Bits von p und q bekannt sind. Dann nämlich kann der Rest gegebenenfalls effizient berechnet werden und das RSA-Verfahren ist nicht mehr sicher. Mehr Informationen über RSA finden sich in Stinson[1] und Menezes, van Oorschot, Vanstone[10].

1.2 Das Gram-Schmidt-Verfahren

Sei $B = \{b_1, \ldots, b_n\}$, $b_i \in \mathbb{R}^n$, eine Basis des $\mathbb{R}^n$, also:

$$\{z \in \mathbb{R}^n | z = \sum_{i=1}^{n} c_i b_i, c_i \in \mathbb{R}\} = \mathbb{R}^n$$

Das Gram-Schmidt-Orthogonalisierungsverfahren wird benutzt, um zu einer solchen Basis B eine sogenannte Orthogonalbasis B^* zu berechnen, deren Eigenschaften wir in den folgenden Kapiteln nutzen werden. Der Orthogonalisierungsalgorithmus beginnnt dabei mit dem ersten Vektor der Basis B, berechnet den zu diesem Vektor orthogonalen Anteil des zweiten Vektors u.s.w.; dazu benutzt man die Berechnungsvorschriften:

$$\begin{aligned}
b_1^* &= b_1 \\
b_2^* &= b_2 - \frac{< b_2, b_1^* >}{\|b_1^*\|^2} \cdot b_1^* \\
b_3^* &= b_3 - \frac{< b_3, b_1^* >}{\|b_1^*\|^2} \cdot b_1^* - \frac{< b_3, b_2^* >}{\|b_2^*\|^2} \cdot b_2^* \\
&\vdots \\
b_n^* &= b_n - \frac{< b_n, b_1^* >}{\|b_1^*\|^2} \cdot b_1^* - \frac{< b_n, b_2^* >}{\|b_2^*\|^2} \cdot b_2^* - \ldots - \frac{< b_n, b_{n-1}^* >}{\|b_{n-1}^*\|^2} \cdot b_{n-1}^*
\end{aligned} \tag{1}$$

Dann ist $B^* = \{b_1^*, \ldots, b_n^*\}$ eine Orthogonalbasis zu B. Sei A eine Matrix mit den Vektoren b_i als Zeilen und A^* eine Matrix mit den Vektoren b_i^* als Zeilen. Wir werden benutzen, dass für Vektoren $c', c'' \in \mathbb{Z}^n$ mit $c' \cdot A = c'' \cdot A^*$ gilt:

$$c'_n = c''_n$$

Dieses Resultat werden wir an dieser Stelle zeigen. Sei also $c', c'' \in \mathbb{Z}^n$ mit $c' \cdot A = c'' \cdot A^*$. Mit $c' \cdot A = c'' \cdot A^*$ erhalten wir:

$$c'_1 \cdot b_1 + \ldots + c'_n \cdot b_n = c'' \cdot b_1^* + \ldots + c''_n \cdot b_n^*$$

Wir können jetzt schrittweise mit Hilfe der Berechnungsvorschriften für die Vektoren b_i^*, siehe dazu (1), alle Vorkommen der b_i^* durch b_i ersetzen. Da b_n nur in der Berechnungsvorschrift zu b_n^* auftaucht, erhalten wir ein Gleichungssystem, in dem c''_n einziger Koeffizient der $(b_n)_i$ ist. Das Gleichungssystem hat also die Form (mit $k_i \in \mathbb{Z}$):

$$c'_1 \cdot b_1 + \ldots + c'_n \cdot b_n = k_1 \cdot b_1 + \ldots + k_{n-1} \cdot b_{n-1} + c''_n \cdot b_n$$

Da die Vektoren b_i eine Basis bilden, sind sie linear unabhängig. Für linear unabhängige Vektoren b_i kann diese Gleichung aber nur erfüllt sein, wenn die Koeffizienten zu den Vektoren b_i auf beiden Seiten der Gleichung übereinstimmen. Also muss $c'_n = c''_n$ gelten.

1.3 Der L^3-Algorithmus

Sei $B = \{\tilde{b}_1, \dots, \tilde{b}_n\}$ wieder eine Basis von $\mathbb{R}^n$. Ein Gitter L ist definiert als die Menge der Punkte:

$$L = \{y \in \mathbb{R}^n | y = \sum_{i=1}^{n} a_i \tilde{b}_i, a_i \in \mathbb{Z}\}$$

Ein Gitter zu einer Basis aus dem $\mathbb{R}^n$ ist also eine Teilmenge des $\mathbb{R}^n$ und enthält genau die Punkte, die mit ganzzahligen Linearkombinationen der Basisvektoren in B erzeugt werden können. Verschiedene Basen können das gleiche Gitter aufspannen, eine Basis B zu einem bestimmten Gitter ist also nicht eindeutig. Wir bezeichnen eine Basis, die ein bestimmtes Gitter erzeugt, im Folgenden als Gitterbasis. Eine Matrix, deren Zeilen eine solche Gitterbasis bilden, werden wir im Folgenden als Gittermatrix bezeichnen. Der L^3-Algorithmus, benannt nach seinen Erfindern Lenstra, Lenstra und Lovasz, berechnet bei Eingabe einer solchen Gitterbasis eine sogenannte reduzierte Gitterbasis. Wir wollen nicht genauer darauf eingehen wie diese Berechnung funktioniert, sondern lediglich die zugehörigen Resultate erläutern, die wir benutzen werden. Sei $B \in \mathbb{R}^{n \times n}$ eine Matrix, deren Zeilen die Vektoren $b_1, \dots, b_n$ einer solchen reduzierten Gitterbasis enthält. Diese reduzierte Gitterbasis hat folgende Eigenschaft:

$$\|b_1\| \leq 2^{\frac{n-1}{4}} \cdot |\det(B)|^{\frac{1}{n}} \tag{2}$$

Wenden wir im nächsten Schritt das Gram-Schmidt-Orthogonalisierungsverfahren auf die reduzierte Basis an, so erhalten wir eine orthogonalisierte, reduzierte Gitterbasis $B^* = \{b_1^*, \dots, b_n^*\}$. Für diese lässt sich zeigen:

$$\|b_n^*\| \geq 2^{-\frac{n-1}{4}} \cdot |\det(B^*)|^{\frac{1}{n}} \tag{3}$$

Diese Resultate sind in den folgenden Kapiteln von essentieller Bedeutung. Die asymptotische Laufzeit des L^3-Algorithmus ist gegeben durch $O(n^6 \log(R)^3)$, wobei $R = \max_{1 \leq i \leq n}\{\|b_i\|^2\}$ das Quadrat der Norm des „längsten“ Basisvektors in B ist. Die Verwendung des L^3-Unterprogrammes zur Berechnung der reduzierten Gitterbasis ist in jedem der im Folgenden vorgestellten Programme der laufzeitbestimmende Teil. Das für diese Arbeit verwendete Programm zur Berechnung einer reduzierten Gitterbasis stammt aus der *C/C++*-Bibliothek *NTL*, mehr Informationen dazu in Abschnitt 1.4.

1.4 Die NTL-Bibliothek

Um die verschiedenen Algorithmen implementieren zu können, wurden verschiedene Funktionen der *C/C++*-Bibliothek *NTL*, Version 5.4, eingesetzt. Die Bibliothek bietet unter anderem Datentypen zum Speichern und Rechnen mit univariaten Polynomen sowie betragsmäßig beliebig großen, ganzen Zahlen. Außerdem stehen dem Nutzer mehrere Varianten des L^3-Algorithmus zur Verfügung. Die Module, welche für die im Rahmen dieser Diplomarbeit entstandenen Programme benötigt werden, sind:

- LLL: Enthält die verschiedenen Programme zur Berechnung von reduzierten Gitterbasen.
- ZZ: Datentyp zum Speichern und Rechnen mit ganzen Zahlen. Enthält außerdem benötigte Routinen zur Erzeugung von Zufallszahlen aus $\mathbb{Z}$.
- mat_ZZ: Datentyp zum Speichern und Rechnen mit Matrizen über $\mathbb{Z}$.
- ZZX: Datentyp zum Speichern und Rechnen mit Polynomen über $\mathbb{Z}[x]$. Enthält benötigte Funktionen, die den größten gemeinsamen Teiler zweier Polynome berechnen.

Zur Nutzung der für diese Arbeit programmierten Funktionen ist die Installation dieser Module von NTL erforderlich. Die Bibliothek ist frei erhältlich auf der Website des verantwortlichen Programmierers Victor Shoup, für Download und Installationsanleitung siehe *http://www.shoup.net/*.

1.5 Die Resultante von zwei Polynomen

Wir werden einen Algorithmus zur Berechnung der Resultante zweier bivariater Polynome benötigen. Die Resultante kann als die Determinante der sogenannten Sylvestermatrix definiert werden. Seien dazu $p(x, y), q(x, y) \in \mathbb{Z}[x, y]$ die Polynome, deren Resultante wir berechnen wollen. Dabei sei:

$$p(x, y) = \sum_{i=0}^{\delta_x} \sum_{j=0}^{\delta_y} p_{ij} \cdot x^i y^j \text{ und } q(x, y) = \sum_{i=0}^{\gamma_x} \sum_{j=0}^{\gamma_y} q_{ij} \cdot x^i y^j$$

Wir schreiben stattdessen:

$$p(x, y) = \sum_{j=0}^{\delta_y} (\sum_{i=0}^{\delta_x} p_{ij} x^i) \cdot y^j \text{ und } q(x, y) = \sum_{j=0}^{\gamma_y} (\sum_{i=0}^{\gamma_x} q_{ij} x^i) \cdot y^j$$

Zur Vereinfachung definieren wir univariate Polynome $a_i(x)$ und $b_i(x)$ so, dass:

$$p(x, y) = \sum_{j=0}^{\delta_y} a_j(x) \cdot y^j \text{ und } q(x, y) = \sum_{j=0}^{\gamma_y} b_j(x) \cdot y^j$$

Mit diesen können wir die zugehörige Sylvestermatrix bilden (aus Gründen der Übersichtlichkeit schreiben wir a_i und b_i anstelle von $a_i(x)$ und $b_i(x)$):

$$
S = \left(\begin{array}{cccccccccc}
a_{\delta_y} & a_{\delta_y-1} & a_{\delta_y-2} & \dots & a_0 & 0 & 0 & 0 & \dots & 0 \\
0 & a_{\delta_y} & a_{\delta_y-1} & \dots & a_1 & a_0 & 0 & 0 & \dots & 0 \\
0 & 0 & a_{\delta_y} & \dots & a_2 & a_1 & a_0 & 0 & \dots & 0 \\
 & & & \ddots & \ddots & & \ddots & \ddots & & \\
0 & 0 & \dots & 0 & a_{\delta_y} & a_{\delta_y-1} & \dots & a_1 & a_0 & 0 \\
0 & 0 & 0 & \dots & 0 & a_{\delta_y} & a_{\delta_y-1} & \dots & a_1 & a_0 \\
\hline
b_{\gamma_y} & b_{\gamma_y-1} & b_{\gamma_y-2} & \dots & b_0 & 0 & 0 & \dots & 0 & \\
0 & b_{\gamma_y} & b_{\gamma_y-1} & \dots & b_1 & b_0 & 0 & 0 & \dots & 0 \\
0 & 0 & b_{\gamma_y} & \dots & b_2 & b_1 & b_0 & 0 & \dots & 0 \\
 & & & \ddots & \ddots & & \ddots & \ddots & & \\
0 & 0 & \dots & 0 & b_{\gamma_y} & b_{\gamma_y-1} & \dots & b_1 & b_0 & 0 \\
0 & 0 & 0 & \dots & 0 & b_{\gamma_y} & b_{\gamma_y-1} & \dots & b_1 & b_0
\end{array}\right)
$$

Wir können nun die Determinante dieser Matrix mit der Zeilenentwicklungsformel berechnen (siehe dazu Lipschutz[5]). Die für diese Arbeit verwendete Funktion zur Resultantenberechnung entspricht dem gerade beschriebenen Algorithmus. Da NTL keine Funktion zur Resultantenberechnung anbietet, ist die Funktion für diese Arbeit implementiert worden.

1.6 Sturm-Sequenz zur Nullstellenberechnung

Mit der Sturm-Sequenz lassen sich, unter anderem, Nullstellen aus $\mathbb{R}$ von univariaten Polynomen über $\mathbb{Z}$ in einem Intervall $(a, b]$ berechnen. Wir werden einen Algorithmus, der dies leistet, des öfteren benötigen und deshalb an dieser Stelle die Sturm-Sequenz erläutern. Sei also $p(x) \in \mathbb{Z}[x]$ ein Polynom vom Grad k. Wir bestimmen sämtliche Glieder der Folge von Ableitungen des Polynoms $p(x)$:

$$u_0(x) = p(x), u_1(x) = (u_0(x))', u_2(x) = (u_1(x))', \dots, u_k(x) = c,\ c \in \mathbb{Z} \text{ konstant}$$

Wir definieren jetzt die Funktion $V_{p(x)}(a)$ als die Anzahl der Vorzeichenwechsel in der Folge $u_0(a), \dots, u_{k+1}(a)$, wobei jedes Ergebnis $u_i(a) = 0$ aus der Reihe gestrichen bzw. ignoriert wird. Betrachten wir beispielsweise folgende Sequenz, erzeugt von einem Wert a:

$$0, 3, -5, -6, 0, 2, 4, -2$$

Die zugehörige Vorzeichensequenz ist:

$$0, +, -, -, 0, +, +, -$$

Wir entfernen die Nullen:

$$+, -, -, +, +, -$$

Der Wert von $V_{p(x)}$ an der Stelle a wäre damit $V_{p(x)}(a) = 3$. Sturms Theorem (siehe Knuth[11]) besagt, dass die Anzahl der Nullstellen von $p(x)$ im Intervall $(a, b]$ gegeben ist durch: $V_{p(x)}(a) - V_{p(x)}(b)$. Wir können also beispielsweise binäre Suche auf dem Intervall $(a, b]$ benutzen, um die Nullstelle schrittweise einzugrenzen und schließlich zu finden. Entsprechend diesem Schema ist die Sturm-Sequenz für diese Arbeit implementiert worden, da sie in NTL nicht enthalten ist.

Kapitel 2

Anwendung von Coppersmiths Algorithmus im univariaten Fall

2.1 Coppersmiths Algorithmus zum univariaten Fall

Der Inhalt dieses Kapitels beschreibt einen Algorithmus, der in Coppersmith[2] vorgestellt wird. Es sind verschiedene Verfahren bekannt, mit denen die Nullstellen aus $\mathbb{Z}$ eines Polynoms effizient berechnet werden können. Eines von diesen ist die Sturm-Sequenz (siehe dazu Abschnitt 1.6), mit der man alle Nullstellen aus $\mathbb{R}$ eines Polynoms berechnen kann und die ganzzahligen unter diesen auswählt. Da die Anzahl von verschiedenen Nullstellen höchstens dem Grad des Polynoms entspricht, ist dies eine effiziente Methode, ganzzahlige Nullstellen von Polynomen zu berechnen. Schwieriger ist das Problem, Nullstellen von Polynomen *modulo* N zu berechnen. Wir definieren dazu das Polynom $p(x)$ von der Form:

$$p(x) = x^\delta + p_{\delta-1}x^{\delta-1} + \ldots + p_2x^2 + p_1x + p_0$$

Man beachte, dass $p_\delta = 1$, das Polynom $p(x)$ ist also als monisch vorausgesetzt. Weiter sei N eine Zahl unbekannter Faktorisierung und x_0 eine Nullstelle *modulo* N, also:

$$p(x_0) = 0 \; mod \; N$$

Es ist möglich, eine Nullstelle effizient zu berechnen, wenn diese in Relation zu N betragsmäßig „klein genug“ ist. Wir werden in diesem Abschnitt das folgende Theorem beweisen:

Theorem 2.1.1. *Sei $x_0 \in \mathbb{Z}$, $X \in \mathbb{N}$ und $\varepsilon \in \mathbb{R}$ mit $0 < \varepsilon < 1$, so dass:*

$$p(x_0) = 0 \; mod \; N \; und \; |x_0| < X \; mit \; X \leq \tfrac{1}{2}N^{\frac{1}{\delta}-\varepsilon}$$

Dann können wir in Laufzeit polynomiell in $(log(N), \delta, \frac{1}{\varepsilon})$ ein Polynom $r(x)$ berechnen, für das:

$$r(x_0) = 0$$

über den ganze Zahlen gilt.

Die Bedingung $|x_0| < X$ gewährleistet, dass die Nullstelle, wie gefordert, klein genug ist, wie wir ebenfalls in diesem Abschnitt zeigen werden. Wir benötigen eine ganze Zahl m, für die gelte:

$$m \geq \max\left(\frac{\delta - 1 + \varepsilon\delta}{\varepsilon\delta^2}, \frac{7}{\delta}\right) \tag{1}$$

Der Grund für die Wahl von X und m wird am Ende dieses Abschnitts deutlich. Mit der Definition von m können wir jetzt das folgende Lemma beweisen, welches wir später benötigen:

Lemma 2.1.2.
Aus der Bedingung:

$$m \geq \max\left(\frac{\delta - 1 + \varepsilon\delta}{\varepsilon\delta^2}, \frac{7}{\delta}\right)$$

folgt:

$$\frac{m-1}{m\delta - 1} \geq \frac{1}{\delta} - \varepsilon$$

Beweis:
Wir benutzen $m \geq \dfrac{\delta - 1 + \varepsilon\delta}{\varepsilon\delta^2}$, und erhalten:

$$\frac{m-1}{m\delta - 1} \geq \frac{\frac{\delta - 1 + \varepsilon\delta}{\varepsilon\delta^2} - 1}{\frac{\delta - 1 + \varepsilon\delta}{\varepsilon\delta^2} \cdot \delta - 1} = \frac{\frac{\delta - 1 + \varepsilon\delta}{\varepsilon\delta^2} - 1}{\frac{\delta - 1 + \varepsilon\delta}{\varepsilon\delta} - 1} = \frac{\frac{\delta - 1 + \varepsilon\delta}{\varepsilon\delta^2} - 1}{\frac{\delta - 1 + \varepsilon\delta}{\varepsilon\delta} - \frac{\varepsilon\delta}{\varepsilon\delta}} = \frac{\frac{\delta - 1 + \varepsilon\delta}{\varepsilon\delta^2} - 1}{\frac{\delta - 1}{\varepsilon\delta}}$$

$$= \frac{\delta - 1 + \varepsilon\delta}{\delta \cdot (\delta - 1)} - \frac{\varepsilon\delta}{\delta - 1} = \frac{1}{\delta} + \frac{\varepsilon}{\delta - 1} - \frac{\varepsilon\delta}{\delta - 1} = \frac{1}{\delta} - \varepsilon$$

Damit ist Lemma 2.1.2 bewiesen. □

Außerdem (wir werden dieses Resultat später verwenden) gilt nach Wahl von m auch:

$$m \geq \frac{7}{\delta}$$

$$\Leftrightarrow m\delta \geq 7$$

Mit diesen definierten Parametern können wir jetzt den eigentlichen Algorithmus beschreiben. Wir definieren die sogenannten „Shifts“ des Polynoms $p(x)$. Für jedes Paar i, j, $0 \leq i < \delta$, $1 \leq j < m$, setzen wir:

$$q_{ij}(x) = x^i \cdot p(x)^j$$

Da $p(x_0) = 0 \; mod \; N$, gilt für all diese Polynome auch:

$$q_{ij}(x_0) = 0 \; mod \; N^j$$

Mit diesen Parametern konstruieren wir eine Gittermatrix $M \in \mathbb{R}^{(2m\delta-\delta)\times(2m\delta-\delta)}$ unter Verwendung der Koeffizienten der Polynome $q_{ij}(x)$. Dazu benutzen wir wieder die Variablen i, j und führen die Variable g ein, wobei:

$$0 \leq g < m\delta \qquad 0 \leq i < \delta \qquad 1 \leq j < m$$

Zur Vereinfachung der Definition der Gittermatrix benutzen wir die folgende Funktion, mit der wir im nächsten Schritt die Position der verschiedenen Elemente in der Matrix bestimmen können:

$$\gamma(i, j) = m\delta + i + (j - 1)\delta.$$

Die Matrix M ist in vier Blöcke aufgeteilt, die wir mit Hilfe der Variablen und der Funktion $\gamma(i, j)$ definieren können.

- Der obere linke $(m\delta) \times (m\delta)$-Block ist eine Diagonalmatrix mit Eintrag X^{-g} an Position (g, g)
- Der untere linke $(m\delta - \delta) \times (m\delta)$-Block besteht vollständig aus Nulleinträgen.
- Der obere rechte Block der Größe $(m\delta) \times (m\delta - \delta)$ enthält jeweils die Koeffizienten der Polynome $q_{ij}(x)$ in den Spalten. Der Eintrag $(g, \gamma(i, j))$ ist der Koeffizient von x^g im Polynom $q_{ij}(x)$.
- Der untere rechte $(m\delta-\delta)\times(m\delta-\delta)$-Block ist ebenfalls eine Diagonalmatrix, mit dem Eintrag N^j in jeder Spalte $\gamma(i, j)$.

Folgendes Beispiel soll den Aufbau der Matrix M verdeutlichen. Für das Polynom $p(x) = x^2 + ax + b$ ist $\delta = 2$, und wir setzen $m = 3$. Weiter sei $p(x)^2 =$

$x^4 + 2a \cdot x^3 + (a^2 + 2b) \cdot x^2 + 2ab \cdot x + b^2$. Dann ist M von der Form:

$$M = \left(\begin{array}{cccccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & b & 0 & b^2 & 0 \\ 0 & X^{-1} & 0 & 0 & 0 & 0 & a & b & 2ab & b^2 \\ 0 & 0 & X^{-2} & 0 & 0 & 0 & 1 & a & a^2+2b & 2ab \\ 0 & 0 & 0 & X^{-3} & 0 & 0 & 0 & 1 & 2a & a^2+2b \\ 0 & 0 & 0 & 0 & X^{-4} & 0 & 0 & 0 & 1 & 2a \\ 0 & 0 & 0 & 0 & 0 & X^{-5} & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & N & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & N & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & N^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & N^2 \end{array}\right)$$

Da der führende Koeffizient des Polynoms $p(x)$ nach Voraussetzung 1 ist, enthält der obere rechte Block eine Untermatrix mit 1 auf der Hauptdiagonalen. Folglich ist es möglich, die Matrix M mit Hilfe elementarer Zeilenoperationen in eine Matrix $\tilde{M}$ zu überführen, so dass $\tilde{M}$ im unteren rechten Block die Einheitsmatrix und im oberen rechten Block die Nullmatrix enthält. Man beachte jedoch, dass $\tilde{M}$ nicht eindeutig bestimmt ist; die erste Zeile könnte beispielsweise noch beliebig oft zu der letzten addiert werden und die geforderte Form wäre noch immer erhalten. Da nur elementare Zeilenoperationen erlaubt sind, kann man die Transformation auch durch Multiplikation mit einer Matrix ausdrücken. Wir definieren die Transformationsmatrix H_1 so, dass:

$$\tilde{M} = H_1 \cdot M$$

gilt. Die Matrix H_1 wird im weiteren Verlauf des Algorithmus noch benötigt. Den oberen linken Block von $\tilde{M}$ bezeichnen wir als $\hat{M}$. Zu $\hat{M}$ wird im nächsten Schritt eine reduzierte Gitterbasis (siehe Abschnitt 1.3) berechnet. Die erhaltene Basis bezeichnen wir mit B. Auch hier kann die Transformation von $\hat{M}$ zu B anhand einer Matrix H_2 ausgedrückt werden. Wir definieren H_2 so, dass:

$$B = H_2 \cdot \hat{M}$$

gilt. Mit Hilfe dieser Definitionen können wir in der folgenden Rechnung zeigen, wie der Algorithmus ein Polynom $r(x)$ mit der Nullstelle x_0 berechnen kann, und unter welcher Bedingung für die Schranke X das möglich ist. Wir setzen dazu:

$$y_0 := \frac{p(x_0)}{N}$$

Betrachten wir jetzt den Vektor c_0, definiert wie folgt:

$$c_0 = (1, x_0, \ldots, x_0^{m\delta-1}, -y_0, -y_0x_0, \ldots, -y_0x_0^{\delta-1}, -y_0^2, -y_0^2x_0, \ldots, -y_0^{\delta-1}x_0^{\delta-1})$$

Weiter definieren wir die Operation $[v]_{sh}$, welche einen Vektor der Länge $2m\delta - \delta$ zu einem Vektor der Länge $m\delta$ verkürzt, indem wir einfach die letzten $(m-1)\delta$ Einträge wegfallen lassen. Es ist:

$$\|c_0M\| = \|(1, \frac{x_0}{X}, \ldots, \left(\frac{x_0}{X}\right)^{m\delta-1}, 0, \ldots, 0)\| \tag{2}$$

Da $\dfrac{|x_0|}{X} \leq 1$:

$$\|(1, \frac{x_0}{X}, \ldots, \left(\frac{x_0}{X}\right)^{m\delta-1}, 0, \ldots, 0)\| \leq \|(\overbrace{1, 1, \ldots, 1}^{m\delta-mal}, 0, \ldots, 0)\| = \sqrt{m\delta}$$

Wir haben also:

$$\sqrt{m\delta} \geq \|c_0M\| = \|c_0H_1^{-1}\tilde{M}\|$$

Nach (2) gilt also $(c_0H_1^{-1}\tilde{M})_i = 0$ für $i > m\delta$. Da der untere rechte Block von $\tilde{M}$ der Einheitsmatrix und der obere rechte Block der Nullmatrix entspricht, werden die letzten $(m-1)\delta$ Elemente von $c_0H_1^{-1}$ auf sich selbst abgebildet. Es muss also bereits $(c_0H_1^{-1})_i = 0$ für $i > m\delta$ gelten. Folglich haben wir:

$$\|c_0H_1^{-1}\tilde{M}\| = \|[c_0H_1^{-1}]_{sh}\hat{M}\| = \|[c_0H_1^{-1}]_{sh}H_2^{-1}B\|$$

Wir setzen $c' := [c_0H_1^{-1}]_{sh}H_2^{-1}$:

$$\|[c_0H_1^{-1}]_{sh}H_2^{-1}B\| = \|c'B\|$$

Sei nun B^* die Gram-Schmidt-Orthogonalisierung der Basis B. Auch die Gram-Schmidt-Orthogonalisierung lässt sich als Matrix ausdrücken. Wir definieren $OB = B^*$ und erhalten so $B = O^{-1}B^*$, also:

$$\|c'B\| = \|c'O^{-1}B^*\|$$

Mit $c'' := c'O^{-1}$:

$$\|c'O^{-1}B^*\| = \|c''B^*\| \geq \|c''_{m\delta}b^*_{m\delta}\|$$

Da $c'_{m\delta} = c''_{m\delta} \in \mathbb{Z}$, wie in Abschnitt 1.2 gezeigt, gilt auch:

$$\|c''_{m\delta} b^*_{m\delta}\| = \|c'_{m\delta} b^*_{m\delta}\| = |c'_{m\delta}| \cdot \|b^*_{m\delta}\|$$

Eine Eigenschaft der reduzierten Gitterbasis ist, dass $\|b^*_{m\delta}\| \geq 2^{-\frac{m\delta-1}{4}} \cdot |\det(\hat{M})|^{\frac{1}{m\delta}}$, siehe Abschnitt 1.3. Mit dieser Ungleichung erhalten wir weiter:

$$|c'_{m\delta}| \cdot \|b^*_{m\delta}\| \geq |c'_{m\delta}| \cdot 2^{-\frac{m\delta-1}{4}} \cdot |\det(\hat{M})|^{\frac{1}{m\delta}}$$

Also ergibt sich insgesamt:

$$\sqrt{m\delta} \geq |c'_{m\delta}| \cdot 2^{-\frac{m\delta-1}{4}} \cdot |\det(\hat{M})|^{\frac{1}{m\delta}} \tag{3}$$

Weiter gilt $\det(\hat{M}) = \det(\tilde{M}) = \det(M)$ auf Grund der Form von M sowie $\tilde{M}$, und $\det(M)$ lässt sich relativ einfach berechnen, da M eine Dreiecksmatrix ist. Wir erhalten damit:

$$\det(\hat{M}) = |\det(\hat{M})| = (N^{\frac{m-1}{2}} \cdot X^{-\frac{m\delta-1}{2}} \cdot (m\delta)^{-\frac{1}{2}})^{m\delta}$$

Wir setzen dieses Resultat in (3) ein und erhalten:

$$\begin{aligned}
&\sqrt{m\delta} \geq |c'_{m\delta}| \cdot 2^{-\frac{m\delta-1}{4}} \cdot N^{\frac{m-1}{2}} \cdot X^{-\frac{m\delta-1}{2}} \cdot (m\delta)^{-\frac{1}{2}} \\
&\Leftrightarrow \sqrt{m\delta}^{-1} \leq |c'_{m\delta}|^{-1} \cdot 2^{\frac{m\delta-1}{4}} \cdot N^{-\frac{m-1}{2}} \cdot X^{\frac{m\delta-1}{2}} \cdot (m\delta)^{\frac{1}{2}} \\
&\Leftrightarrow |c'_{m\delta}| \leq \sqrt{m\delta} \cdot 2^{\frac{m\delta-1}{4}} \cdot N^{-\frac{m-1}{2}} \cdot X^{\frac{m\delta-1}{2}} \cdot (m\delta)^{\frac{1}{2}}
\end{aligned}$$

Da $c'_{m\delta} \in \mathbb{Z}$, muss $c'_{m\delta} = 0$ sein für $|c'_{m\delta}| < 1$. Wir wollen garantieren, dass $c'_{m\delta} = 0$ gilt, und fordern also:

$$\sqrt{m\delta} \cdot 2^{\frac{m\delta-1}{4}} \cdot N^{-\frac{m-1}{2}} \cdot X^{\frac{m\delta-1}{2}} \cdot (m\delta)^{\frac{1}{2}} < 1$$

Nach X umgestellt erhalten wir:

$$X < N^{\frac{m-1}{m\delta-1}} \cdot (m\delta)^{\frac{-1}{m\delta-1}} \cdot 2^{-\frac{1}{2}} \tag{4}$$

Mit Lemma 2.1.2 wissen wir, dass:

$$\frac{m-1}{m\delta-1} \geq \frac{1}{\delta} - \varepsilon \tag{5}$$

Nach Voraussetzung gilt $m\delta \geq 7$, und es folgt:

$$(m\delta)^{\frac{-1}{m\delta-1}} > 2^{-\frac{1}{2}} \tag{6}$$

Gleichungen (5) und (6) ergeben:

$$N^{\frac{m-1}{m\delta-1}} \cdot (m\delta)^{\frac{-1}{m\delta-1}} \cdot 2^{-\frac{1}{2}} > N^{\frac{m-1}{m\delta-1}} \cdot 2^{-\frac{1}{2}} \cdot 2^{-\frac{1}{2}} = \tfrac{1}{2} N^{\frac{m-1}{m\delta-1}} > \tfrac{1}{2} N^{\frac{1}{\delta}-\varepsilon}$$

Also gilt mit:

$$X \leq \tfrac{1}{2} N^{\frac{1}{\delta}-\varepsilon}$$

auch:

$$X < N^{\frac{m-1}{m\delta-1}} \cdot (m\delta)^{-\frac{1}{m\delta-1}} \cdot 2^{-\frac{1}{2}}$$

An dieser Ungleichung lässt sich ablesen, dass ein größerer Parameter m gleichzeitig erlaubt, die Grenze X zu erhöhen. Wir definieren jetzt den Vektor $c(x)$ der Länge $2m\delta - \delta$ mit:

$$\begin{aligned} c(x) = (1, x, \ldots, x^{mk-1}, -\tfrac{p(x)}{N}, -\tfrac{xp(x)}{N}, \ldots, -\tfrac{x^{k-1}p(x)}{N}, (\tfrac{p(x)}{N})^2, \\ -x(\tfrac{p(x)}{N})^2, \ldots, -x^{k-1}(\tfrac{p(x)}{N})^2, \ldots, -x^{k-1}(\tfrac{p(x)}{N})^{m-1}) \end{aligned}$$

Im Unterschied zu c_0 können wir den Vektor $c(x)$ direkt bestimmen, weil dazu nicht die Nullstelle x_0 benötigt wird, deren Berechnung ja das eigentliche Ziel dieses Abschnitts ist. Wir definieren das Polynom:

$$r(x) = [c(x) \cdot H_1^{-1}]_{sh} \cdot ((H_2^{-1})^T)_{m\delta}$$

Der Ausdruck $((H_2^{-1})^T)_{m\delta}$ bezeichnet dabei die letzte Spalte von H_2^{-1}. Anhand der durchgeführten Rechnungen für den Vektor c_0 wissen wir, dass $r(x_0) = 0$ gilt, falls $|x_0| < X$, da wie gezeigt $c'_{m\delta} = 0$ für $|c'_{m\delta}| < 1$. Die Nullstellen in $\mathbb{Z}$ eines solchen Polynoms $r(x)$ können jetzt im letzten Schritt mit der schon am Anfang dieses Abschnitts erwähnten Sturm-Sequenz berechnet werden; wir berechnen alle Nullstellen in $\mathbb{R}$ und selektieren die ganzzahligen unter diesen. Eine davon ist folglich die gesuchte Nullstelle x_0.

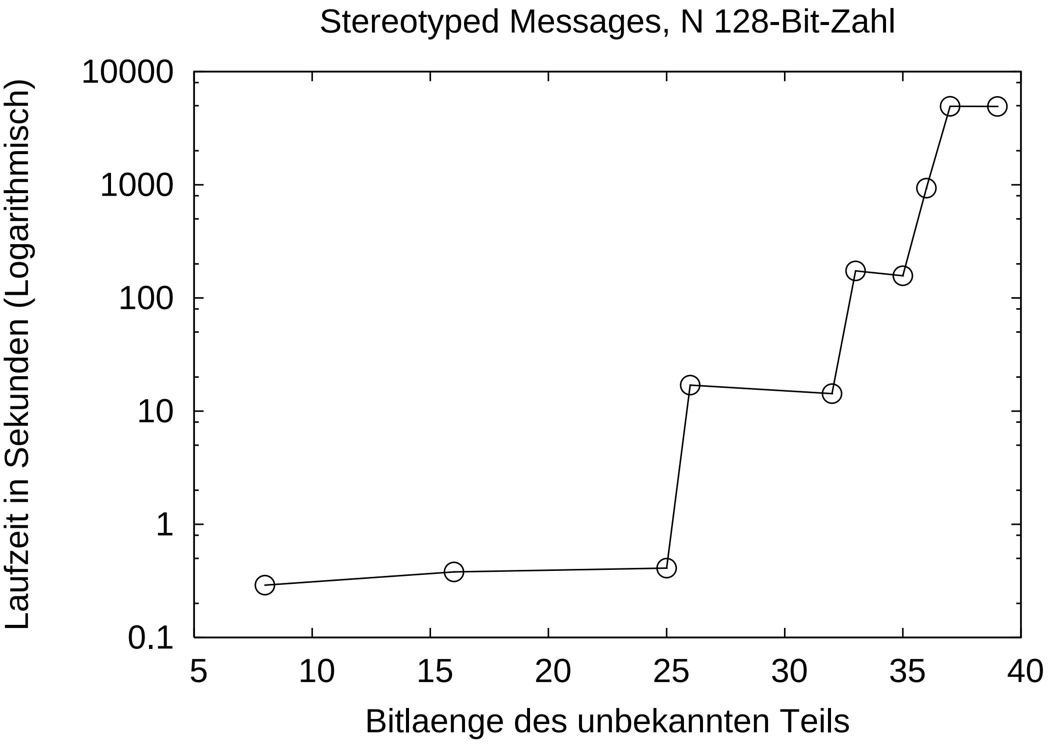

Abbildung 2.1: Laufzeitkurve zur Anwendung „Stereotyped Messages“ (N 128-Bit-Zahl)

2.2 Implementierung zu Coppersmith im univariaten Fall

Im Rahmen dieser Arbeit wurde der in Abschnitt 2.1 vorgestellte Algorithmus auch implementiert. Die Signatur der entsprechenden *C/C++*-Funktion ModularMonovariateRoots, hier abgekürzt MMR, hat die Form:

```
long MMR(ZZ* coeff, ZZ* roots, ZZ N, ZZ X, long delta, long m, int out)
```

Sei $p(x)$ das Polynom, dessen Nullstellen *modulo* N berechnet werden sollen. Sei δ der Grad von $p(x)$ in x. Die Signatur setzt sich zusammen aus:

- Der erste Parameter coeff ist ein Pointer auf ein Array, welches die Länge $\delta + 1$ haben und den Koeffizienten zu x^i von $p(x)$ an Position i enthalten muss.

- Der zweite Parameter roots erwartet einen Pointer auf ein Array der Länge δ, der Inhalt ist beim Funktionsaufruf unwichtig. In dem Array werden

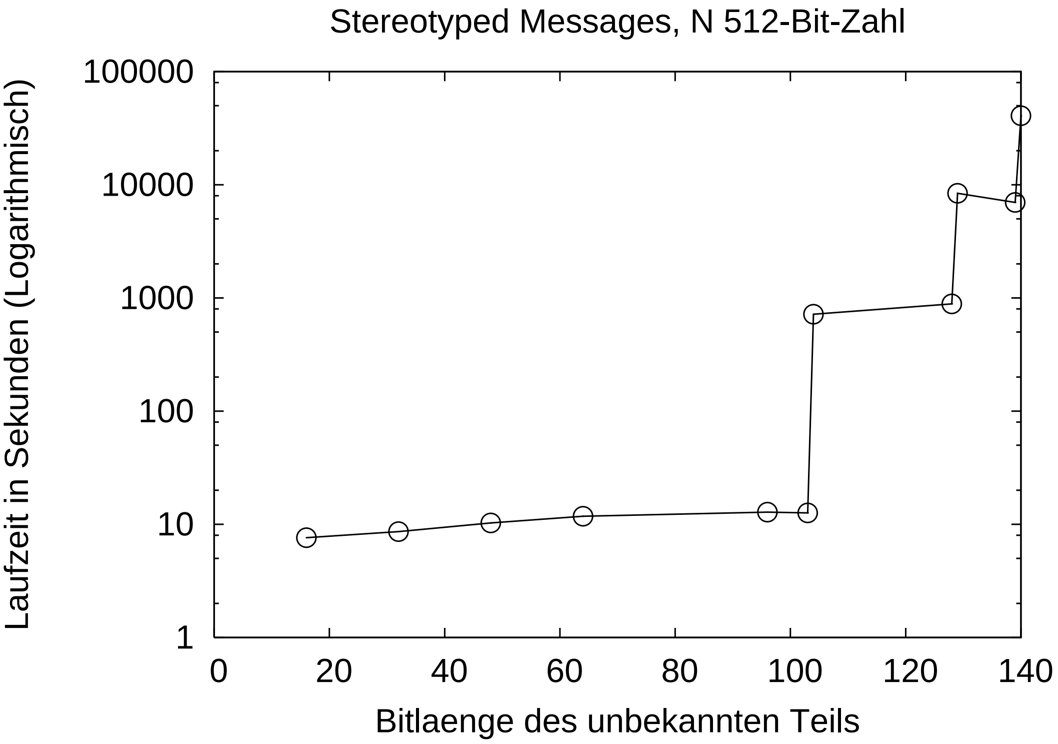

Abbildung 2.2: Laufzeitkurve zur Anwendung „Stereotyped Messages“ (N 512-Bit-Zahl)

später die gefundenen Nullstellen abgespeichert und können nach beendeter Berechung ausgelesen werden.

- Der dritte Parameter N ist für den Modulus gedacht, mit dem gerechnet werden soll.
- Der vierte Parameter X entspricht der oberen Schranke für die gesuchte Nullstelle.
- Der fünfte Parameter `delta` ist zur Übergabe des Grades δ des Polynoms $p(x)$ bestimmt.
- Der sechste Parameter `m` entspricht der Beschreibung in Abschnitt 2.1, mit diesem Parameter lassen sich die Größe der Matrix M und die maximal erlaubte Größe für X kontrollieren.
- Der siebte und letzte Parameter `out` bestimmt, ob der Nutzer Ausgaben der im Verlaufe des Algorithmus benötigten Matrizen, Vektoren und Polynome sehen möchte. Setzung von out $= 1$ führt zur Ausgabe der Zwischenergebnisse, out $\neq 1$ unterdrückt die Ausgabe.

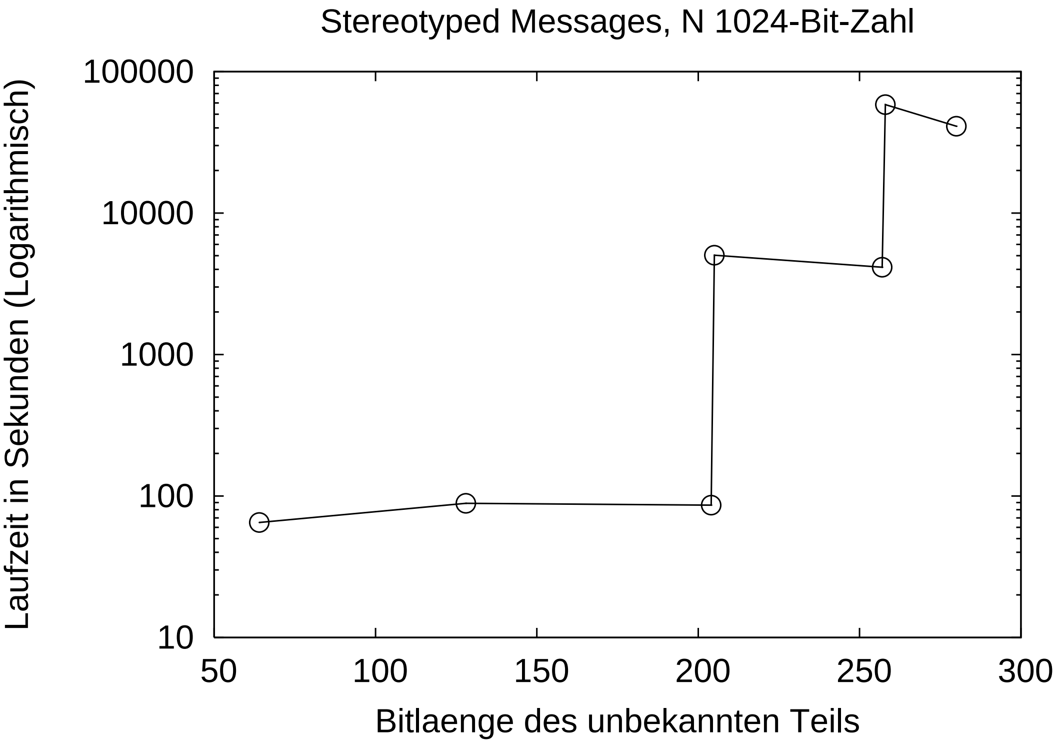

Abbildung 2.3: Laufzeitkurve zur Anwendung „Stereotyped Messages“ (N 1024-Bit-Zahl)

Es ist übrigens durchaus möglich, dass der Algorithmus die Nullstelle x_0 finden kann, obwohl nicht $|x_0| < X$ eingehalten wird. Dass auch in diesem Fall die Nullstelle korrekt berechnet werden kann ist nicht ungewöhnlich, da bei der Berechnung der Schranke pessimistische Abschätzungen gemacht worden sind. Garantiert werden kann die korrekte Berechnung mit dem in Abschnitt 2.1 geführten Beweis jedoch nur, wenn $|x_0| < X$ eingehalten wird. Wie in Abschnitt 2.1 erläutert, wird die Matrix M erzeugt und mit Hilfe elementarer Zeilenoperationen in eine Matrix $\tilde{M}$ überführt, so dass $\tilde{M}$ im unteren rechten Block die Einheitsmatrix und im oberen rechten Block nur noch Nullen enthält. Wir definieren wieder die Transformationsmatrix H_1 so, dass:

$$\tilde{M} = H_1 \cdot M$$

gilt. Im Programm wird H_1 dabei bestimmt, indem man jeden einzelnen Transformationsschritt in eine entsprechende Transformationsmatrix übersetzt und diese jeweils mit der bisher benutzten Transformationsmatrix, im ersten Schritt einfach die Einheitsmatrix, multipliziert. Das ist, unseren Experimenten zufolge, eine wesentlich schnellere Methode, als H_1 erst nach der Transformation durch $H_1 = \tilde{M} \cdot M^{-1}$ zu berechnen. Von der transformierten Matrix $\tilde{M}$ benötigen wir wie gesagt den oberen linken Block $\hat{M}$. In diesem werden die Brüche durch Multiplikation mit der Zahl $X^{m\delta-1}$ eliminiert. Das ist nötig, weil die Berechnung einer reduzierten Gitterbasis in der verwendeten Programmbibliothek NTL als Eingabe ein ganzzahliges Gitter erwartet. In der theoretischen Betrachtung in Abschnitt 2.1 ist diese Skalierung nicht von Belang und wird deshalb nicht berücksichtigt. Diese Skalierung ist der einzige größere Unterschied zwischen dem Konzept des Algorithmus' und seiner eigentlichen Implementierung. Zu $\hat{M}$ wird mit Hilfe einer NTL-Methode eine reduzierte Gitterbasis B berechnet und gleichzeitig H_2 bestimmt, so dass:

$$B = H_2 \cdot \hat{M}$$

gilt. Im nächsten Schritt werden, ebenfalls unter Verwendung von NTL-Routinen, die Inversen von H_1 und H_2 bestimmt. Dann erzeugt das Programm den Vektor:

$$\begin{aligned} c(x) = (1, x, \ldots, x^{mk-1}, -\tfrac{p(x)}{N}, -\tfrac{xp(x)}{N}, \ldots, -x^{k-1}\tfrac{p(x)}{N}, -(\tfrac{p(x)}{N})^2, \\ -x(\tfrac{p(x)}{N})^2, \ldots, -x^{k-1}(\tfrac{p(x)}{N})^2, \ldots, -x^{k-1}(\tfrac{p(x)}{N})^{m-1}) \end{aligned}$$

und berechnet:

$$r(x) = [c(x) \cdot H_1^{-1}]_{sh} \cdot ((H_2^{-1})^T)_{m\delta}$$

Im letzten Schritt wird jetzt die Nullstelle x_0 des Polynoms $r(x)$ (und gleichzeitig von $p(x)$) mit der Sturm-Sequenz berechnet. Da, wie schon gesagt, unter Umständen auch Nullstellen von $p(x)$ gefunden werden können, die betragsmäßig größer als X sind, sucht die Sturm-Sequenz im Intervall $-N$ bis N und nicht nur im Intervall $-X$ bis X. Das erhöht zwar die Laufzeit der Nullstellensuche, ist aber in Anbetracht der Tatsache, dass die Nullstellensuche, wie die Testergebnisse zeigen, jeweils nur einen verschwindend kleinen Bruchteil der Laufzeit benötigt, absolut vertretbar.

2.3 Testergebnisse zu Coppersmith im univariaten Fall

Eine mögliche Anwendung des in Abschnitt 2.1 vorgestellten Algorithmus' wird in Coppersmith[2] beschrieben. Betrachten wir das RSA-Verfahren und setzen den Verschlüsselungsexponenten e auf eine „kleine“ Zahl, z.B. $e = 3$, dann gilt für die Verschlüsselung c einer Nachricht $m \in \mathbb{Z}_N$ folglich:

$$c = m^3 \ mod \ N$$

Sei ein Teil der Nachricht bekannt. Den bekannten Teil bezeichnen wir mit B, den unbekannten Teil mit x_0, so dass:

$$m = B + x_0$$

Also ist:

$$c = (B + x_0)^3 \ mod \ N$$

Betrachten wir nun das Polynom:

$$p(x) = (B + x)^3 - c \ mod \ N$$

Dann können wir folglich mit dem Algorithmus aus Abschnitt 2.1, angewendet auf das Polynom $p(x)$, die Nullstelle x_0 berechnen, falls die Bedingung:

$$|x_0| < \frac{1}{2} N^{\frac{1}{\delta} - \varepsilon} = \frac{1}{2} N^{\frac{1}{3} - \varepsilon}$$

erfüllt ist. Ein großer Parameter m erlaubt dabei, näher an die Schranke $\frac{1}{2} N^{\frac{1}{3}}$ heranzugehen, wie an der exakteren Schranke:

$$X < N^{\frac{m-1}{m\delta - 1}} \cdot (m\delta)^{-\frac{1}{m\delta - 1}} \cdot 2^{-\frac{1}{2}}$$

aus Abschnitt 2.1 abgelesen werden kann. Im Prinzip funktioniert diese Anwendung natürlich auch mit einem größeren Verschlüsselungsexponenten e, allerdings beachte man die Rolle von $\delta = e$ bei der Bestimmung der oberen Schranke für X; bei großem δ können nur noch kleine unbekannte Teile der Nachricht mit dem in Abschnitt 2.1 vorgestellten Algorithmus bestimmt werden, da die Zahl $N^{\frac{1}{\delta}}$ bei zunehmend großem δ schnell klein wird. Wie in Coppersmith[2] werden wir die-

Anzahl Bits, x_0:	m:	Gitterdimension:	Laufzeit, L^3:	Laufzeit, gesamt:
64	2	9	11.03	11.79
128	3	15	880.71	886.54
139	4	21	6974.68	6988.85
140	5	27	40737.01	40761.71

Abbildung 2.4: Laufzeittabelle (in Sekunden) zu N als 512-Bit-Zahl

se Anwendung im Folgenden mit „Stereotyped Messages“ bezeichnen. Wir haben Tests für N als 128-Bit-Zahl (p und q jeweils 64-Bit-Zahlen), N als 512-Bit-Zahl (p und q jeweils 256-Bit-Zahlen) sowie N als 1024-Bit-Zahl (p und q jeweils 512-Bit-

Zahlen) durchgeführt. Bei allen Tests haben wir den Verschlüsselungsexponenten $e = 3$ benutzt. Wir haben mehrere „Pentium III"-Prozessoren mit einer Taktung von 1000 Mhz und einem Arbeitsspeicher von 512 Megabyte RAM verwendet. Abbildung 2.1 zeigt die Laufzeitkurve zu N als 128-Bit-Zahl, Abbildung 2.2 zu N als 512-Bit-Zahl, und Abbildung 2.3 zu N als 1024-Bit-Zahl. Man beachte

m:	Maximale Bitlänge, Test:	Maximale Bitlänge, Vorhersage:
2	103	93
3	128	117
4	139	126
5	>140	132

Abbildung 2.5: Vergleich Vorhersage mit Implementierung (N 512-Bit-Zahl)

die logarithmische Skalierung auf der Laufzeitachse. Die Laufzeit nimmt an den deutlich sichtbaren Sprungstellen jeweils stark zu. Der Grund dafür ist, dass an jeder dieser Sprungstellen der Parameter m erhöht werden muss, um den jeweiligen unbekannten Teil der Nachricht berechnen zu können. Die Zahl m ist dabei immer optimal, d.h. so klein wie möglich, gewählt. Ein größeres m bedeutet auch eine größere Gitterdimension und die Laufzeit des L^3-Unterprogrammes nimmt mit Abstand den größten Teil der Gesamtlaufzeit in Anspruch, siehe dazu Tabelle 2.4. Auch die Qualität der Schranke wurde getestet. Dazu haben wir zu einem gegebenen m den Wert $3 \cdot N^{\frac{m-1}{m\delta-1}} \cdot (m\delta)^{-\frac{1}{m\delta-1}} \cdot 2^{-\frac{1}{2}} \approx \log_2(10) \cdot N^{\frac{m-1}{m\delta-1}} \cdot (m\delta)^{-\frac{1}{m\delta-1}} \cdot 2^{-\frac{1}{2}}$ berechnet. So erhalten wir eine gute Näherung an die Anzahl von Bits, die mit dem entsprechenden m berechnet werden können, siehe dazu Tabelle 2.5.

Kapitel 3

Anwendung von Coppersmiths Algorithmus im bivariaten Fall

3.1 Coppersmiths Algorithmus zum bivariaten Fall

Der in diesem Kapitel vorgestellte Algorithmus hat den Zweck, einen RSA-Modulus $N \in \mathbb{Z}$ zu faktorisieren, also p und q zu finden, so dass:

$$N = p \cdot q \text{ und } p,\, q \text{ prim}$$

Angenommen einige der höherwertigen Bits von p sind bekannt. Bezeichne $\tilde{p}$ den bekannten Teil von p, die unbekannten Bits werden auf 0 gesetzt. Die Definition von $\tilde{q}$ ist ähnlich, wir setzen:

$$\tilde{q} = \left\lfloor \frac{N}{\tilde{p}} \right\rfloor$$

Wir werden die Faktorisierung des RSA-Modulus N auf die Berechnung der Nullstelle eines Polynoms zurückführen. Dazu betrachten wir das folgende Polynom:

$$p(x, y) = (\tilde{p} + x) \cdot (\tilde{q} + y) - N = xy + \tilde{p}y + \tilde{q}x + (\tilde{p}\tilde{q} - N)$$

Wäre die Nullstelle $x_0 = p - \tilde{p}$, $y_0 = q - \tilde{q}$ bekannt, könnte man die Zahlen $p = x_0 + \tilde{p}$ und $q = y_0 + \tilde{q}$ berechnen. Wir werden in diesem Kapitel einen Algorithmus vorstellen, der diese Nullstelle berechnen kann und zeigen, unter welchen Bedingungen diese Berechnung möglich ist. Wir beweisen in diesem Abschnitt das folgende Theorem:

Theorem 3.1.1. *Sei $N = p \cdot q$ ein RSA-Modulus, wobei p und q unbekannt sind. Weiter sei $\tilde{p}$ eine Approximation von p, so dass:*

$$p - \tilde{p} < N^{\frac{1}{4}}$$

Dann können wir in Laufzeit polynomiell in $(log(N), \frac{1}{\varepsilon})$ *ein Polynom* $Q(x)$ *berechnen, für das:*

$$Q(p - \tilde{p}) = 0$$

über den ganze Zahlen gilt.

Zum Beweis dieses Theorems werden wir ähnlich wie in Abschnitt 2.1 ein Gitter definieren und ein Polynom $r(x, y)$ berechnen, so dass $r(x_0, y_0) = 0$ gilt. Weiter berechnen wir $Q(x)$ als Resultante der Polynome $p(x, y)$ und $r(x, y)$ mit Hilfe der in Abschnitt 1.5 vorgestellten Methode. Dazu definieren wir:

$$X,Y \text{ mit } X,Y \leq N^{\frac{1}{4}}$$

Wir benutzen auch hier die schon in Abschnitt 2.1 eingeführten „Shifts“. Wir wählen ein $m \in \mathbb{Z}$ und definieren:

$$q_{ij}(x, y) = x^i y^j \cdot p(x, y) \text{ für } 0 \leq i, j < m$$

Mit Hilfe dieser Polynome werden wir eine Gittermatrix definieren. Allgemein ist die Dimension der Gittermatrix gegeben durch $(m+1)^2$, da in den Polynomen $q_{ij}(x, y)$ alle Monome $x^i y^j$, $0 \leq i, j \leq m$, vorkommen und für jedes vorkommende Monom benutzen wir eine eigene Zeile. Da die Gittermatrix quadratisch sein soll, benutzen wir die gleiche Anzahl an Spalten.
Die letzten m^2 Spalten werden mit den Koeffizienten der Polynome $q_{ij}(x, y)$ gefüllt. Dazu muss eine Ordnung auf den Monomen definiert werden, d.h. jedem Monom wird genau eine Zeile der Matrix zugewiesen. Wir können die Ordnungsfunktion F_m also als bijektive Abbildung betrachten:

$$F_m : \{0, \dots, m-1\} \times \{0, \dots, m-1\} \to \{1, \dots, m^2\}$$

Die Ordnungsfunktion wird zu einem Parameter m definiert, denn die Positionierung der Monome hängt auch von dem maximalen Grad der q_{ij} in x bzw. y ab. Die Matrix enthält den Koeffizienten zum Monom $x^l y^k$ des Polynoms $q_{ij}(x, y)$ an Position $M(F_{m+1}(l, k), (2m + 1) + F_m(i, j))$. Es gibt verschiedene Möglichkeiten, eine solche Ordnungsfunktion zu definieren. Wichtig ist, dass das resultierende Gitter immer Dreiecksgestalt hat, denn diese Eigenschaft werden wir im Folgenden benötigen. Es ist immer möglich, eine Ordnung zu finden, die das leistet. Die übrigen Einträge im rechten Block sind Null.
Die ersten $2m + 1$ Spalten enthalten die Einträge $(X^l Y^k)^{-1}$ für die Werte l, k,

denen durch die Ordnungsfunktion $F_{m+1}(l,k)$ die Werte m^2+1 bis $(m+1)^2$ zugeordnet werden. Sie stehen auf der Hauptdiagonalen. Die erste Zeile enthält dabei $(X^lY^k)^{-1}$ mit $F_{m+1}(l,k) = (m+1)^2$, die zweite Zeile $(X^sY^t)^{-1}$ mit $F_{m+1}(s,t) = (m+1)^2 - 1$, und so weiter. Die übrigen Einträge im linken Block sind Null.
Wir wählen beispielsweise $m = 2$. Zur Abkürzung setzen wir $c := \tilde{p}\tilde{q} - N$ und erhalten folgendes Gitter, in dem die Koeffizienten der Polynome $q_{ij}(x,y)$ die letzten vier Spalten füllen:

$$M = \begin{array}{c} \begin{array}{ccccc|cccc} \phantom{(X^2Y^2)^{-1}} & \phantom{(XY^2)^{-1}} & \phantom{(Y^2)^{-1}} & \phantom{(X^2Y)^{-1}} & \phantom{(X^2)^{-1}} & & q_{01} & & q_{00} \end{array} \\ \left(\begin{array}{ccccc|cccc} (X^2Y^2)^{-1} & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & (XY^2)^{-1} & 0 & 0 & 0 & \tilde{p} & 1 & 0 & 0 \\ 0 & 0 & (Y^2)^{-1} & 0 & 0 & 0 & \tilde{p} & 0 & 0 \\ 0 & 0 & 0 & (X^2Y)^{-1} & 0 & \tilde{q} & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & (X^2)^{-1} & 0 & 0 & \tilde{q} & 0 \\ 0 & 0 & 0 & 0 & 0 & c & \tilde{q} & \tilde{p} & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & c & 0 & \tilde{p} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & c & \tilde{q} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & c \end{array}\right) \\ \begin{array}{ccccc|cccc} \phantom{(X^2Y^2)^{-1}} & \phantom{(XY^2)^{-1}} & \phantom{(Y^2)^{-1}} & \phantom{(X^2Y)^{-1}} & \phantom{(X^2)^{-1}} & q_{11} & & q_{10} & \end{array} \end{array}$$

Im nächsten Schritt wird M ähnlich wie in Abschnitt 2.1 transformiert. Wir transformieren M mit Hilfe elementarer Zeilenoperationen in eine Matrix $\tilde{M}$, so dass $\tilde{M}$ im unteren rechten Block die Einheitsmatrix und im oberen rechten Block die Nullmatrix enthält. Auch hier können wir die Transformation durch eine Matrix ausdrücken, da nur elementare Zeilenoperationen verwendet worden sind. Wir definieren H_1 so, dass:

$$\tilde{M} = H_1 \cdot M$$

Den oberen linken Block von $\tilde{M}$ bezeichnen wir mit $\hat{M}$ und berechnen eine reduzierte Gitterbasis mit Hilfe des L^3-Algorithmus zu dieser Untermatrix. Die Matrix, welche die reduzierte Gitterbasis enthält, bezeichnen wir mit B. Auch hier können die Transformationsschritte des L^3-Algorithmus in Matrixform übersetzt werden. Wir definieren H_2 so, dass:

$$B = H_2 \cdot \hat{M}$$

Wir können jetzt mit Hilfe dieser Definitionen zeigen, wie, und gleichzeitig unter welchen Bedingungen für X und Y, das Polynom $r(x,y)$ berechnet werden kann. Die Determinante ist aufgrund der Dreiecksform von M relativ einfach als Produkt der Diagonalelemente zu berechnen. Die Matrix hat, wie gesagt, $(m+1)^2$ Spalten, davon enthalten m^2 Spalten das Diagonalelement c. Die übrigen $(m+1)^2 - m^2 = 2m+1$ Spalten haben die Form $(0,\ldots,0,X^{-i}Y^{-j},0,\ldots,0)$. Dabei tauchen in diesen Spalten alle Zahlen von 0 bis $m-1$ genau einmal als

Exponent von X bzw. von Y auf, sonst immer mit Exponenten m. Folglich ergeben die Diagonalelemente des oberen rechten Blocks aufmultipliziert:

$$(XY)^{-(m\cdot(2m+1)-\sum\limits_{i=1}^{m} i)} = (XY)^{-(2m^2+m-0.5\cdot(m^2-m))} = (XY)^{-1.5m^2-1.5m}$$

ergeben. Die Determinante von M ist folglich:

$$\det(M) = c^{m^2} \cdot (XY)^{-1.5m^2-1.5m}$$

Wir betrachten den Vektor:

$$c_0 = (x_0^m y_0^m, \ldots, x_0 y_0, y_0, x_0, 1)$$

und rechnen ähnlich wie auch in Abschnitt 2.1 (die Operation $[v]_{sh}$ verkürzt in diesem Fall einen Vektor der Länge $(m+1)^2$ zu einem Vektor der Länge $2m+1$):

$$\|c_0 \cdot M\| = \|(\overbrace{\frac{(x_0y_0)^m}{(XY)^m}, \ldots, \frac{x_0^m}{X^m}}^{2m+1\ Einträge}, 0, \ldots, 0)\|$$

Da $\frac{|x_0|}{X} \leq 1$:

$$\|(\overbrace{\frac{(x_0y_0)^m}{(XY)^m}, \ldots, \frac{x_0^m}{X^m}}^{2m+1\ Einträge}, 0, \ldots, 0)\| \leq \|(\overbrace{1, \ldots, 1,}^{(2m+1)-mal}\ 0, \ldots, 0)\| = \sqrt{2m+1}$$

Also haben wir:

$$\sqrt{2m+1} \geq \|c_0 \cdot M\| = \|c_0 \cdot H_1^{-1} \cdot \tilde{M}\|$$

Analog zu dem entsprechenden Argument in Abschnitt 2.1 gilt $(c_0 H_1^{-1})_i = 0$ für $i > 2m+1$, da die letzten m^2 Einträge von $c_0 H_1^{-1} \tilde{M}$ alle gleich 0 sind. Also:

$$\|c_0 \cdot H_1^{-1} \cdot \tilde{M}\| = \|[c_0 \cdot H_1^{-1}]_{sh} \cdot \hat{M}\| = \|[c_0 \cdot H_1^{-1}]_{sh} \cdot H_2^{-1} \cdot B\|$$

Wieder sei $c' := [c_0 \cdot H_1^{-1}]_{sh} \cdot H_2^{-1}$, und wir erhalten:

$$\|[c_0 \cdot H_1^{-1}]_{sh} \cdot H_2^{-1} \cdot B\| = \|c' \cdot B\|$$

Wir definieren B^* als die Gram-Schmidt-Orthogonalisierung der Basis B und schreiben wieder $OB = B^*$ mit der Orthogonalisierungsmatrix O. Wir erhalten:

$$\|c' \cdot B\| = \|c' \cdot O^{-1} B^*\|$$

Mit $c'' = c' \cdot O^{-1}$:

$$\|c' \cdot O^{-1}B^*\| = \|c'' \cdot B^*\| \geq \|c''_{2m+1} \cdot b^*_{2m+1}\|$$

Wir haben in Abschnitt 1.2 gezeigt, dass $c'_{2m+1} = c''_{2m+1} \in \mathbb{Z}$, und so erhalten wir:

$$\|c''_{2m+1} \cdot b^*_{2m+1}\| = \|c'_{2m+1} \cdot b^*_{2m+1}\| = |c'_{2m+1}| \cdot \|b^*_{2m+1}\|$$

Wir benutzen $\|b^*_{2m+1}\| \geq 2^{-\frac{(2m+1)-1}{4}} \cdot |\det(M)|^{\frac{1}{2m+1}}$, eine Eigenschaft der reduzierten Gitterbasis, siehe dazu Abschnitt 1.3:

$$|c'_{2m+1}| \cdot \|b^*_{2m+1}\| \geq |c'_{2m+1}| \cdot 2^{-\frac{(2m+1)-1}{4}} \cdot |\det(M)|^{\frac{1}{2m+1}}$$

Also insgesamt:

$$\sqrt{2m+1} \geq |c'_{2m+1}| \cdot 2^{-\frac{1}{2}m} \cdot |\det(M)|^{\frac{1}{2m+1}}$$

$$\Leftrightarrow \frac{1}{|c'_{2m+1}|} \geq \frac{1}{\sqrt{2m+1}} \cdot 2^{-\frac{1}{2}m} \cdot |\det(M)|^{\frac{1}{2m+1}}$$

$$\Leftrightarrow |c'_{2m+1}| \leq \sqrt{2m+1} \cdot 2^{\frac{1}{2}m} \cdot |\det(M)|^{-\frac{1}{2m+1}}$$

Da $c'_{2m+1} \in \mathbb{Z}$, gilt $c'_{2m+1} = 0$, falls $|c'_{2m+1}| < 1$. Wir fordern:

$$|c'_{2m+1}| \leq \sqrt{2m+1} \cdot 2^{\frac{1}{2}m} \cdot |\det(M)|^{-\frac{1}{2m+1}} < 1$$

und haben also:

$$\sqrt{2m+1} \cdot 2^{\frac{1}{2}m} \cdot |\det(M)|^{-\frac{1}{2m+1}} < 1$$

$$\Leftrightarrow \sqrt{2m+1} \cdot 2^{\frac{1}{2}m} < |\det(M)|^{\frac{1}{2m+1}}$$

$$\Leftrightarrow \sqrt{2m+1} \cdot 2^{\frac{1}{2}m} < (|c|^{m^2} \cdot (XY)^{-1.5m^2-1.5m})^{\frac{1}{2m+1}}$$

Umstellen und Vernachlässigung von Termen niederer Ordnung führt zu:

$$(XY)^{\frac{3}{4}m} < (\tfrac{|c|}{2})^{\frac{1}{2}m} \cdot (XY)^{-\frac{3}{4}}$$

$$\Leftrightarrow (XY)^{\frac{3}{4}} < (\tfrac{|c|}{2})^{\frac{1}{2}} \cdot (XY)^{-\frac{3}{4m}}$$

$$\Leftrightarrow XY < (\tfrac{|c|}{2})^{\frac{2}{3}} \cdot (XY)^{-\frac{1}{m}}$$

Wieder vernachlässigen wir Terme niederer Ordnung:

$$XY < |c|^{\frac{2}{3}} \cdot \frac{1}{(XY)^{\frac{1}{m}}}$$

$$\Leftrightarrow (XY)^{1+\frac{1}{m}} < |c|^{\frac{2}{3}}$$

$$\Leftrightarrow XY < |c|^{\frac{2m}{3m+3}}$$

Ähnlich wie in Abschnitt 2.1 können wir allerdings auch in diesem Fall den Vektor c_0 nicht direkt berechnen, da x_0 und y_0 im Vorfeld unbekannt sind. Wir definieren stattdessen wieder einen Variablenvektor $c(x, y)$:

$$c(x, y) = (x^m y^m, \ldots, xy, y, x, 1)$$

Wir können jetzt anstelle von $c' := [c_0 \cdot H_1^{-1}]_{sh} \cdot H_2^{-1}$ das bivariate Polynom $r(x, y) := [c(x, y) \cdot H_1^{-1}]_{sh} \cdot ((H_2^{-1})^T)$ berechnen und wissen durch den gerade mit c_0 anstelle von $c(x, y)$ geführten Beweis, dass folglich $r(x_0, y_0) = 0$ gilt. Wir berechnen:

$$Q(x) := Resultante_y(r(x, y), p(x, y))$$

Die Polynome $r(x, y)$ und $p(x, y)$ sind teilerfremd. Wir wollen diese Tatsache an dieser Stelle nicht beweisen. Ein Resultat aus Mignotte[8] besagt, dass, da $r(x, y)$ und $p(x, y)$ teilerfremd sind, die Resultante $Q(x)$ dieser beiden Polynome ungleich dem Nullpolynom sein muss. Es gilt weiter $Q(x_0) = 0$, und wir können x_0 mit der Sturm-Sequenz bestimmen. Setzen wir x_0 für x in das Polynom $p(x, y)$ ein, erhalten wir das univariate Polynom $p(x_0, y)$ mit der Nullstelle y_0, welche wir dann ebenfalls mit der Sturm-Sequenz ausrechnen können. Wir haben gezeigt, dass:

$$XY < |c|^{\frac{2m}{3m+3}}$$

und $|x_0| < X$ sowie $|y_0| < Y$ gelten muss. Mit wachsendem m wächst auch der Ausdruck $|c|^{\frac{2m}{3m+3}}$ und geht gegen $\sqrt{c}$, und wir erhalten eine bessere obere Schranke für XY. Allerdings verlängert sich damit auch die Laufzeit des Algorithmus', da mit zunehmend großem m die Gitterdimension wächst und somit die Berechnung einer reduzierten Gitterbasis mehr Rechenzeit beansprucht. Diese Schranke leistet für die praktische Anwendung gute Dienste, mit ihr lässt sich gut einschätzen, wie viele unbekannte Bits der Primfaktoren mit einem bestimmten m noch berechnet werden können. Wir wollen allerdings noch eine kleine Ergänzung angeben und die Schranke abhängig von N und m formulieren. Setzen wir:

$$X, Y := N^{\gamma}$$

Wir wollen jetzt γ möglichst groß wählen, aber immer noch die korrekte Berechnung der Nullstelle (x_0, y_0) garantieren können. Es ist:

$$p(x_0, y_0) = 0$$

$$\Leftrightarrow c = -(\overbrace{x_0 y_0}^{\approx N^{\frac{1}{2}}} + \overbrace{\tilde{p} y_0}^{\approx N^{\frac{1}{2}+\gamma}} + \overbrace{\tilde{q} x_0}^{\approx N^{\frac{1}{2}+\gamma}})$$

Damit ist also:

$$\lim_{n\to\infty}(|c|^{\frac{2m}{3m+3}}) = c^{\frac{2}{3}} \approx (N^{\frac{1}{2}+\gamma})^{\frac{2}{3}}$$

Daraus folgt mit $XY < |c|^{\frac{2m}{3m+3}}$:

$$XY = N^{2\gamma} \leq (N^{\frac{1}{2}+\gamma})^{\frac{2}{3}} \approx c^{\frac{2}{3}}$$

$$\Rightarrow 2\gamma \leq (\tfrac{1}{2} + \gamma) \cdot \tfrac{2}{3}$$

$$\Leftrightarrow 2\gamma \leq \tfrac{1}{3} + \tfrac{2}{3}\gamma$$

$$\Leftrightarrow \tfrac{4}{3}\gamma \leq \tfrac{1}{3}$$

$$\Leftrightarrow \gamma \leq \tfrac{1}{4}$$

Wir haben also:

$$XY < N^{\frac{1}{4}} \cdot N^{\frac{1}{4}} = N^{\frac{1}{2}}$$

Diese Schranke hat allerdings eher Nutzen in der Theorie, da mit ihr keine Aussage darüber getroffen werden kann, wie groß der Parameter m gewählt werden muss, um die Nullstelle (x_0, y_0) berechnen zu können.

3.2 Implementierung von Coppersmith im bivariaten Fall

Auch der in Abschnitt 3.1 beschriebene Algorithmus ist in dem für diese Arbeit erstellten Softwarepaket enthalten. Die Signatur der entsprechenden *C/C++*-Funktion bivariatePolynomialRoots, hier abgekürzt BPR, sieht folgendermaßen aus:

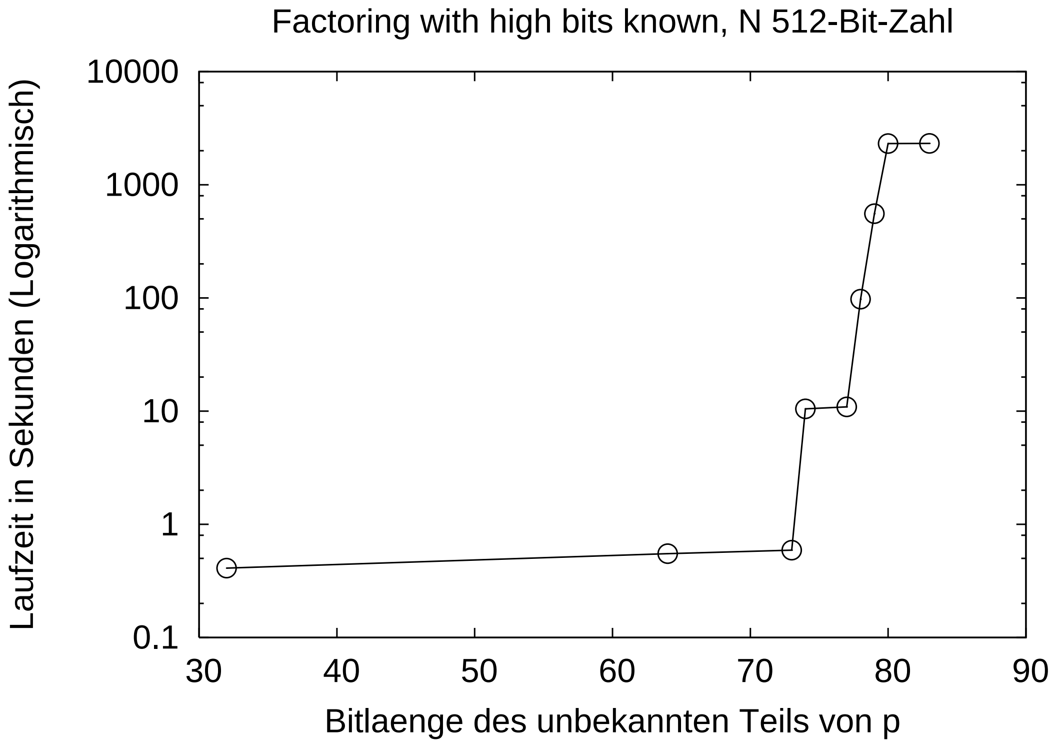

Abbildung 3.1: Laufzeitkurve zur Faktorisierung mit Coppersmith (N 512-Bit-Zahl)

```
long BPR(ZZ* coefficients, ZZ* root, ZZ X, ZZ Y, long deltaX, long deltaY, long m, int out)
```

Sei $p(x, y)$ das Polynom, dessen Nullstellen berechnet werden sollen. Sei δ_x der Grad von $p(x, y)$ in x und sei δ_y der Grad von $p(x, y)$ in y. Die Signatur setzt sich zusammen aus:

- Der erste Parameter `coeff` ist ein Pointer auf ein Array der Länge $(\delta_x + 1) \cdot (\delta_y + 1)$, mit dem Koeffizienten zu $x^i y^j$ von $p(x, y)$ an Position $i \cdot (\delta_y + 1) + j$.
- Der zweite Parameter `roots` erwartet einen Pointer auf ein Array der Länge $\delta_x \cdot \delta_y$, hier werden im Verlauf des Programms die gefundenen Nullstellen abgespeichert und können nach beendeter Berechung ausgelesen werden.
- Der dritte Parameter X entspricht der oberen Schranke für die erste Komponente x_0 der gesuchten Nullstelle (x_0, y_0).
- Der vierte Parameter Y entspricht der oberen Schranke für die erste Komponente y_0 der gesuchten Nullstelle (x_0, y_0).

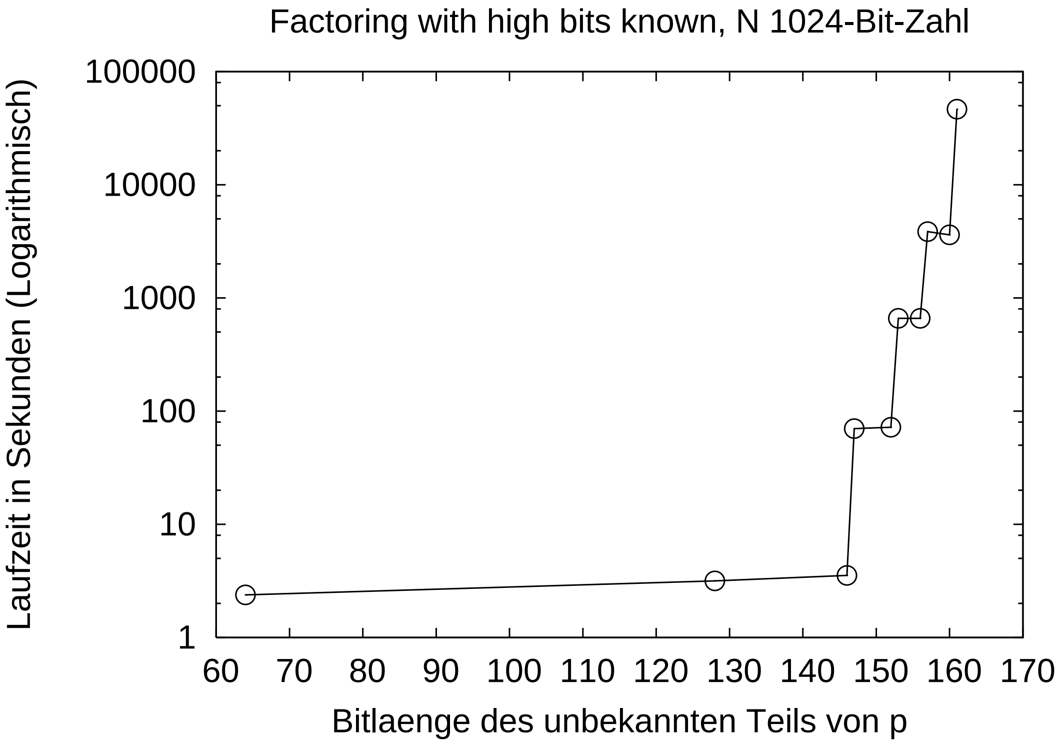

Abbildung 3.2: Laufzeitkurve zur Faktorisierung mit Coppersmith (N 1024-Bit-Zahl)

- Der fünfte Parameter deltaX ist für die Übergabe von δ_x bestimmt.
- Der sechste Parameter deltaY ist analog für die Übergabe von δ_y vorgesehen.
- Der siebte Parameter m entspricht der Beschreibung in Abschnitt 3.1.
- Der achte und letzte Parameter out bestimmt, ob der Nutzer die Ausgabe der im Verlaufe des Algorithmus benötigten Matrizen, Vektoren und Polynome sehen möchte. Setzung von out $= 1$ führt zur Ausgabe der Zwischenergebnisse, out $\neq 1$ unterdrückt die Ausgabe.

Auch bei diesem Programm ist es möglich, dass der Algorithmus die Nullstelle (x_0, y_0) finden kann, obwohl nicht $|x_0| < X$ und $|y_0| < Y$ gilt. Die Matrix M wird wie in Abschnitt 3.1 erklärt und mit Hilfe elementarer Zeilenoperationen in eine Matrix $\tilde{M}$ überführt, so dass $\tilde{M}$ später im unteren rechten Block die Einheitsmatrix und im oberen rechten Block die Nullmatrix enthält. Wir drücken die Transformation auch hier durch Multiplikation mit einer Matrix H_1 aus:

$$\tilde{M} = H_1 \cdot M$$

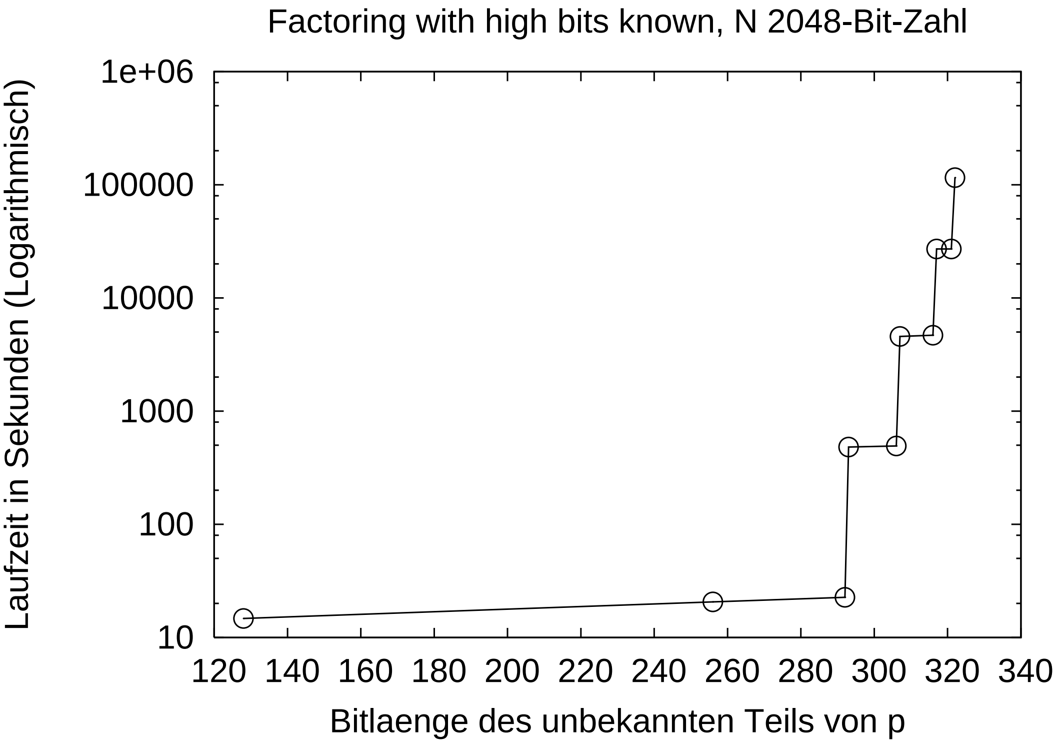

Abbildung 3.3: Laufzeitkurve zur Faktorisierung mit Coppersmith (N 2048-Bit-Zahl)

H_1 wird dabei berechnet, indem man jeden Transformationsschritt in Matrixform übersetzt und mit einer Matrix multipliziert, in der alle bisher gemachten Transformationsschritte gespeichert sind. Anfangs ist diese Matrix einfach die Einheitsmatrix und zum Ende der Transformation entspricht sie H_1.
Aus der transformierten Matrix $\tilde{M}$ wird der obere linke $(2m+1)\times(2m+1)$-Block selektiert. Die Brüche in diesem Block werden durch Multiplikation mit der Zahl X^mY^m eliminiert, denn auch in diesem Programm wird im nächsten Schritt eine reduzierte Gitterbasis mit dem L^3-Unterprogramm aus NTL zu diesem Block berechnet, wobei die Eingabematrix nur ganzzahlige Einträge enthalten darf. Den oberen linken Block von $\tilde{M}$ bezeichnen wir mit $\hat{M}$. Die reduzierte Gitterbasis zu $\hat{M}$ wird mit B bezeichnet, die L^3-Transformationsmatrix mit H_2, so dass:

$$B = H_2 \cdot \hat{M}$$

gilt. Das L^3-Unterprogramm berechnet ebenfalls H_2 während der Erzeugung der reduzierten Gitterbasis. Die Inversen von H_1 und H_2 werden mit Hilfe von entsprechenden NTL-Methoden bestimmt. Der Vektor:

$$c(x,y) = (x^m y^m, \ldots, y, x, 1)$$

wird erzeugt und im nächsten Schritt berechnet das Programm:

$$r(x,y) = [c(x,y) \cdot H_1^{-1}]_{sh} \cdot ((H_2^{-1})^T)_{h\delta}$$

Dann wird die Resultante $Q(x)$ von $r(x,y)$ und $p(x,y)$ bestimmt. Mit einem Resultat aus Mignotte[8] wissen wir, dass $Q(x)$ ungleich dem Nullpolynom ist, da $r(x,y)$ und $p(x,y)$ teilerfremd sind. Mit der Sturm-Sequenz wird x_0 als Nullstelle von $Q(x)$ berechnet und diese Nullstelle für x in das Polynom $p(x,y)$ eingesetzt. Wir erhalten das jetzt univariate Polynom $p_{x_0}(y)$, bestimmen dessen Nullstelle y_0 ebenfalls mit der Sturm-Sequenz und haben so die Nullstelle (x_0, y_0) von $p(x,y)$ berechnet.

3.3 Testergebnisse zu Coppersmith im bivariaten Fall

Wir können den in Abschnitt 3.1 vorgestellten Algorithmus benutzen, um die Faktorisierung eines RSA-Modulus zu berechnen. Es sei $N \in \mathbb{Z}$ mit:

$$N = p \cdot q \text{ und } p,\ q \text{ prim}$$

p und q seien dabei unbekannt. Angenommen wir kennen eine begrenzte An-

Anzahl Bits, x_0:	m:	Gitterdimension:	Laufzeit, L^3:	Laufzeit, gesamt:
64	2	9	0.47	0.59
74	3	16	9.97	10.46
78	4	25	95.41	97.59
79	5	36	545.95	555.77
83	6	49	2277.78	2317.55

Abbildung 3.4: Laufzeittabelle (Laufzeit in Sekunden) zu N als 512-Bit-Zahl

zahl der höherwertigen Bits von p. Die bekannten Bits von p interpretieren wir als die Zahl $\tilde{p}$ und setzen:

$$\tilde{q} := \left\lceil \frac{N}{\tilde{p}} \right\rceil$$

Damit gilt $\tilde{p}, \tilde{q} \in \mathbb{Z}$. Weiter definieren wir das Polynom:

$$p(x,y) = (\tilde{p} + x) \cdot (\tilde{q} + y) - N = xy + \tilde{p}y + \tilde{q}x + \tilde{p}\tilde{q} - N$$

Können wir eine Nullstelle (x_0, y_0), $x_0, y_0 \in \mathbb{Z}$, bestimmen, so haben wir auch die Primfaktorisierung von N berechnet, da folglich gilt:

$$(\tilde{p} + x_0) \cdot (\tilde{q} + y_0) - N = 0$$

$$\Leftrightarrow (\tilde{p} + x_0) \cdot (\tilde{q} + y_0) = N$$

$$\Leftrightarrow (\tilde{p} + x_0) \cdot (\tilde{q} + y_0) = p \cdot q$$

Da $x_0, y_0, \tilde{p}, \tilde{q} \in \mathbb{Z}$, ist $\tilde{p} + x_0 = p$ und $\tilde{q} + y_0 = q$, und wir haben die Primfaktorisierung von N bestimmt. Nach der in Abschnitt 3.1 errechneten Schranke wissen wir, dass wir die Nullstelle (x_0, y_0) berechnen können, falls die Bedingung:

$$x_0 y_0 \leq XY < |c|^{\frac{2m}{3m+3}}$$

eingehalten wird. Wir haben dazu Tests für N als 512-Bit-Zahl (p und q jeweils 256-Bit-Zahlen), N als 1024-Bit-Zahl (p und q jeweils 512-Bit-Zahlen) sowie N als 2048-Bit-Zahl (p und q jeweils 1024-Bit-Zahlen) durchgeführt. Als Testrechner wurden mehrere „Pentium III“-Prozessoren mit einer Taktung von 1000 Mhz und einem Arbeitsspeicher von 512 Megabyte RAM verwendet. Wie auch in Abschnitt

m:	Maximale Bitlänge, Test:	Maximale Bitlänge, Vorhersage:
2	73	78
3	77	87
4	78	93
5	79	96
6	83	99

Abbildung 3.5: Vergleich Vorhersage mit Implementierung (N 512-Bit-Zahl)

2.3 wird an den Laufzeitkurven auf Abbildung 3.1, Abbildung 3.2 und Abbildung 3.3 deutlich, dass ein wachsender Parameter m die Laufzeit schnell ansteigen lässt. Auch hier haben wir die Laufzeitachse zu diesen Graphen logarithmisch skaliert. Die Größenordnung der Laufzeiten lassen sich besser in Tabelle 3.4 ablesen. Außerdem wird auch bei dieser Anwendung deutlich, wie sehr der L^3-Algorithmus die Gesamtlaufzeit dominiert. Wieder ist der Parameter m bei allen Testläufen optimal, das heißt so klein wie möglich, gewählt worden. Die Schranke erlaubt es, recht genau zu berechnen, wie viele unbekannte Bits mit einem bestimmten m berechnet werden können. Wir haben dazu $3 \cdot \log_2(|\tilde{p}\tilde{q} - N|^{\frac{2m}{3m+3}}) \approx \log_2(10) \cdot \log_2(|c|^{\frac{2m}{3m+3}})$ berechnet und erhalten so eine Näherung an die Anzahl der unbekannten Bits, die mit entsprechenden m berechnet werden können (siehe Tabelle 3.5).

Kapitel 4

Anwendung von Corons Algorithmus im bivariaten Fall

4.1 Corons Algorithmus im bivariaten Fall

Auch dieses Kapitel beschäftigt sich, wie das vorangegangene, mit einem Algorithmus, der im Rahmen dieser Arbeit zur Faktorisierung eines RSA-Modulus eingesetzt wurde. Er wurde nur zu diesem Zweck implementiert und getestet, bietet allerdings noch mehr Möglichkeiten. Der Algorithmus wurde erstmalig in Coron[4] vorgestellt. Er basiert auf den folgenden Resultaten:

Lemma 4.1.1. *(Howgrave-Graham)*
Sei $h(x, y) \in \mathbb{Z}[x, y]$ ein Polynom mit maximal ω verschiedenen Monomen. Angenommen $h(x_0, y_0) = 0 \ mod \ n$ mit $|x_0| < X$, $|y_0| < Y$ und $\|h(x, y)\| < \frac{n}{\sqrt{w}}$. Dann gilt $h(x_0, y_0) = 0$ über den ganzen Zahlen.

Beweis:
Wir haben:

$$|h(x_0, y_0)| = |\sum h_{ij} \cdot x_0^i y_0^j| = |\sum h_{ij} X^i Y^j (\tfrac{x_0}{X})^i (\tfrac{y_0}{Y})^j|$$

Die Dreiecksungleichung liefert:

$$|\sum h_{ij} X^i Y^j (\tfrac{x_0}{X})^i (\tfrac{y_0}{Y})^j| \leq \sum |h_{ij} X^i Y^j (\tfrac{x_0}{X})^i (\tfrac{y_0}{Y})^j|$$

Mit $|x_0| < X$ und $|y_0| < Y$:

$$\sum |h_{ij} X^i Y^j (\tfrac{x_0}{X})^i (\tfrac{y_0}{Y})^j| \leq \sum |h_{ij} X^i Y^j| \leq \sqrt{\omega} \|h(xX, yY)\| < n$$

Nach Voraussetzung ist $h(x_0, y_0) = 0 \ mod \ n$, und wir haben gerade gezeigt, dass $|h(x_0, y_0)| < n$. Damit folgt sofort, dass $h(x_0, y_0) = 0$. □

Lemma 4.1.2.
Seien $a(x,y), b(x,y) \in \mathbb{Z}[x,y]$ Polynome $\neq 0$ vom Grad maximal ω in je einer der Variablen x und y, so dass $b(x,y)$ ein Vielfaches von $a(x,y)$ in $\mathbb{Z}[x,y]$ ist. Außerdem sei $a(0,0) \neq 0$ und $b(x,y)$ teilbar durch eine Zahl $r \neq 0$, so dass $ggT(r, a(0,0)) = 1$. Dann ist $b(x,y)$ teilbar durch $r \cdot a(x,y)$ und es gilt:

$$\|b\| \geq 2^{-(\omega+1)^2} \cdot \|a\|_\infty$$

Beweis:
Um dieses Lemma zu beweisen benötigen wir ein Resultat von Mignotte[9]. Dieses Resultat besagt folgendes:

Mignotte Monovariat:
Seien $f(x), g(x) \in \mathbb{Z}[x]$ mit $f(x), g(x) \neq 0$ und $deg(f) = k$ sowie $f(x)$ teilt $g(x)$. Dann gilt:

$$g \geq 2^{-k} \cdot \|f\|_\infty$$

Wir werden dieses Resultat nun auf den bivariaten Fall übertragen. Seien dazu $a(x,y), b(x,y) \in \mathbb{Z}[x,y]$, $a(x,y), b(x,y) \neq 0$, mit Grad höchstens d jeweils in x und y und $a(x,y)$ teilt $b(x,y)$. Wir setzen $f(x) := a(x, x^{d+1})$ und $g(x) := b(x, x^{d+1})$. Damit wissen wir, dass $deg(f), deg(g) \leq (d+1)^2$ und $\|f\|_\infty = \|a\|_\infty$ sowie $\|g\|_\infty = \|b\|_\infty$. Außerdem wissen wir: f teilt g, weil nach Voraussetzung gilt: a teilt b. Folglich können wir *Mignotte Monovariat* auf die Polynome $f(x)$ und $g(x)$ anwenden und erhalten insgesamt:

Mignotte Bivariat:
Seien nun $a(x,y), b(x,y) \in \mathbb{Z}[x,y]$, $a(x,y), b(x,y) \neq 0$, mit Grad höchstens d jeweils in x und y und $a(x,y)$ teilt $b(x,y)$. Dann gilt:

$$\|b\| \geq 2^{-(d+1)^2} \cdot \|a\|_\infty$$

Mit Hilfe dieses Resultats können wir jetzt Lemma 4.1.2 beweisen. Sei $\lambda(x,y) \in \mathbb{Z}[x,y]$ das Polynom, für das $a(x,y) \cdot \lambda(x,y) = b(x,y)$ gilt. Wir zeigen zuerst: r teilt $\lambda(x,y)$.
Nehmen wir das Gegenteil an: r teilt nicht $\lambda(x,y)$. Wir definieren λ_{ij} als den Koeffizienten des Polynoms $\lambda(x,y)$ zu dem Monom $x^i y^j$. Dann gibt es einen Koeffizienten λ_{ij}, für den gilt: r teilt nicht λ_{ij}. Wir nehmen an, dass λ_{ij} der erste der Koeffizienten mit dieser Eigenschaft in lexikografisch aufsteigender Reihenfolge nach den Paaren (i,j) sortiert ist. Wir definieren b_{ij} als den Koeffizienten des Polynoms $b(x,y)$ zu dem Monom $x^i y^j$. Da λ_{ij} der erste der Koeffizienten in lexikografisch aufsteigender Reihenfolge ist, ist b_{ij} *mod* r von $a(0,0)$ abhängig, nicht aber von den anderen Koeffizienten von $a(x,y)$, denn dann müsste das zugehörige

λ_{lk} in der lexikografischen Reihenfolge vor λ_{ij} auftauchen. Wir erhalten also:

$$b_{ij} = \lambda_{ij} \cdot a(0,0) \; mod \; r$$

Da $ggT(r, a(0,0)) = 1$ gilt, ist a invertierbar *modulo* r und wir haben:

$$a(0,0)^{-1} \cdot b_{ij} = \lambda_{ij} \; mod \; r$$

Da $b_{ij} = 0 \; mod \; r$:

$$\lambda_{ij} = 0 \; mod \; r$$

Das aber ist ein Widerspruch zu der Annahme: r teilt nicht $\lambda(x,y)$. Folglich muss diese Annahme falsch sein, es gilt folglich: r teilt $\lambda(x,y)$. Damit erhalten wir sofort: $r \cdot a(x,y)$ teilt $b(x,y)$. Wenden wir nun *Mignotte Bivariat* auf die Polynome $r \cdot a(x,y)$ und $b(x,y)$ an, erhalten wir Lemma 4.1.2. □

Sei $p(x,y)$ ein bivariates Polynom mit Grad δ_x in x und Grad δ_y in y. Wir werden ein Polynom $h(x,y)$ berechnen, so dass, ähnlich wie auch in Abschnitt 3.1 beschrieben, $h(x,y)$ und $p(x,y)$ teilerfremd sind. Für die Resultante $Q(x)$ dieser beiden Polynome gilt damit $Q(x_0) = 0$, und die Nullstelle (x_0, y_0) kann mit ein paar weiteren Schritten bestimmt werden. Wir beginnen mit der Berechnung von $h(x,y)$. Dazu sei δ definiert mit:

$$\delta := max(\delta_y, \delta_y)$$

Also ist:

$$p(x,y) = \sum_{i=0}^{\delta} \sum_{j=0}^{\delta} p_{ij} x^i y^j$$

Wir setzen voraus, dass $p_{00} \neq 0$. Sei (x_0, y_0) mit $p(x_0, y_0) = 0$ eine Nullstelle von $p(x,y)$. Wir definieren:

$$W := \|p(xX, yY)\|_\infty \qquad u := W + ((1-W) \; mod \; |p_{00}|) \qquad n := u \cdot (XY)^{m-1}$$

Außerdem verlangen wir $ggT(p_{00}, XY) = 1$. Falls $ggT(p_{00}, XY) > 1$, generieren wir zwei Primzahlen X' und Y' mit $X < X' < 2X$ und $Y < Y' < 2Y$, so dass $ggT(p_{00}, X') = 1$ und $ggT(p_{00}, Y') = 1$ gilt. Damit folgt sofort $ggT(p_{00}, X'Y') = 1$, und wir nutzen im weiteren Verlauf des Algorithmus' X' an Stelle von X und Y' an Stelle von Y. Weiter definieren wir:

$$q(x,y) = p_{00}^{-1} \cdot p(x,y) \; mod \; n$$

$$= 1 + \sum_{(i,j)\neq(0,0)} a_{ij} x^i y^j$$

und die „Shifts“:

$$q_{ij}(x,y) = x^i y^j X^{m-i-1} Y^{m-j-1} q(x,y) \text{ für } 0 \leq i,j < m$$

$$q_{ij}(x,y) = x^i y^j n \text{ für } (i,j) \in [0, \delta + m - 1]^2 \backslash [0, m-1]^2$$

Also erhalten wir insgesamt $\omega = (\delta + m)^2$ Polynome $q_{ij}(x,y)$. Zu jedem dieser Polynome definieren wir:

$$\tilde{q}_{ij}(x,y) := q_{ij}(xX, yY)$$

und formen aus den Koeffizienten der $\tilde{q}_{ij}(x,y)$ ein Gitter M der Dimension ω. Dazu benötigen wir, wie in Abschnitt 3.1, eine passende Ordnungsfunktion (in dieser Arbeit wurde für den in diesem Abschnitt und den in Abschnitt 3.1 vorgestellten Algorithmus die gleiche Ordnungsfunktion verwendet). Für $\delta = 1$ und $m = 2$ erhalten wir beispielsweise:

$$M = \begin{pmatrix}
XY & a_{10}X^2Y & a_{01}XY^2 & a_{11}X^2Y^2 & 0 & 0 & 0 & 0 & 0 \\
0 & XY & 0 & a_{01}XY^2 & a_{10}X^2Y & a_{11}X^2Y^2 & 0 & 0 & 0 \\
0 & 0 & XY & a_{10}X^2Y & 0 & 0 & a_{01}XY^2 & a_{11}X^2Y^2 & 0 \\
0 & 0 & 0 & XY & 0 & a_{10}X^2Y & 0 & a_{01}XY^2 & a_{11}X^2Y^2 \\
0 & 0 & 0 & 0 & X^2n & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & X^2Yn & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & Y^2n & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & XY^2n & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X^2Y^2n
\end{pmatrix}$$

Auch dieses Mal ist die Determinante des Gitters einfach zu berechnen, da L Dreiecksgestalt hat. Es ist:

$$\det(L) = (XY)^{\frac{(\delta+m-1)(\delta+m)^2+(m-1)m^2}{2}} \cdot n^{\delta(\delta+2m)}$$

Wir benutzen jetzt den L^3-Algorithmus, um eine reduzierte Gitterbasis zu L zu berechnen. Bezeichne n die Dimension von L. Wir benutzen die in Abschnitt 1.3 erwähnte Eigenschaft des L^3-Algorithmus', dass:

$$\|b_1\| \leq 2^{\frac{n-1}{4}} \cdot \det(L)^{\frac{1}{n}}$$

Wir können also einen Vektor berechnen, der einem Polynom $h(xX, yY)$ entspricht, so dass:

$$\|h(xX, yY)\| \leq 2^{\frac{\omega-1}{4}} \cdot \det(L)^{\frac{1}{\omega}}$$

Wir machen nun eine Voraussetzung, die später relevant wird, um eine Schranke für XY errechnen zu können. Es gelte:

$$2^{\frac{\omega-1}{4}} \cdot \det(L)^{\frac{1}{\omega}} < 2^{-\omega} \cdot (XY)^{m-1} \cdot W$$

Damit gilt auch:

$$\|h(xX, yY)\| < 2^{-\omega} \cdot (XY)^{m-1} \cdot W$$

Mit Lemma 4.1.2 erhalten wir, dass $h(x,y)$ kein Vielfaches von $p(x,y)$ ist. Jetzt wollen wir zeigen, dass auch Lemma 4.1.1 angewendet werden kann. Betrachten wir dazu:

$$n = u \cdot (XY)^{m-1} = (W + \underbrace{((1-W) \ mod \ p_{00})}_{\geq 0}) \cdot (XY)^{m-1} \geq W \cdot (XY)^{m-1}$$

Da $\frac{\sqrt{\omega}}{2^\omega} < 1$ für $\omega \geq 1$, erhalten wir:

$$W \cdot (XY)^{m-1} > \tfrac{\sqrt{\omega}}{2^\omega} \cdot W \cdot (XY)^{m-1}$$

Damit haben wir:

$$2^{-\omega} \cdot W \cdot (XY)^{m-1} < \tfrac{n}{\sqrt{\omega}}$$

Insgesamt haben wir gezeigt:

$$\|h(xX, yY)\| \leq 2^{\frac{\omega-1}{4}} \cdot \det(L)^{\frac{1}{\omega}} < 2^{-\omega} \cdot W \cdot (XY)^{m-1} < \tfrac{n}{\sqrt{\omega}}$$

Also können wir auch Lemma 4.1.1 anwenden und haben $h(x_0, y_0) = 0$. Da $p(x,y)$ nach Voraussetzung irreduzibel ist, erhalten wir mit:

$$Q(x) := Resultante_y(h(x,y), p(x,y))$$

ein Polynom ungleich dem Nullpolynom, so dass $Q(x_0) = 0$ mit Lemma 4.1.1 gelten muss. Weiter wissen wir mit Lemma 4.1.2, dass $h(x,y)$ und $p(x,y)$ teilerfremd sind. Folglich ist $Q(x)$ ungleich dem Nullpolynom, da $h(x,y)$ und $p(x,y)$ teilerfremd sind und ein Resultat aus Mignotte[8] besagt, dass die Resultante dieser beiden Polynome damit ungleich dem Nullpolynom sein muss. Mit der Sturm-Sequenz können wir die Nullstelle x_0 berechnen. Setzen wir x_0 in $p(x,y)$ ein, erhalten wir das univariate Polynom $p(x_0, y)$ und können mit der Sturm-Sequenz y_0 bestimmen. Damit haben wir eine Nullstelle von $p(x,y)$ berechnet. Wir werden jetzt eine Schranke für XY bestimmen, für die wir garantieren können, dass folgende Bedingung erfüllt ist:

$$2^{\frac{\omega-1}{4}} \cdot \det(L)^{\frac{1}{\omega}} < 2^{-\omega} \cdot (XY)^{m-1} \cdot W$$

$$\Leftrightarrow 2^{\frac{(\delta+m)^2-1}{4}} \cdot \det(L)^{\frac{1}{(\delta+m)^2}} < 2^{-(\delta+m)^2} \cdot (XY)^{m-1} \cdot W$$

Wir setzen die Determinante $\det(L) = (XY)^{\frac{(\delta+m-1)(\delta+m)^2+(m-1)m^2}{2}} \cdot n^{\delta(\delta+2m)}$ ein und erhalten:

$$2^{\frac{(\delta+m)^2-1}{4}} \cdot (XY)^{\frac{(\delta+m-1)\cdot(\delta+m)^2+(m-1)\cdot m^2}{2\cdot(\delta+m)^2}} \cdot n^{\frac{\delta\cdot(\delta+2m)}{(\delta+m)^2}} < 2^{-(\delta+m)^2} \cdot (XY)^{m-1} \cdot W$$

Da $n < 2 \cdot (XY)^k \cdot W$, ist diese Bedingung erfüllt, wenn folgende, strengere Bedingung gilt:

$$2^{\frac{(\delta+m)^2-1}{4}} \cdot (XY)^{\frac{(\delta+m-1)\cdot(\delta+m)^2+(m-1)\cdot m^2}{2\cdot(\delta+m)^2}} \cdot (2 \cdot (XY)^k \cdot W)^{\frac{\delta\cdot(\delta+2m)}{(\delta+m)^2}} < 2^{-(\delta+m)^2} \cdot (XY)^{m-1} \cdot W$$

$$\Leftrightarrow XY < 2^{-\left(\frac{5(\delta+m)^4-(\delta+m)^2+4\delta\cdot(\delta+2m)}{2(\delta+m-1)\cdot(\delta+m)^2-2(m-1)\cdot m^2}\right)} \cdot W^{\frac{2m^2}{(\delta+m-1)\cdot(\delta+m)^2-(m-1)\cdot m^2}}$$

Falls also diese Schranke eingehalten wird und $|x_0| < X$ sowie $|y_0| < Y$ gilt, können wir also mit dem beschriebenen Algorithmums die Nullstelle (x_0,y_0) berechnen. Wir benutzen nun zwei Ungleichungen zur Vereinfachung. Betrachten wir dazu die Exponenten. Für den ersten Exponenten haben wir:

$$\frac{5(\delta+m)^4-(\delta+m)^2+4\delta\cdot(\delta+2m)}{2(\delta+m-1)\cdot(\delta+m)^2-2(m-1)\cdot m^2}$$

$$= \frac{5\delta^4+20\delta^3m+24\delta^2m^2+20\delta m^3+5m^4-\delta^2-2\delta m-m^2+4\delta^2+8\delta m}{2\delta^3+6\delta^2m+6\delta m^2-2\delta^2-4\delta m}$$

$$= \frac{4m^2}{\delta} \cdot \left(1+\frac{4\delta^2-\delta m-\frac{19}{4}m^2+2\delta+4m+\frac{5}{4}\frac{\delta^4}{m^2}+5\frac{\delta^3}{m}-\frac{1}{4}\frac{\delta^2}{m^2}-\frac{1}{2}\delta m-\frac{1}{4}+\frac{\delta^2}{m^2}+2\frac{\delta}{m}}{2\delta^2+6\delta m+6m^2-2\delta-4m}\right)$$

$$= \frac{4m^2}{\delta}+\frac{16\delta^2m^2-6\delta m^3-19m^4+8\delta m^2+16m^3+5\delta^4+20\delta^3m-\delta^2-m^2+4\delta^2+8\delta m}{2\delta^3+6\delta^2m+6\delta m^2-2\delta^2-4\delta m}$$

Es gilt:

$$\lim_{m\to\infty}\left(\frac{16\delta^2m^2-6\delta m^3-19m^4+8\delta m^2+16m^3+5\delta^4+20\delta^3m-\delta^2-m^2+4\delta^2+8\delta m}{2\delta^3+6\delta^2m+6\delta m^2-2\delta^2-4\delta m}\right)$$

$$\leq -\frac{3m^2}{\delta}+3\delta$$

Damit erhalten wir insgesamt für hinreichend großes m:

$$\frac{5(\delta+m)^4-(\delta+m)^2+4\delta\cdot(\delta+2m)}{2(\delta+m-1)\cdot(\delta+m)^2-2(m-1)\cdot m^2}\leq\frac{m^2}{\delta}+3\delta$$

Für den zweiten Exponenten erhalten wir:

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

$$=\frac{2m^2}{3\delta m^2}\cdot\frac{1}{\frac{\delta^2}{3m^2}+\frac{\delta}{m}+1-\frac{\delta}{3m^2}-\frac{2}{3m}}$$

$$=\frac{2}{3\delta}\cdot\left(\frac{1+\frac{\delta^2}{3m^2}+\frac{\delta}{m}-\frac{\delta}{3m^2}-\frac{2}{3m}}{1+\frac{\delta^2}{3m^2}+\frac{\delta}{m}-\frac{\delta}{3m^2}-\frac{2}{3m}}-\frac{\frac{\delta^2}{3m^2}+\frac{\delta}{m}-\frac{\delta}{3m^2}-\frac{2}{3m}}{1+\frac{\delta^2}{3m^2}+\frac{\delta}{m}-\frac{\delta}{3m^2}-\frac{2}{3m}}\right)$$

$$=\frac{2}{3\delta}\cdot\left(1-\frac{\delta^2+3\delta m-\delta-2m}{3m^2+\delta^2+3\delta m-\delta-2m}\right)=\frac{2}{3\delta}-\frac{2}{3m}\cdot\frac{\delta^2+3\delta m-\delta-2m}{3\delta m+\frac{\delta^3}{m}+3\delta^2-\frac{\delta^2}{m}-2\delta}$$

Es ist für $m\to\infty$:

$$\delta^2m+\delta^2+\delta m<\delta^3+3\delta^2m+2m^2$$

$$\Leftrightarrow \delta^2+3\delta m-\delta-2m<3\delta m+\frac{\delta^3}{m}+3\delta^2-\frac{\delta^2}{m}-2\delta$$

Damit ist $\dfrac{\delta^2+3\delta m-\delta-2m}{3\delta m+\frac{\delta^3}{m}+3\delta^2-\frac{\delta^2}{m}-2\delta}<1$, und wir erhalten:

$$\frac{2}{3\delta} - \frac{2}{3m} \cdot \frac{\delta^2 + 3\delta m - \delta - 2m}{3\delta m + \dfrac{\delta^3}{m} + 3\delta^2 - \dfrac{\delta^2}{m} - 2\delta} > \frac{2}{3\delta} - \frac{2}{3m}$$

Also haben wir:

$$\frac{2m^2}{(\delta + m - 1) \cdot (\delta + m)^2 - (m-1) \cdot m^2} > \frac{2}{3\delta} - \frac{2}{3m}$$

Damit ergibt sich insgesamt für hinreichend großes m:

$$XY < 2^{-\frac{m^2}{\delta} - 3\delta} \cdot W^{\frac{2}{3\delta} - \frac{2}{3m}}$$

Setzen wir zusätzlich $m := \left\lceil \frac{1}{\varepsilon} \right\rceil$ (und vernachlässigen den Faktor $\frac{2}{3}$ zu ε im Exponenten zur Basis W), erhalten wir:

$$XY < 2^{-\frac{1}{\delta \cdot \varepsilon^2} - 3\delta} \cdot W^{\frac{2}{3\delta} - \varepsilon}$$

Diese vereinfachte Schranke zeigt das asymptotische Verhalten gut. Mit dieser Schranke ist die Laufzeit von Corons Algorithmus polynomiell in $(\log(W), \delta, \frac{1}{\varepsilon})$. Zur praktischen Abschätzung der erlaubten Bitlänge des unbekannten Teils von p mit gegebenem m eignet sich die nicht vereinfachte Schranke allerdings besser.

4.2 Implementierung von Coron im bivariaten Fall

Der in Abschnitt 4.1 beschriebene Algorithmus ist in dem zu dieser Arbeit gehörigen Softwarepaket unter dem Namen CoronBivariateRoots enthalten. Die Signatur der entsprechenden *C/C++*-Funktion CoronBivariateRoots, hier abgekürzt CBR, sieht folgendermaßen aus:

```
long CBR(ZZ* coefficients, ZZ* root, ZZ X, ZZ Y, long deltaX, long deltaY, long m, int out)
```

Sei $p(x, y)$ das Polynom, dessen Nullstellen berechnet werden sollen. Sei δ_x der Grad von $p(x, y)$ in x und sei δ_y der Grad von $p(x, y)$ in y. Die Signatur setzt sich zusammen aus:

- Der erste Parameter coeff ist ein Pointer auf ein Array der Länge $(\delta_x + 1) \cdot (\delta_y + 1)$, mit dem Koeffizienten zu $x^i y^j$ von $p(x)$ an Position $i \cdot \delta_x + j$.
- Der zweite Parameter roots erwartet ebenfalls einen Pointer auf ein Array der Länge $\delta_x \cdot \delta_y$, hier werden im Verlauf des Programms die gefundenen

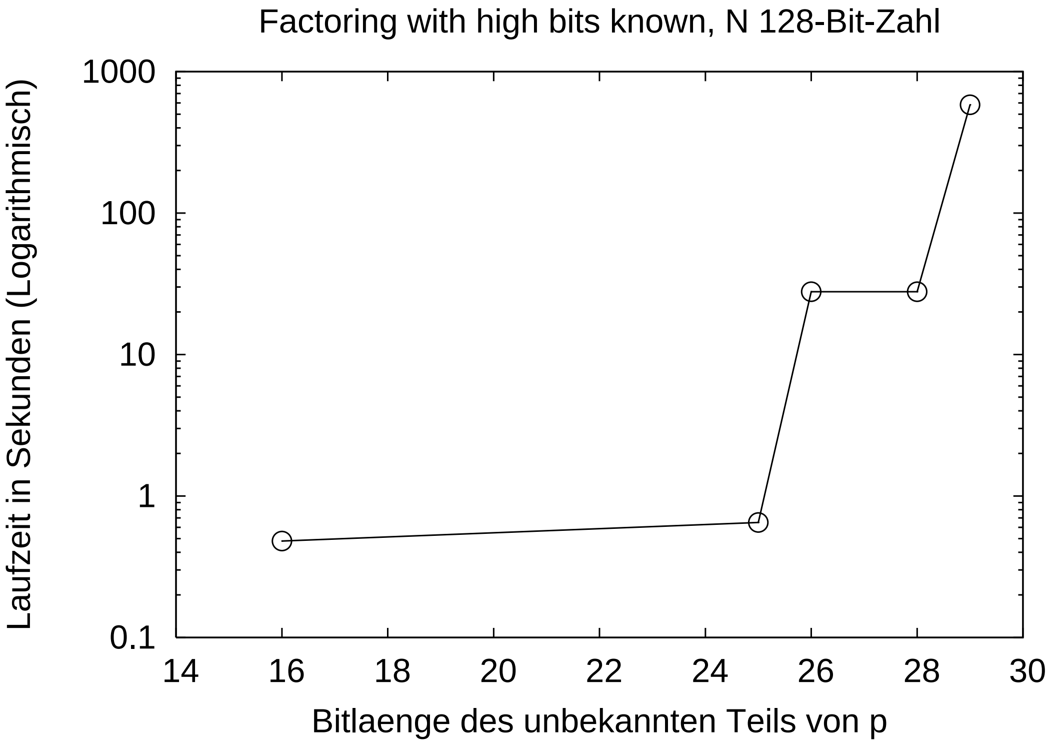

Abbildung 4.1: Laufzeitkurve zur Faktorisierung mit Coron (N 128-Bit-Zahl)

Nullstellen abgespeichert und können nach beendeter Berechung ausgelesen werden.

- Der dritte Parameter X entspricht der oberen Schranke für die erste Komponente x_0 der gesuchten Nullstelle (x_0, y_0).
- Der vierte Parameter Y entspricht der oberen Schranke für die zweite Komponente y_0 der gesuchten Nullstelle (x_0, y_0).
- Der fünfte Parameter `deltaX` ist für die Übergabe von δ_x vorgesehen.
- Der sechste Parameter `deltaY` ist für die Übergabe von δ_y bestimmt.
- Der siebte Parameter m entspricht der Beschreibung in Abschnitt 4.1.
- Der achte und letzte Parameter `out` bestimmt, ob der Nutzer Ausgaben der im Verlaufe des Algorithmus benötigten Matrizen, Vektoren und Polynome sehen möchte.

Die Implementierung hält sich im Wesentlichen genau an die Vorgaben aus Abschnitt 4.1. Wie in Coron[4] vorgeschlagen, verwenden wir allerdings auch hier

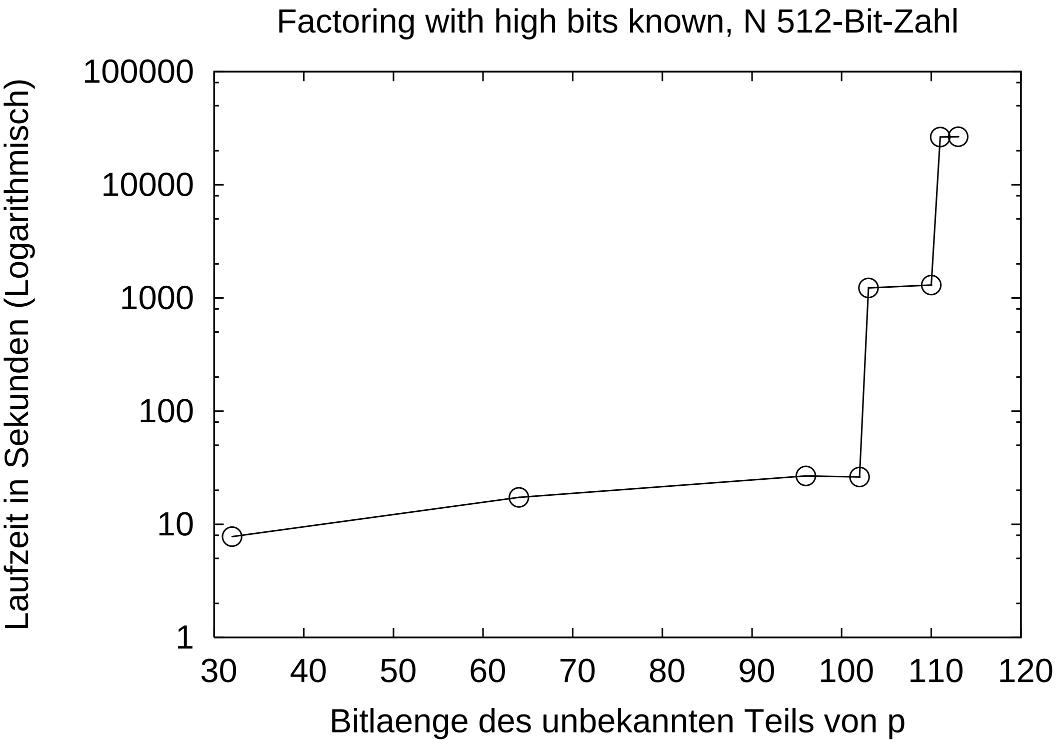

Abbildung 4.2: Laufzeitkurve zur Faktorisierung mit Coron (N 512-Bit-Zahl)

eine Skalierung der Gittermatrix. Wir haben in den vorangegangenen Kapiteln eine Skalierung verwendet, um dafür zu sorgen, dass eine gegebene Gittermatrix nur noch ganzzahlige Einträge enthält. Wird ein Gitter wie in Abschnitt 4.1 beschrieben aufgebaut, enthält es aber auch ohne Skalierung bereits nur ganzzahlige Einträge. Zu diesem Zweck ist es also nicht nötig, das Gitter noch zu verändern. In Coron[4] wird allerdings noch eine Skalierung vorgeschlagen, um die Zeit zu verkürzen, die zur Berechnung einer reduzierten Gitterbasis benötigt wird.
Wir multiplizieren das Gitter vor der Verwendung des L^3-Unterprogrammes noch mit der Zahl $(XY)^{-(m-1)}$. Dadurch werden die Einträge der Gittermatrix kleiner und die reduzierte Gitterbasis kann schneller berechnet werden, da die Laufzeit dieser Berechnung, wie in Abschnitt 1.3 erwähnt, auch von der Länge des längsten Vektors in der Gitterbasis abhängt. Nach Berechnung der reduzierten Gitterbasis wird das resultierende Gitter mit der Zahl $(XY)^{m-1}$ multipliziert, um die Skalierung wieder rückgängig zu machen und das Programm fährt wie in Abschnitt 4.1 beschrieben fort.

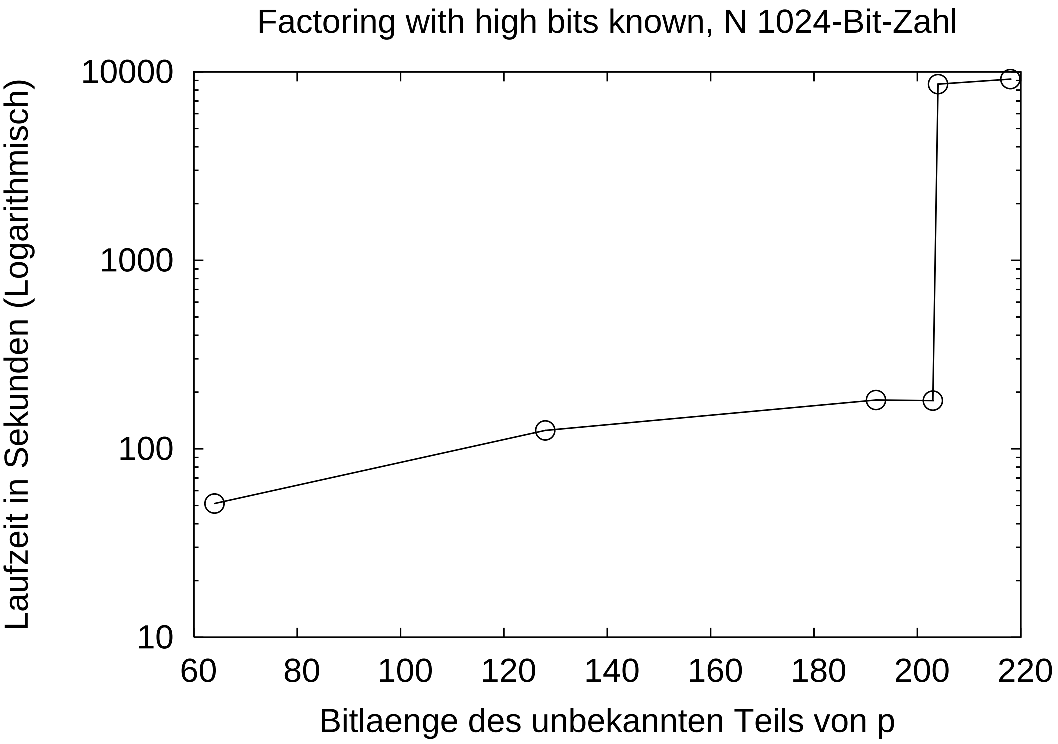

Abbildung 4.3: Laufzeitkurve zur Faktorisierung mit Coron (N 1024-Bit-Zahl)

4.3 Testergebnisse zu Coron im bivariaten Fall

Ähnlich wie in Abschnitt 3.3 können wir den Algorithmus aus Abschnitt 4.1 verwenden, um die Faktorisierung eines RSA-Modulus zu bestimmen. Wieder sei $N \in \mathbb{Z}$ mit:

$$N = p \cdot q \text{ und } p,\, q \text{ prim}$$

Dabei seien $p, q, \tilde{p}, \tilde{q}$ wie in Abschnitt 3.3 und wir definieren wieder das Polynom:

$$p(x, y) = (\tilde{p} + x) \cdot (\tilde{q} + y) - N = xy + \tilde{p}y + \tilde{q}x + \tilde{p}\tilde{q} - N$$

Können wir eine Nullstelle (x_0, y_0), $x_0, y_0 \in \mathbb{Z}$, bestimmen, so hätten wir auch wieder die Primfaktorisierung von N berechnet, da gilt:

$$(\tilde{p} + x_0) \cdot (\tilde{q} + y_0) - N = 0$$

$$\Leftrightarrow (\tilde{p} + x_0) \cdot (\tilde{q} + y_0) = N$$

$$\Leftrightarrow (\tilde{p} + x_0) \cdot (\tilde{q} + y_0) = p \cdot q$$

Mit $x_0, y_0, \tilde{p}, \tilde{q} \in \mathbb{Z}$ haben wir $\tilde{p} + x_0 = p$ und $\tilde{q} + y_0 = q$ und kennen also die Primfaktorisierung von N. Mit der Schranke aus Abschnitt 4.1 wissen wir, dass wir die Nullstelle (x_0, y_0) berechnen können, falls:

$$x_0 y_0 \leq XY < XY < 2^{-\frac{1}{\delta \cdot \varepsilon^2} - 3\delta} \cdot W^{\frac{2}{3\delta} - \varepsilon}$$

gilt. Wir haben dabei Tests für N als 128-Bit-Zahl (p und q jeweils 64-Bit-Zahlen), N als 512-Bit-Zahl (p und q jeweils 256-Bit-Zahlen) sowie N als 1024-Bit-Zahl (p und q jeweils 512-Bit-Zahlen) durchgeführt. Wieder haben wir „Pentium III“-Prozessoren mit einer Taktung von 1000 Mhz und einem Arbeitsspeicher von 512 Megabyte RAM benutzt. Die entsprechenden Laufzeitkurven sind auf Abbildung

Anzahl Bits, x_0:	m:	Gitterdimension:	Laufzeit, L^3:	Laufzeit, gesamt:
64	2	9	7.74	7.79
103	3	16	1226.51	1227.22
111	4	25	26420.31	26423.11

Abbildung 4.4: Laufzeittabelle (in Sekunden) zu N als 512-Bit-Zahl

4.1, Abbildung 4.2 und Abbildung 4.3 dargestellt. Die Skalierung der Laufzeitachse wurde wieder logarithmisch gewählt. Im Vergleich zu dem bivariaten Ansatz von Coppersmith benötigt Corons Algorithmus deutlich mehr Laufzeit für das äquivalente Problem. Der Parameter m muss schon bei in der Bitlänge kürzerem, unbekannten Teil des Faktors p relativ groß gewählt werden. Auch ist die Laufzeit bei gleichem m für Coron höher, weil eine reduzierte Gitterbasis zu dem gesamten Gitter, nicht nur zu einem Teilgitter, berechnet werden muss. Die Laufzeitkurven sehen sonst ähnlich aus wie in den Abschnitten 2.3 und 3.3. Auffällig ist, das bei Coron die Laufzeit, anders als bei den bisher vorgestellten Algorithmen, mit wachsender Bitlänge ansteigt, auch wenn der Parameter m unverändert bleibt. Der Grund dafür ist, dass die Koeffizienten der Polynome $\tilde{q}(x, y)_{ij}$ durch den Modulus n von der Größe von X und Y abhängen, und X sowie Y werden in den Testreihen immer so groß wie x_0 bzw. y_0 gewählt. Einträge der Gittermatrix und die Größenordnung der Laufzeiten sind in Tabelle 4.4 zu sehen. Außerdem wird auch bei dieser Anwendung deutlich, wie sehr der L^3-Algorithmus die Gesamtlaufzeit dominiert. Wieder ist der Parameter m bei allen Testläufen optimal, das heißt so klein wie möglich, gewählt worden. Wie schon in Abschnitt 4.2 erwähnt, wird jedes von CBR erzeugte Gitter mit dem Faktor $(XY)^{-(m-1)}$ skaliert. Nach dieser Skalierung steht in der Gittermatrix immer mindestens eine 1, so dass eine einfache Skalierung des Gesamtgitters nicht weiter möglich scheint, um zur Beschleunigung des Verfahrens eine Verkürzung der Gitterbasisvektoren zu errei-

m:	Maximale Bitlänge, Test:	Maximale Bitlänge, Vorhersage:
2	102	92
3	110	100
4	113	108

Abbildung 4.5: Vergleich Vorhersage mit Implementierung (N 512-Bit-Zahl)

chen.
Weiter haben wir die errechneten Schranken zu Corons Algorithmus getestet. Es hat sich in unseren Tests gezeigt, dass die Schranke:

$$XY < 2^{-\left(\frac{5(\delta+m)^4-(\delta+m)^2+4\delta\cdot(\delta+2m)}{2(\delta+m-1)\cdot(\delta+m)^2-2(m-1)\cdot m^2}\right)} \cdot W^{\frac{2m^2}{(\delta+m-1)\cdot(\delta+m)^2-(m-1)\cdot m^2}}$$

verwendet werden sollte, um brauchbare Vorhersagen über die maximale Bitlänge des unbekannten Teils von p zu bekommen, der mit gegebenem m noch berechnet werden kann. Die abgekürzte Version liefert insbesondere für kleine m keine brauchbaren Ergebnisse. Für unsere Anwendung haben wir $\delta = 1$. Wir können nun jeweils den Ausdruck:

$$4 \cdot 2^{-\left(\frac{5(m+1)^4-(m+1)^2+4\cdot(2m+1)}{2m\cdot(m+1)^2-2(m-1)\cdot m^2}\right)} \cdot W^{\frac{2m^2}{m\cdot(m+1)^2-(m-1)\cdot m^2}}$$

$$\approx \log_2(10) \cdot 2^{-\left(\frac{5(m+1)^4-(m+1)^2+4\cdot(2m+1)}{2m\cdot(m+1)^2-2(m-1)\cdot m^2}\right)} \cdot W^{\frac{2m^2}{m\cdot(m+1)^2-(m-1)\cdot m^2}}$$

zur Vorhersage für ein gegebenes m berechnen. Die Tabelle 4.5 zeigt die maximale Bitlänge des unbekannten Teils von p für ein bestimmtes m im Vergleich zu dem von der Schranke vorhergesagten Wert.

Kapitel 5

Faktorisierung mit Kenntnis des Entschlüsselungsexponenten d

5.1 Algorithmus zur Faktorisierung mit d

Der in diesem Kapitel beschriebene Algorithmus stammt aus Blömer, May[7]. Er ermöglicht es, aus den im RSA-Verfahren verwendeten Verschlüsselungsexponenten e und Entschlüsselungsexponenten d die Faktorisierung vom zugehörigen Modulus N zu berechnen. Der Entschlüsselungsexponent d wird also für diese Anwendung als bekannt vorausgesetzt. In einer praktischen Anwendung wäre dieser Algorithmus also nicht interessant, denn wenn dem Angreifer der Entschlüsselungsexponent d bereits bekannt ist, kann er alle Nachrichten einfach dechiffrieren. Dieser Algorithmus zielt also nicht darauf ab, die Sicherheit von RSA zu brechen, sondern dient zum Beweis einer Aussage über die Sicherheit von RSA. Wir können nämlich mit Hilfe dieses Kapitels die Aussage beweisen, dass, wenn der Exponent d bekannt ist, auch die Faktorisierung von N in Polynomialzeit berechnet werden kann. Damit wissen wir, dass die Berechnung von d mindestens so schwer ist wie die Faktorisierung von N. Das ist eine wichtige Erkenntnis, denn daraus folgt, dass es für einen Angreifer ein schwieriges Problem darstellt, den Entschlüsselungsexponenten d effizient zu bestimmen. Denn hätte ein Angreifer einen solchen Algorithmus zur Verfügung, könnte er, wie wir jetzt zeigen werden, auch effizient Zahlen faktorisieren, die das Produkt zweier Primzahlen sind. Ein Algorithmus, der das leistet, ist aber momentan unseres Wissens nach nicht bekannt. Dieses Kapitel beschäftigt sich also im komplexitätstheoretischen Sinne mit einer Reduktion.
Seien e und d definiert wie im RSA-Verfahren (siehe dazu Abschnitt 1.1). Wegen $1 < e, d < \phi(N) < N$ gilt:

$$ed \leq N^2$$

Wir nehmen an, dass die Darstellung von p und q die gleiche Anzahl von Bits benötigt. Folglich haben wir:

$$p < N^{\frac{1}{2}} < q < 2p < 2N^{\frac{1}{2}}$$

Wir erhalten direkt die Abschätzung:

$$p + q < 3N^{\frac{1}{2}}$$

Außerdem gilt:

$$\phi(N) = N + 1 - (p + q) > N + 1 - 3N^{\frac{1}{2}} > N - 3N^{\frac{1}{2}}$$

Für $N > 36$ erhalten wir $N - 3N^{\frac{1}{2}} > \frac{1}{2}N$ und eine weitere Abschätzung:

$$\phi(N) > \frac{1}{2}N$$

Wir machen jetzt folgende Fallunterscheidung:

Fall 1: $ed \leq N^{\frac{3}{2}}$

Mit:

$$ed = 1 \ mod \ \phi(N)$$

erhalten wir:

$$ed = 1 + k\phi(N)$$

$$\Leftrightarrow k = \frac{ed - 1}{\phi(N)}$$

für ein unbekanntes $k \in \mathbb{N}$. Wir können allerdings eine Näherung für k berechnen. Wir definieren:

$$\tilde{k} := \frac{ed - 1}{N}$$

Also ist:

$$k - \tilde{k} = \frac{ed - 1}{\phi(N)} - \frac{ed - 1}{N} = \frac{N(ed - 1) - \phi(N)(ed - 1)}{\phi(N)N}$$

$$= \frac{N(ed-1) - (N-p-q+1)(ed-1)}{\phi(N)N} = \frac{(p+q-1)(ed-1)}{\phi(N)N}$$

Benutzen wir die Ungleichungen $p + q - 1 < 3N^{\frac{1}{2}}$ und $\phi(N) > \frac{1}{2}N$, ergibt sich weiter:

$$k - \tilde{k} = \frac{(p+q-1)(ed-1)}{\phi(N)N} < \frac{3N^{\frac{1}{2}} \cdot (ed-1)}{\frac{1}{2}N \cdot N} = 6N^{-\frac{3}{2}}(ed-1)$$

Insgesamt haben wir:

$$k - \tilde{k} < 6N^{-\frac{3}{2}}(ed-1)$$

Benutzen wir an dieser Stelle die Voraussetzung $ed \leq N^{\frac{3}{2}}$, erhalten wir schließlich $k - \tilde{k} < 6$. Mit diesem Resultat wissen wir sofort, dass $k = \lceil \tilde{k} \rceil + i$ für ein $i \in \{0, 1, \ldots, 5\}$. Da $\phi(N)$ ganzzahlig ist, kann man die Frage, ob k korrekt gewählt ist, eventuell schon negativ beantworten, wenn man $\frac{ed-1}{k}$ berechnet hat und das Ergebnis nicht ganzzahlig ist. Für die in Frage kommenden Kandidaten k können wir die Zahl $N+1+\frac{1-ed}{k}$ berechnen und erhalten ein Gleichungssystem aus zwei Gleichungen in zwei Unbekannten p und q, welches dann ohne großen Aufwand gelöst werden kann. Die Definition des RSA-Verfahrens liefert uns die erste Gleichung:

$$N = p \cdot q$$

Die zweite Gleichung ist:

$$N + 1 + \tfrac{1-ed}{k} = p + q$$

Entweder die Resultate für p und q sind Faktoren von N, was durch einfache Division leicht zu prüfen ist, oder die Wahl von i war nicht korrekt und wir müssen einen anderen Wert für i ausprobieren. Also können wir die Faktoren p und q von N effizient berechnen, falls $ed \leq N^{\frac{3}{2}}$ gilt. □

Fall 2: $N^{\frac{3}{2}} < ed \leq N^2$

Auch hier beginnen wir mit der Definition $\tilde{k} := \frac{ed-1}{N}$, und können analog zum ersten Fall berechnen, dass:

$$k - \tilde{k} < 6N^{-\frac{3}{2}}(ed-1)$$

gilt. Dann benutzen wir $ed \leq N^2$ und erhalten eine Fehlerabschätzung:

$$k - \tilde{k} < 6N^{-\frac{3}{2}}(ed-1) < 6N^{-\frac{3}{2}}N^2 = 6N^{\frac{1}{2}}$$

Als nächstes benötigen wir eine Approximation von $\phi(N)$. Es ist:

$$N - \phi(N) = p + q - 1 < 3N^{\frac{1}{2}}$$

Also muss $\phi(N)$ in dem Intervall $[N-3N^{\frac{1}{2}}, N]$ liegen. Wir unterteilen dieses Intervall nun in sechs gleich große Teilintervalle, jeweils von der Länge $\frac{1}{2}N^{\frac{1}{2}}$. Die Mittelpunkte dieser Teilintervalle sind gegeben durch $N - \frac{2i-1}{4}N^{\frac{1}{2}}$. Sei $g := N - \frac{2i'-1}{4}N^{\frac{1}{2}}$ mit i' als die richtige Wahl für i. Es gilt:

$$|N - g - \phi(N)| \leq |\tfrac{1}{4}N^{\frac{1}{2}}|$$

Anders ausgedrückt $\phi(N) = N - g - y$ für ein y mit $y \leq \frac{1}{4}N^{\frac{1}{2}}$. Die korrekte Wahl von g lässt sich nicht direkt prüfen, da $\phi(N)$ unbekannt ist und so die obige Bedingung nicht einfach nachgerechnet werden kann. Aber auch hier können wir die verschiedenen Werte für i bis zur erfolgreichen Berechnung von p und q, welche wir im Folgenden beschreiben, nacheinander ausprobieren.
Wir setzen jetzt $x := k - \tilde{k}$. Setzen wir diese Resultate in die Formel $ed = 1 + k\phi(N)$ ein, erhalten wir:

$$ed - 1 - (\tilde{k} + x)(N - g - y) = xy - (N-g)x + \tilde{k}y - \tilde{k}(N-g) + ed - 1 = 0$$

Wir betrachten also das bivariate Polynom:

$$p(x,y) = xy - (N-g)x + \tilde{k}y - \tilde{k}(N-g) + ed - 1$$

mit der Nullstelle $(x_0, y_0) = (k - \tilde{k}, p + q - 1 - g)$. $\tilde{k}$ und g können dabei ohne weiteres aufgerundet werden. Dadurch werden alle Koeffizienten des Polynoms $p(x,y)$ ganzzahlig. Mit Setzung von $X := 6N^{\frac{1}{2}}$ und $Y := \frac{1}{4}N^{\frac{1}{2}}$ haben wir also $|x_0| < X$ und $|y_0| < Y$, und es gilt für die korrekte Wahl von g:

$$W \geq (N-g) \cdot X \geq (N - 3N^{\frac{1}{2}}) \cdot 6N^{\frac{1}{2}} = 6N^{\frac{3}{2}} - 18N \geq 3N^{\frac{3}{2}}$$

Damit können wir zeigen, dass die Schranke $XY < W^{\frac{2}{3}}$ für den bivariaten Ansatz aus Coppersmith[2] eingehalten wird:

$$XY = 6N^{\frac{1}{2}} \cdot \tfrac{1}{4}N^{\frac{1}{2}} = \tfrac{3}{2}N < (3N^{\frac{3}{2}})^{\frac{2}{3}} \leq W^{\frac{2}{3}}$$

Wir können auch die in den Abschnitt 3.1 und Abschnitt 4.1 vorgestellten Algorithmen verwenden, um die Nullstellen von $p(x,y)$ zu berechnen. Verwenden wir beispielsweise den Algorithmus aus Abschnitt 4.1, haben wir die Schranke:

$$XY < 2^{-\frac{1}{\delta \cdot \varepsilon^2} - 3\delta} \cdot W^{\frac{2}{3\delta} - \varepsilon}$$

Diese Schranke ist allerdings nicht so gut wie die aus Coppersmith[2] und würde an dieser Stelle nicht für den Reduktionsbeweis genügen. Sind die Nullstellen von $p(x, y)$ berechnet, erhalten wir die Zahl $x_0 = k - \tilde{k}$, addieren $\tilde{k}$, und können damit k bestimmen. Weiter kennen wir $y_0 = p + q - 1 - g$, addieren $1 + g$ und erhalten so die Zahl $S := p + q$. Auch hier haben wir also wieder ein Gleichungssystem aus zwei Gleichungen in den zwei Variablen p und q. Die erste Gleichung ist wie in Fall 1 gegeben durch:

$$N = p \cdot q$$

Die zweite Gleichung ist:

$$S = p + q$$

Ist eine der Lösungen des Systems ein Faktor von p, ist das Ziel erreicht und N kann faktorisiert werden. Ob p ein Faktor von N ist, kann auch hier durch einfache Division überprüft werden. Ist die Wahl von g und damit i nicht korrekt, müssen wir ein anderes i probieren, bis die Wahl zum gewünschten Ergebnis führt.

5.2 Implementierung der Faktorisierung mit d

Der gerade in Abschnitt 5.1 beschriebene Algorithmus ist ebenfalls für das in dieser Arbeit entstandene Softwarepaket implementiert worden. Die Signatur der entsprechenden *C/C++*-Funktion ExponentToFactorization, abgekürzt ETF, ist:

```
ZZ ETF(ZZ N, ZZ X, ZZ Y, ZZ e, ZZ d, long m, int out)
```

Die Signatur setzt sich zusammen aus:

- Der erste Parameter N ist für den RSA-Modulus bestimmt, dessen Faktoren berechnet werden sollen.
- Der zweite Parameter X ist die obere Schranke für die betragsmäßige Größe der Komponente x_0 der Nullstelle.
- Der dritte Parameter Y ist die obere Schranke für die betragsmäßige Größe der Komponente y_0 der Nullstelle.
- Der vierte Parameter e muss den Verschlüsselungsexponenten des entsprechenden RSA-Systems enthalten.

- Der fünfte Parameter `d` ist für den entsprechenden Entschlüsselungsexponenten vorgesehen.
- Der sechste Parameter `m` dient zur Kontrolle der Laufzeit des verwendeten Algorithmus' zur Nullstellenberechnung.
- Der siebte und letzte Parameter `out` dient, wie auch bei den anderen Funktionen, zur Kontrolle der Ausgabe des Programmes.

Der Rückgabewert der Funktion ist dann bei korrekter Berechnung ein Faktor von N. Sollte der Benutzer auch den anderen Faktor von N benötigen, kann er diesen durch Division von N durch den berechneten Rückgabewert bestimmen.
Die Approximation von $\phi(N)$ durch $\frac{ed-1}{N}$ haben wir in der Implementierung durch eine bessere Approximation ersetzt. Wir wissen, dass $\phi(N) = N - 1 - p - q$ ist. Da p und q nach Voraussetzung von der gleichen Bitlänge sind, liefert uns die Approximation $\phi(N) \approx N - 1 - 2 \cdot \sqrt{N}$ eine recht gute Näherung für die praktische Anwendung, die es eventuell ermöglicht, die Faktorisierung von N auch mit relativ kleinem Parameter m zu berechnen.
Da die Parameter X und Y bei dieser Anwendung relativ groß gewählt werden müssen, muss auch der Parameter m zur Berechnung der Faktorisierung groß gewählt werden. Dadurch benötigt das L^3-Unterprogramm unter realistischen Bedingungen (N 512-Bit-Zahl oder 1024-Bit-Zahl) viel Rechenzeit, um die reduzierte Gitterbasis zu berechnen. Aus diesem Grund haben wir nur Korrektheitstests für kleinere N durchgeführt, die erfolgreich verlaufen sind, allerdings sonst wenig Aussagekraft besitzen und die wir deshalb an dieser Stelle nicht vorstellen werden.

Kapitel 6

Zusammenfassung der Ergebnisse dieser Arbeit

6.1 Vergleich Coppersmith-Coron im bivariaten Fall

Wir haben zwei Algorithmen zur Faktorisierung eines RSA-Modulus getestet, den bivariaten Ansatz von Coppersmith und den von Coron. Corons Variante benötigt dabei deutlich mehr Rechenzeit als die von Coppersmith bei gleich großem Parameter m, siehe Abbildung 6.1 und Abbildung 6.2. Der Grund dafür ist, dass Corons Algorithmus die Berechnung einer reduzierten Gitterbasis zu dem gesamten konstruierten Gitter erfordert, in Coppersmiths Variante dagegen muss nur die reduzierte Gitterbasis zu einem Teilgitter berechnet werden. Da die Gesamtlaufzeiten alle praktisch vollständig von der Laufzeit des L^3-Unterprogrammes abhängen, wirkt sich dieser Unterschied dermaßen stark aus. Die Laufzeit von Coron steigt damit auch schneller in m als die von Coppersmith, da reduzierte Gitterbasen zu schneller wachsenden Gittergrößen berechnet werden müssen.
Andererseits ist Corons Ansatz deutlich einfacher zu implementieren, da die Gittermatrix nach Konstruktion nicht weiter umgeformt werden muss, bevor die Berechnung der reduzierten Gitterbasis beginnt. Außerdem kann Corons Algorithmus mit einem kleineren Parameter m mehr unbekannte Bits berechnen.
Nähert man sich mit der Größe des unbekannten Teils von p der jeweiligen Schranke, kennt man also nur vergleichsweise wenige Bits von p, legt die Theorie eine Nutzung von Coppersmiths Ansatz nah, da es uns möglich war, eine bessere Schranke für diesen Algorithmus herzuleiten. Beobachtungen des Laufzeitverhaltens der Algorithmen im Vergleich, siehe Abbildung 6.1 und Abbildung 6.2, stützen diese Vermutung, denn die Zunahme der Laufzeit bei Erhöhung von m ist bei Corons Algorithmus deutlich drastischer, und für Bitlängen nahe an der theoretischen Schranke muss m groß gewählt werden. Für Anwendungen in diesem Bereich sollten jedoch sehr schnelle Rechner benutzt werden, und eine Paralle-

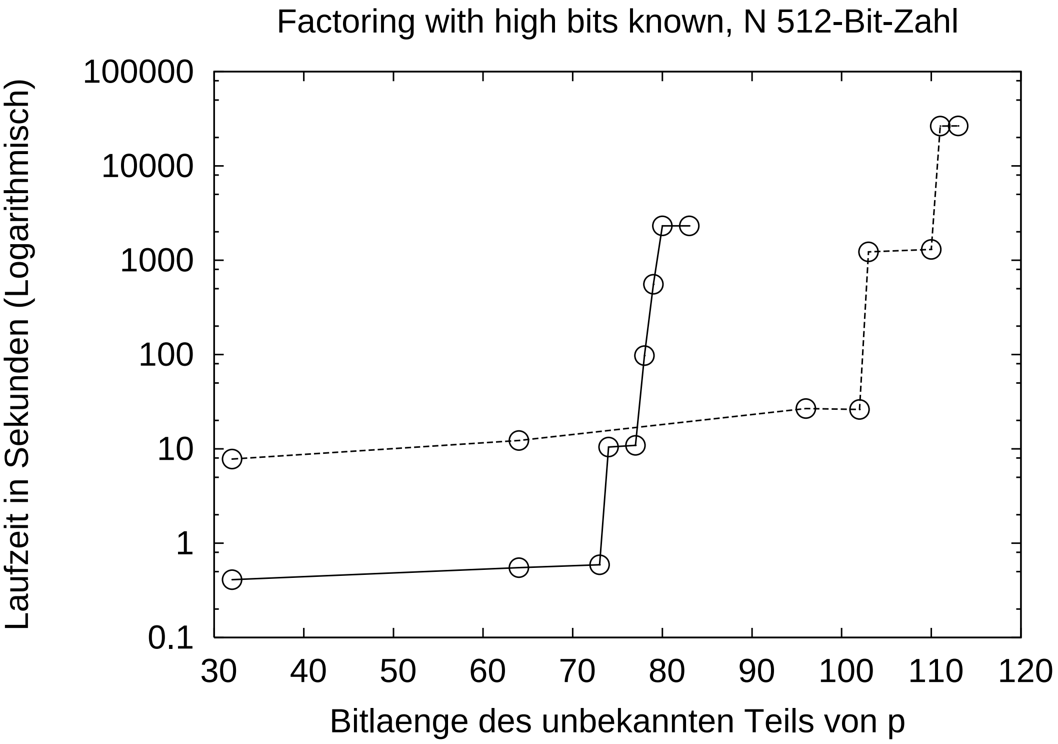

Abbildung 6.1: Laufzeitvergleich von Coppersmiths (durchgehende Linie) und Corons (gestrichelte Linie) Faktorisierungsalgorithmen (N 512-Bit-Zahl)

lisierung des Verfahrens wäre sicherlich hilfreich. Da die Laufzeit der Faktorisierungsalgorithmen praktisch vollständig von der Laufzeit des L^3-Programmes abhängt, ist ein Master-Slave-Prinzip der naheliegendste Ansatz, in dem nur die Berechnung der reduzierten Gitterbasis wirklich parallel stattfindet und die übrigen Operationen (Gittertransformation, Resultantenberechnung, Sturm-Sequenz) vollständig vom Master ausgeführt werden. Erstens könnte dann auf bestehenden Code zur parallelen Berechnung einer reduzierten Gitterbasis zurückgegriffen werden, und zweitens müsste der im Rahmen dieser Arbeit produzierte Code kaum verändert werden.

Wir wollen an dieser Stelle noch darauf hinweisen, dass die sowohl bei Coppersmiths als auch bei Corons Algorithmus gemachte Voraussetzung, das zur Verwendung der Faktorisierungsalgorithmen einige höherwertige Bits von p bekannt sein müssen, eigentlich nichts anderes bedeutet, als dass wir eine „gute Approximation“ von p benötigen. In diesem wie im üblichen Sinne würde aber die Binärzahl $A = 0111$ besser durch $B = 1000$ approximiert als durch $C = 0100$, obwohl mehr höherwertige Bits von C mit A übereinstimmen als von B mit A. Trotzdem wird in anderen Arbeiten zu der gleichen Thematik in der Regel davon ausgegangen,

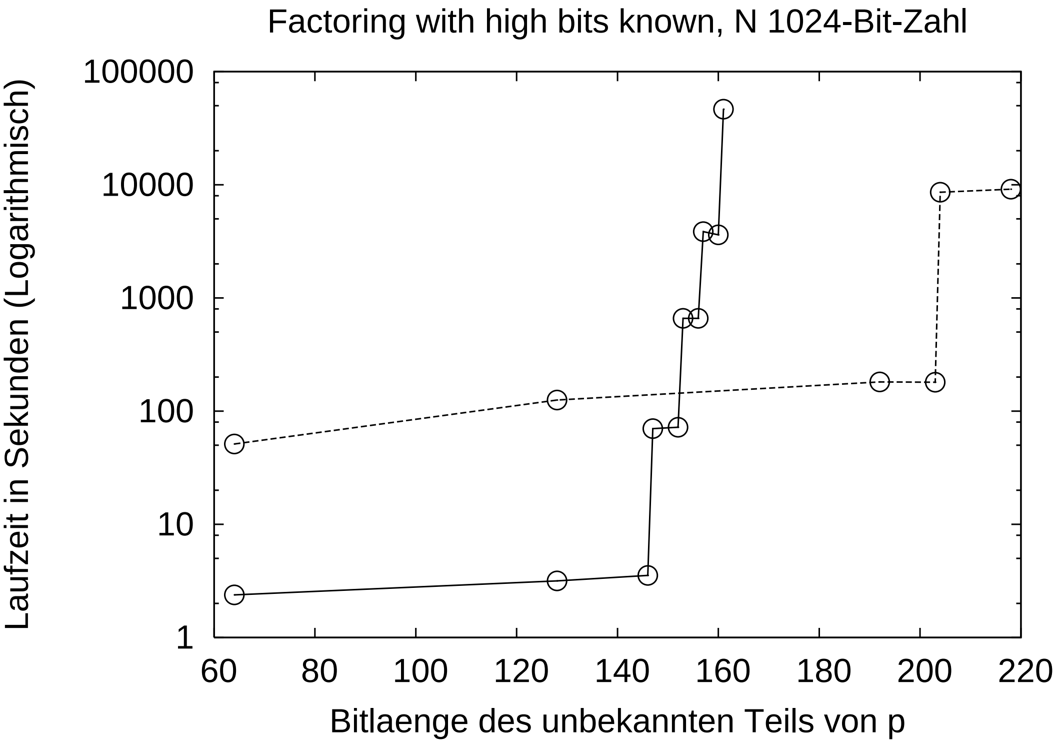

Abbildung 6.2: Laufzeitvergleich von Coppersmiths (durchgehende Linie) und Corons (gestrichelte Linie) Faktorisierungsalgorithmen (N 1024-Bit-Zahl)

dass Approximationen von p immer bitweise vorliegen (die Anwendung der Algorithmen von Coppersmith und Coron zur Faktorisierung wird als „Factoring with high bits known“ bezeichnet). An dieser Stelle ist allerdings wichtig zu sagen, dass dies keine Einschränkung der Algorithmen ist. Sollte es dem Angreifer also möglich sein, p auf andere Art zu approximieren, kann er diese Approximation ebenfalls zur Faktorisierung verwenden. Alle Resultate, die hinreichend gute Näherungen für die Primfaktoren eines RSA-Modulus liefern, können also in Zusammenhang mit den in dieser Arbeit vorgestellten Faktorisierungsalgorithmen interessant sein.

6.2 Zusammenfassung und Ausblick

Coppersmiths Algorithmus zum univariaten Fall lässt sich verwenden, um effizient Teile von Nachrichten zu berechnen, die mit RSA und einem relativ kleinen RSA-Verschlüsselungsexponenten chiffriert worden sind. Wir haben in Abschnitt 2.3 Testergebnisse zu dieser Anwendung mit dem Verschlüsselungsexponenten

$e = 3$ gezeigt, aber auch mit anderen Exponenten kann dieser Angriff angewendet werden.
Der Vergleich der Algorithmen von Coppersmith und Coron zeigt, dass beide Algorithmen verschiedene Stärken und Schwächen aufweisen. Je nach Anwendung kann die Verwendung eines anderen Algorithmus vorteilhaft sein, siehe dazu Abschnitt 6.1.
Ein weiterer Ansatz zur Faktorisierung wird in Blömer, May[6] vorgestellt. In den Abschnitten 4.1 und 3.1 haben wir das Polynom $p(x, y) = (\tilde{p} + x) \cdot (\tilde{q} + y) - N$ zur Faktorisierung verwendet. Eine andere Möglichkeit ist die Verwendung des Polynoms $p(x, y) = (\tilde{p} + x) \cdot y - N$ mit der Nullstelle $(p - \tilde{p}, q)$. Es scheint auf den ersten Blick nicht sinnvoll, auf eine leicht berechenbare Approximation für q zu verzichten und die Grenze Y für die betragsmäßige Größe der Nullstelle y_0 dadurch erhöhen zu müssen, aber ein Ansatz dieser Form bietet in der Theorie auch Vorteile, die es in der praktischen Anwendung noch auszuloten gilt.
Bei realen RSA-Anwendungen ist es oft möglich, Bits des geheimen Dechiffrierexponenten d über sogenannte Seitenangriffe auf den dechiffrierenden Rechner zu erhalten (z.B. über Stromverbrauchsmessung o.ä.). Es existiert eine Heuristik, mit der es möglich ist, unter gewissen Randbedingungen den noch unbekannten Teil von d sowie die Faktoren p und q des Modulus N zu berechnen. Diese Heuristik basiert ebenfalls auf Coppersmith[2] und wird in Ernst, Jochemsz, May, de Weger[12] vorgestellt. Eine Implementierung dieser Angriffsvariante wäre sicherlich als Fortführung dieser Arbeit interessant, da es für einen Angreifer wahrscheinlich ein leichteres Problem darstellt, sich genug Bits von d durch solche Seitenangriffe zu verschaffen, um dann die Heuristik anzuwenden, als sich genug Bits von p bzw. q zu beschaffen, um einen der in dieser Arbeit vorgestellten Faktorisierungsalgorithmen zu verwenden; denn p und q werden im Gegensatz zu d zur Auswertung der Entschlüsselungsfunktion nicht benötigt (siehe Abschnitt 1.1), so dass sie durch Seitenangriffe in der Regel nur bei der Erzeugung des gesamten Schlüssels angreifbar sind.
Zur Implementierung dieser Heuristik wäre es nützlich, eine *C/C++*-Klasse zu implementieren, die es erlaubt, mit trivariaten Polynomen umzugehen. Die Bibliothek *NTL* bietet diese Funktionen nicht an. Im Zusammenhang mit der Anwendung wichtiger Funktionen wie Nullstellenberechnung und Resultantenbildung zu trivariaten Polynomen sollten ebenfalls programmiert und von dieser Klasse zur Verfügung gestellt werden.

Literaturverzeichnis

[1] Stinson, *Cryptography: Theory and Practice, Second Edition,* CRC Press, pages 155-219, 2002

[2] Coppersmith, *Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities,* IBM Research, pages 233-260, 1996

[3] Howgrave-Graham, *Finding small roots of univariate modular equations revisited,* Proceedings of Cryptography and Coding, Lecture Notes in Computer Science Vol. 1355, Springer-Verlag, pages 131-142, 1997

[4] Coron, *Finding Small Roots of Bivariate Integer Polynomial Equations Revisited,* Advances in Cryptology (Eurocrypt 2004), Lecture Notes in Computer Science Volume 3027, Springer-Verlag, pages 1-10, 2004

[5] Lipschutz, *Lineare Algebra* McGraw-Hill, pages 246-279, 1999

[6] Blömer, May, *A Tool Kit for Finding Small Roots of Bivariate Polynomials over the Integers,* Advances in Cryptology (Eurocrypt 2005), Lecture Notes in Computer Science Volume 3494, Springer-Verlag, pages 251-267, 2005

[7] May, *Computing the RSA Secret Key is Deterministic Polynomial Time Equivalent to Factoring,* Advances in Cryptology (Crypto 2004), Lecture Notes in Computer Science Volume 3152, Springer-Verlag, pages 213-219, 2004

[8] Mignotte, *Mathematics for Computer Algebra,* Springer Verlag, pages 259-263, 1992

[9] Mignotte, *An inequality about factors of polynomials,* Math. Comp. 28, pages 1153-1157, 1974

[10] Menezes, van Oorschot, Vanstone, *Handbook of Applied Cryptography,* CRC Press, pages 285-292, 1996

[11] Knuth, *The Art of Computer Programming,* Addison-Wesley, pages 434-435, 1981

[12] ERNST, JOCHEMSZ, MAY, DE WEGER, *Partial Key Exposure Attacks on RSA up to Full Size Exponents,* Advances in Cryptology (Eurocrypt 2005), Lecture Notes in Computer Science Volume 3494, Springer-Verlag, pages 371-386, 2005