Skript zur Vorlesung

Public Key Kryptanalyse

Alexander May

*

Ruhr-Universität Bochum

Wintersemester 2009/10

1 Einleitung

1.1 Kryptologie = Kryptographie + Kryptoanalyse

Als Kryptologie bezeichnet man die Wissenschaft von der Ver- und Entschlüsselung von Informationen. Man gliedert die Kryptologie in die Teilgebiete

- *Kryptographie*, die sich mit der Verschlüsselung von Nachrichten befasst und
- *Kryptoanalyse (auch: Kryptanalyse)*, die sich mit der Entschlüsselung von verschlüsselten Nachrichten befasst.

Dabei unterscheiden wir, ob ein legitimierter Empfänger die Nachricht entschlüsselt oder ein Angreifer.

Der legitimierte Empfänger verwendet Methoden der Kryptographie zum Entschlüsseln, wohingegen ein Angreifer Methoden der Kryptanalyse benutzt. Der grundlegende Unterschied zwischen einem legitimierten Empfänger einer Nachricht und einem Angreifer besteht darin, dass der legitimierte Empfänger eine geheime Informationen besitzt, den *Schlüssel* k. Dieser Schlüssel k ermöglicht es ihm, aus einer verschlüsselten Nachricht, dem sogenannten *Chiffretext* c, die zugrundeliegende Nachricht, den *Klartext* m ($m \mathrel{\hat{=}}$ message), effizient zu berechnen.

Hierbei gilt in der modernen Kryptologie das nach Auguste Kerckhoff benannte Prinzip:

Kerckhoff'sches Prinzip (1883): Die Spezifikation eines Kryptosystems muss öffentlich sein. Die einzige Information, die ein Angreifer nicht kennt, ist der geheime Schlüssel. D.h. die Sicherheit des Kryptosystems beruht einzig auf dem geheimen Schlüssel.

Der Angreifer hat nun zwei Möglichkeiten aus dem Chiffretext c die Nachricht m zu ermitteln:

- Er versucht, in den Besitz des geheimen Schlüssels des legitimierten Empfängers zu kommen. Damit könnte er auch zukünftige Nachrichten an diesen Empfänger effizient entschlüsseln. In diesem Fall ist das Kryptosystem komplett gebrochen.
- Er versucht, die Nachricht m aus dem Chiffretext c zu rekonstruieren, ohne den geheimen Schlüssel zu verwenden. Dies erlaubt ihm im allgemeinen nicht, weitere Nachrichten effizient zu entschlüsseln.

1.2 Public-Key Kryptosystem

Wir wollen uns hier mit Angriffen auf *Public-Key* Kryptosysteme beschäftigen. Diese werden in der Literatur auch *asymmetrische* Kryptosysteme genannt, da man zum Verschlüsseln nur öffentlich bekannte Parameter verwendet, wohingegen die Entschlüsselungsfunktion den geheimen Schlüssel verwendet. D.h. jeder kann eine Nachricht verschlüsseln, aber nur der legitimierte Empfänger sollte in der Lage sein, eine verschlüsselte Nachricht effizient entschlüsseln zu können.

Im Gegensatz dazu gibt es die *Secret-Key* Kryptographie bzw. *symmetrische* Kryptographie, bei der sowohl bei der Verschlüsselung als auch bei der Entschlüsselung der geheime Schlüssel verwendet wird. Bekannte Secret-Key Kryptosysteme sind z.B. DES und AES.

Wir definieren ein Public-Key Kryptosystem wie folgt

Definition 1 (Public-Key Kryptosystem:) *Ein Public-Key Kryptosystem ist ein 5-Tupel* $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ *mit:*

1. $\mathcal{P}$ *ist eine endliche Menge von möglichen Klartexten* ($\mathcal{P} \mathrel{\hat{=}}$ *plaintext).*
2. $\mathcal{C}$ *ist eine endliche Menge von möglichen Chiffretexten* ($\mathcal{C} \mathrel{\hat{=}}$ *ciphertext).*
3. *Der Schlüsselraum* $\mathcal{K}$ *ist eine endliche Menge von möglichen Schlüsseln* ($\mathcal{K} \mathrel{\hat{=}}$ *key space).*
4. *Für jedes* $k \in \mathcal{K}$ *gibt es eine Verschlüsselungsfunktion* $e_k \in \mathcal{E}, e_k : \mathcal{P} \mapsto \mathcal{C}$ *und eine korrespondierende Entschlüsselungsfunktion* $d_k \in \mathcal{D}, d_k : \mathcal{C} \mapsto \mathcal{P}$ *mit*

$$d_k(e_k(m)) = m.$$

5. *Die Verschlüsselungsfunktion* e_k *ist* öffentlich*, wohingegen die Entschlüsselungsfunktion* d_k geheim *ist. Beide Funktionen müssen effizient berechenbar sein.*

Wir werden in den folgenden Kapiteln die bekanntesten Public-Key Kryptosysteme RSA und ElGamal kennenlernen.

Typen von Angreifern und Sicherheit der Verschlüsselung

Da in der Kryptographie immer wieder der Sender einer Nachricht, der Empfänger und ein potentieller Angreifer vorkommen, hat man diesen drei Hauptakteuren eigene Namen gegeben:

Alice sendet eine verschlüsselte Nachricht an Bob, und die böse Eve (= Eavesdropper) belauscht die Kommunikation der beiden und versucht, die Nachricht zu entschlüsseln. Dabei unterscheiden wir in der Kryptologie verschiedene Typen von Angriffen nach den Möglichkeiten, die Eve zur Verfügung hat. Wir ordnen die Angriffe hier aufsteigend nach Stärke:

Ciphertext only: Eve hat nur Kopien von Chiffretexten.

Known Plaintext Angriff (KPA): Eve hat Kopien von Klartexten mit zugehörigen Chiffretexten. Dabei kann sie allerdings die Klartexte nicht selbst wählen.

Chosen Plaintext Angriff (CPA): Eve kann sich Chiffretext von beliebigen Klartexten ihrer Wahl selbst erzeugen.

Chosen Ciphertext Angriff (CCA): Eve hat Zugriff auf ein Entschlüsselungsorakel und kann sich Chiffretexte ihrer Wahl entschlüsseln lassen.

Man beachte, dass bei Public-Key Kryptosystemen der Angreifer stets einen Chosen Plaintext Angriff durchführen kann, da die Verschlüsselungsfunktion öffentlich bekannt ist.

Wir nennen ein System *Chosen Plaintext sicher* (CPA secure), wenn es sicher unter Chosen Plaintext Angriffen ist. Analog heißt ein System *Chosen Ciphertext sicher* (CCA secure), wenn es sicher unter Chosen Ciphertext Angriffen ist.

Was heißt es aber für ein Kryptosystem, "sicher zu sein"? Naiv würde man sagen, dass es einem Angreifer Eve nicht effizient möglich sein sollte, aus einem Chiffretext den zugehörigen Klartext zu rekonstruieren. Es gibt aber eine einige darüber hinausgehende Eigenschaften, die wünschenswert sind:

- Wir müssen annehmen, dass Eve den Nachrichtenraum $\mathcal{M} \subseteq \mathcal{P}$, d.h. die Menge aller möglichen Nachrichten kennt. Dies kann die Menge aller möglichen deutschen Sätzen sein, es kann aber auch nur die Menge {Ja, Nein} sein. Zusätzlich müssen wir annnehmen, dass Eve die Wahrscheinlichkeitsverteilung auf dem Nachrichtenraum $\mathcal{M}$ kennt. Trotzdem sollte ihr das nicht bei der Entschlüsselung helfen.

- Es sollte für Eve schwer sein, partielle Informationen über den zugrundeliegenden Klartext zu erfahren, wie z.B. die erste Hälfte des Klartextes.

- Es sollte schwer sein, nützliche Informationen über den Nachrichtenverkehr zu erfahren, wie z.B. die Tatsache dass dieselbe Nachricht zweimal geschickt wurde.

- Die obigen Eigenschaften müssen mit hoher Wahrscheinlichkeit gelten.

D.h. es wäre wünschenswert, dass ein Chiffretext ein digitales Analog zum klassischen, blickdichten Briefumschlag ist. Sieht man zwei Briefumschläge von außen, dann kann man die innenliegenden Briefe nicht unterscheiden. Oder in der digitalen Welt: Ein Angreifer Eve darf nicht in der Lage sein, anhand von zwei Chiffretexten die zugrundeliegenden Nachrichten zu unterscheiden.

Daher führen wir den folgenden Test ein, in dem Eve die Verschlüsselung zweier selbstgewählter Nachrichten unterscheiden muss:

1. Eve wählt zwei Nachrichten m_0, m_1 aus dem Klartextraum $\mathcal{P}$ und sendet sie an Alice.
2. Alice wählt $b \in \{0, 1\}$ zufällig und sendet $e_k(m_b)$ zurück an Eve.
3. Eve muss entscheiden, ob m_0 oder m_1 verschlüsselt wurde. D.h. sie versucht, das Bit b zu raten.

Eve kann das Bit b natürlich stets mit Wahrscheinlichkeit $\frac{1}{2}$ erraten. Sei b' die Ausgabe von Eve. Wir nennen die Wahrscheinlichkeit

$$\Pr(b = b') - \frac{1}{2}$$

den Vorteil von Eve.

Wir bezeichnen ein Kryptosystem als *semantisch sicher*, wenn der Vorteil eines Angreifers vernachlässigbar klein ist (d.h. der Vorteil ist kleiner als das Inverse jeder polynomiellen Funktion im Sicherheitsparameter des Kryptosystems).

Übung 2 *Kein Public-Key Kryptosystem mit* deterministischer *Verschlüsselungsfunktion ist semantisch sicher.*

Man sollte beachten, dass semantische Sicherheit ein starker Sicherheitsbegriff ist. Zudem gibt ein Chosen Ciphertext Angriff einem Angreifer umfangreiche Möglichkeiten. Erstaunlicherweise kennt man trotzdem Public-Key Kryptosysteme, von denen man unter gewissen Komplexitätsannahmen zeigen kann, dass sie semantisch sicher gegenüber Chosen Ciphertext Angriffen sind (CCA sicher).

Wir werden im nachfolgenden Kapitel die bekanntesten Public-Key Kryptosysteme RSA und ElGamal kennenlernen und sehen, dass diese Systeme zunächst nicht semantisch sicher gegenüber CCA sind. Es gibt aber Varianten beider Systeme von denen man die CCA-Sicherheit zeigen kann.

1.3 Digitale Signaturen

Ein Signaturverfahren ist ebenso wie ein Public-Key Kryptosystem ein asymmetrisches Verfahren, d.h. ein Benutzer hat einen öffentlichen und einen korrespondierenden geheimen Schlüssel. Unter Verwendung des geheimen Schlüssels kann eine Nutzerin Alice eine digitale Signatur an eine Nachricht anhängen, deren Korrekheit jeder mit Hilfe des öffentlichen Schlüssels von Alice überprüfen kann.

Nehmen wir an, dass Alice unter Verwendung ihres geheimen Schlüssels k eine Nachricht m digital signiert. Bezeichnen wir die digitale Signatur mit $\text{sig}_k(m)$. Nun veröffentlicht Alice das Tupel $(m, \text{sig}_k(m))$. Durch das Anhängen einer Unterschrift $\text{sig}_k(m)$ an eine Nachricht m will man die folgenden Eigenschaften erzielen:

Authentizität: Jeder kann überprüfen, dass wirklich Alice das Dokument m unterschrieben hat, da die Signatur mit Alices öffentlichem Schlüssel überprüft werden muss. Es sollte für einen Angreifer nicht möglich sein, eine Unterschrift zu fälschen, d.h. eine Unterschrift von einer beliebigen Nachricht m' zu erzeugen, die mit Alices öffentlichem Schlüssel verifiziert werden kann.

Nicht-Abstreitbarkeit: Da die Unterschrift $\text{sig}_k(m)$ eine Funktion von Alices geheimen Schlüssel ist, kann Alice auch nicht abstreiten, dass sie die Unterschrift gegeben hat. Sie kann nicht behaupten, dass ein anderer die Unterschrift geleistet hat. Daher ist eine Unterschrift einer Person an einen Nachrichtentext (z.B. einen Vertragstext) bindend.

Integrität: Die Unterschrift $\text{sig}_k(m)$ ist ebenfalls eine Funktion der Nachricht m. D.h. es soll sichergestellt werden, dass eine Nachricht m nicht verändert wird. Einem Angreifer soll es nicht möglich sein, die Nachricht m durch eine andere Nachricht m' (z.B. einen anderen Vertragstext) zu ersetzen, so dass $\text{sig}_k(m)$ eine gültige Signatur von m' ist.

Wir definieren nun ein Signaturschema und anschließend analog zum Public-Key Kryptosystem verschiedene Typen von Angreifern sowie verschiedene Sicherheitsziele.

Definition 3 (Signaturschema) *Ein* Signaturschema *ist ein* 5*-Tupel* $(\mathcal{P}, \mathcal{U}, \mathcal{K}, \mathcal{S}, \mathcal{V})$ *mit den folgenden Eigenschaften:*

1. $\mathcal{P}$ *ist eine endliche Menge von möglichen Nachrichten.*
2. $\mathcal{U}$ *ist eine endliche Menge von möglichen Unterschriften.*
3. *Der Schlüsselraum* $\mathcal{K}$ *ist eine endliche Menge von Schüsseln.*
4. *Für jedes* $k \in \mathcal{K}$ *gibt es eine Signierfunktion* $sig_k \in \mathcal{S}$, $sig_k : \mathcal{P} \to \mathcal{U}$ *und eine korrespondierende Verifikationsfunktion* $ver_k \in \mathcal{V}$, $ver_k : \mathcal{P} \times \mathcal{U} \to \{wahr, falsch\}$, *so dass für alle* $x \in \mathcal{P}$, $y \in \mathcal{U}$ *gilt:*

$$ver_k(x, y) = wahr \;\Leftrightarrow\; y = sig_k(x).$$

5. *Die Signierfunktion* sig_k *ist* geheim, *wohingegen die Verifikationsfunktion* ver_k öffentlich *ist. Beide Funktionen müssen effizient berechenbar sein.*

Wir werden in den folgenden Kapitel das RSA-Signaturschema und das DSA-Signaturverfahren kennenlernen.

Typen von Angriffen und Sicherheitsziele

Wir betrachten bei Signaturschemata genau wie bei Public-Key Kryptosystemen verschiedene Typen von Angreifern Eve und ordnen diese aufsteigend nach der Stärke des Angriffs:

Key only: Der Angreifer Eve kennt nur den öffentlichen Schlüssel des Unterschreibers und kann überprüfen, ob eine Unterschrift zu einer Nachricht gültig ist.

Known Signature Angriff: Der Angreifer Eve kennt den öffentlichen Schlüssel und einige gültige Nachrichten/Unterschrift-Paare des Unterschreibers, wobei Eve keine Kontrolle über die Nachrichten hat.

Chosen Message Angriff: Der Angreifer Eve kann sich Nachrichten seiner Wahl vom Unterschreiber signieren lassen. Dabei kann er die Nachricht adaptiv wählen, d.h. abhängig von den zuvor gesehenen Nachricht/Unterschrift-Paaren.

In der Realität muss man meistens davon ausgehen, dass ein Angreifer zumindest einen Known Signature Angriff auf ein Signaturverfahren durchführen kann, da er Nachricht/ Unterschrift-Paare des Unterzeichners abhören kann. Für einen Chosen Message Angriff benötigt er allerdings ein sogenanntes *Signier-Orakel*. Ein solches Orakel ist sicherlich nicht für jeden Angreifer verfügbar. Trotzdem ist es aber sinnvoll, Fälschungssicherheit von Unterschriften auch gegen Angreifer mit sehr weitreichenden Möglichkeiten sicherzustellen.

Was bedeutet es nun aber, dass ein Angreifer eine Nachricht erfolgreich fälschen kann? Wir geben hier eine Liste von Zielen für einen Angreifer Eve in Reihenfolge aufsteigenden Erfolgs.

Existentielle Fälschung: Eve kann die Unterschrift einer Nachricht x fälschen, wobei sie nicht notwendigerweise Kontrolle über x hat.

Selektive Fälschung: Eve kann eine Unterschrift zu einer von ihr gewählten Nachricht x konstruieren.

Universelle Fälschung: Obwohl sie den geheimen Schlüssel des Unterschreibers nicht kennt, kann Eve Unterschriften zu beliebigen Nachrichten fälschen.

Total break: Eve kann den geheimen Schlüssel des Unterzeichners berechnen.

Wir nennen ein Signaturverfahren *sicher*, wenn es keinen Angreifer gibt, der mit Hilfe eines Chosen Message Angriffs in der Lage ist, eine existentielle Fälschung zu berechnen (in Zeit polynomiell im Sicherheitsparameter des Signaturschemas). D.h. nachdem der Angreifer eine Reihe von Nachricht/Unterschrift-Paaren gesehen hat, sollte er nicht in der Lage sein, die Unterschrift einer Nachricht, die er noch nicht gesehen hat, zu fälschen.

Analog zur semantischen Sicherheit bei Public-Key Kryptosystem definieren wir also wieder die Sicherheit hinsichtlich des *stärksten Angreifers* und des *schwächsten Angriffsziels*. Auch hier wird es so sein, dass die klassischen Signaturverfahren wie RSA nicht sicher bezüglich dieser Definition sind. Aber auch hier kann man Varianten dieser Systeme definieren, die unter gewissen Annahmen beweisbar sicher sind. Dies ist allerdings

nicht das Ziel dieser Vorlesung. Wir wollen uns nicht auf beweisbar sichere Systeme konzentrieren, sondern auf Angriffe für Praxis-relevante Systeme.

Bevor wir allerdings Public-Key Kryptosysteme, Signaturverfahren und Angriffe auf dieselben beschreiben können, benötigen wir zunächst ein paar mathematische Grundlagen.

2 Mathematische Grundlagen der Public-Key Kryptographie

Fast alle Public-Key Kryptosysteme, die wir heutzutage kennen, basieren auf Arithmetik in Gruppen. Daher wollen wir hier kurz einen Überblick geben, was eine Gruppe ist und welche Eigenschaften in einer Gruppe gelten

Eine Gruppe ist eine Menge zusammen mit einer Verknüpfung, die gewisse Eigenschaften erfüllt:

Definition 4 (abelsche Gruppe) *Eine* abelsche Gruppe *ist ein Tupel* $(G, \circ)$*, bestehend aus einer nichtleeren Menge* G *und einer Verknüpfung* $\circ$ *mit den folgenden Eigenschaften:*

Abgeschlossenheit: *Die Verknüpfung* $\circ$ *definiert eine Abbildung* $\circ : G \times G \to G$, $(a, b) \mapsto a \circ b$.

Assoziativität: *Für alle* $a, b, c \in G$ *gilt:* $(a \circ b) \circ c = a \circ (b \circ c)$

Kommutativität: *Für alle* $a, b \in G$ *gilt:* $a \circ b = b \circ a$

neutrales Element: *Es gibt ein (eindeutiges)* $e \in G : a \circ e = a$ *für alle* $a \in G$

inverses Element: *Für jedes* $a \in G$ *gibt es ein (eindeutiges)* $a^{-1} \in G$ *mit:* $a \circ a^{-1} = e$.

Es gibt auch nicht-abelsche Gruppen. Diese besitzen die obigen Eigenschaften außer der Kommutativität. Da wir im folgenden aber stets kommutative Gruppen betrachten werden, bezeichnen wir der Einfachheit halber abelsche Gruppen einfach als *Gruppen.* Wenn die Verknüpfung $\circ$ eine Addition ist, so sprechen wir von additiven Gruppen, falls die Verknüpfung eine Multiplikation ist, so sprechen wir von multiplikativen Gruppen. Statt von $(G, \cdot)$ sprechen wir dann auch einfach von der multiplikativen Gruppe G.

Betrachten wir einige Beispiel für Gruppen:

Beispiel 5

1. $\mathbb{Z}$ *ist eine additive Gruppe. Das neutrale Element ist die* 0*. Das inverse Element zu* $a \in \mathbb{Z}$ *ist* $-a$.

2. $\mathbb{Q} \setminus \{0\}$ *ist eine multiplikative Gruppe. Das neutrale Element ist die* 1*. Das inverse Element zu* $\frac{a}{b} \in \mathbb{Q}$ *ist* $\frac{b}{a}$.

3. *$\mathbb{Z}$ ist keine multiplikative Gruppe, denn 1 ist zwar ein neutrales Element, aber es gibt kein Inverses von 2 in den ganzen Zahlen.*

4. *Die Menge $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ zusammen mit der Addition modulo n ist eine additive Gruppe. Das neutrale Element der Gruppe ist die Null. Das Inverse zu $a \neq 0$ ist $n - a$.*

Aus Beispiel 5 wissen wir, dass $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ zusammen mit der Addition modulo n eine additive Gruppe ist. Da die Menge $\mathbb{Z}_n$ endlich ist, bezeichnet man eine solche Gruppe auch als *endliche* Gruppe.

Wir werden im folgenden zeigen, dass diejenigen Zahlen in $\mathbb{Z}_n$, die zu n teilerfremd sind, eine endliche multiplikative Gruppe bilden. Diese endliche Gruppe ist von sehr großer Bedeutung in der Public-Key Kryptographie und wird zur Definition der Public-Key Kryptosyteme RSA und ElGamal benötigt.

Definition 6 ($\mathbb{Z}_n^*$) *Sei $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$. Wir definieren die Menge $\mathbb{Z}_n^*$ als die Menge*

$$\mathbb{Z}_n^* := \{a \in \mathbb{Z}_n \mid ggT(a, n) = 1\}.$$

Die Größe dieser Menge wird durch die sogenannte Eulersche ϕ-Funktion *beschrieben*

$$\phi(n) := |\mathbb{Z}_n^*|.$$

Übung 7 *Sei $n = pq$, wobei p, q prim sind. Zeigen Sie, dass $\phi(n) = (p-1)(q-1)$.*

Betrachten wir nun $(\mathbb{Z}_n^*, \cdot)$, d.h. die Menge $\mathbb{Z}_n^*$ zusammen mit der Multiplikation modulo n. Das neutrale Element der Multiplikation ist die 1, Abgeschlossenheit, Assoziativität und Kommutativität der Multiplikation modulo n gelten. D.h. um zu zeigen, dass $\mathbb{Z}_n^*$ eine multiplikative Gruppe ist, genügt es zu zeigen, dass es für jedes $a \in \mathbb{Z}_n^*$ ein $a^{-1} \in \mathbb{Z}_n^*$ gibt mit $a \cdot a^{-1} = 1 \bmod n$.

Hierzu verwenden wir einen bekannten Satz aus der Algebra zum Erweiterten Euklidischen Algorithmus (EEA).

Fakt 8 (Erw. Euklidischer Algorithmus) *Seien $a, n \in \mathbb{Z}$. Dann kann man in Zeit $\mathcal{O}(\log^2(\max\{a, n\}))$ ganze Zahlen u, v berechnen mit*

$$a \cdot u + n \cdot v = ggT(a, n).$$

Satz 9 *Zu jedem $a \in \mathbb{Z}_n^*$ kann man ein Inverses $a^{-1} \in \mathbb{Z}_n^*$ mit $a \cdot a^{-1}$ in Zeit $\mathcal{O}(\log^2 n)$ bestimmen.*

Beweis: Berechne mit dem EEA ganze Zahlen u, v mit $au + nv = \mathrm{ggT}(a, n)$. Da $a \in \mathbb{Z}_n^*$ ist, gilt $\mathrm{ggT}(a, n) = 1$. Daraus folgt $au + nv = 1$ bzw. $au = 1 \bmod n$. D.h. $a^{-1} := u \bmod n$ ist unser gesuchtes Inverses von a.

Korollar 10 *$\mathbb{Z}_n^*$ ist eine multiplikative Gruppe der Größe $|\mathbb{Z}_n^*| = \phi(n)$.*

Wir sprechen auch abkürzend von der multiplikativen Gruppe $\mathbb{Z}_n^*$. Ein wichtiger Spezialfall ist die Menge $\mathbb{Z}_p$, wobei p eine Primzahl ist. In $\mathbb{Z}_p$ sind alle Zahlen außer der Null teilerfremd zu p. Daher ist $\mathbb{Z}_p^* = \mathbb{Z}_p \setminus \{0\}$ eine multiplikative Gruppe.

Machen wir ein einfaches Beispiel für eine Inversenberechnung mit Hilfe des Erweiterten Euklidischen Algorithmus.

Beispiel 11 *Wir berechnen das Inverse von* 11 *in der multiplikativen Gruppe Z_{15}^*. Für den größten gemeinsamen Teiler gilt $ggT(n, a) = ggT(a, n \bmod a)$. Daher gilt*

$$ggT(15, 11) = ggT(11, 4) = ggT(4, 3) = ggT(3, 1) = 1.$$

Wir schreiben dies in Form von drei Gleichungen:

$$\begin{aligned} 15 - 1 \cdot \mathbf{11} &= \mathbf{4} \\ 11 - 2 \cdot \mathbf{4} &= \mathbf{3} \\ 4 - 1 \cdot \mathbf{3} &= \mathbf{1} \end{aligned}$$

Jetzt ersetzen wir die 3 *in der dritten Gleichung durch den Ausdruck in der zweiten Gleichung:*

$$1 = 4 - 1 \cdot (11 - 2 \cdot 4) = 3 \cdot 4 - 11$$

Analog ersetzen wir nun die 4 *durch den Ausdruck in der ersten Gleichung:*

$$1 = 3 \cdot (15 - 11) - 11 = 3 \cdot 15 - 4 \cdot 11.$$

D.h. $-4 = 11 \mod 15$ *ist das Inverse von* 11 *in $\mathbb{Z}_{15}^*$. Eine Probe liefert* $11 \cdot 11 = 121 = 1 \bmod 15$.

Weiterhin definieren wir folgende wichtige Kenngrößen einer Gruppe.

Definition 12 (Ordnung) *Sei G eine multiplikative endliche Gruppe mit neutralem Element* 1.

1. *Die* Ordnung der Gruppe *G wird bezeichnet mit $ord(G)$ und ist definiert als die Anzahl der Elemente in G: $ord(G) = |G|$.*
2. *Die* Ordnung eines Elements *$a \in G$ wird bezeichnet mit $ord(a)$ und ist definiert als $ord(a) = \min\{i \in \mathbb{N} \setminus \{0\} \mid a^i = 1\}$*
3. *Man bezeichnet $H \subseteq G$ als* Untergruppe von *G, falls H eine Gruppe ist.*

4. *Sei $a \in G$. Wir definieren $\langle a \rangle := \{a, a^2, \ldots, a^{ord(a)}\}$. Man bezeichnet $\langle a \rangle$ auch als die von a* generierte Untergruppe *(dass $\langle a \rangle$ die Gruppeneigenschaften erfüllt, ist eine Übungsaufgabe). Das Element a bezeichnet man als* Generator der Untergruppe. *(Unter-)Gruppen, die von einem Element generiert werden, nennt man* zyklisch.

Geben wir ein Beispiel zu obiger Definition.

Beispiel 13 *Betrachten wir die multiplikative Gruppe $\mathbb{Z}_7^*$.*

1. *$ord(\mathbb{Z}_7^*) = 6$, denn $|\mathbb{Z}_7^*| = |\{1, 2, \ldots, 6\}|$.*
2. *$ord(4) = 3$, denn $4^1 = 4$, $4^2 = 2$, $4^3 = 1$.*
3. *$H = \{1, 2, 4\}$ ist eine Untergruppe. Es $2 \cdot 4 = 1$, d.h. $2^{-1} = 4$ und $4^{-1} = 2$.*
4. *Es gilt $H = \langle 4 \rangle$, d.h. H ist die von 4 erzeugte zyklische Untergruppe.*

In jeder endlichen Gruppe gilt der Satz von Euler.

Satz 14 (Euler) *Sei G eine endliche multiplikative Gruppe mit neutralem Element 1. Dann gilt für alle $a \in G$:*

$$a^{|G|} = 1.$$

Beweis: Sei $|G| = n$ und $G = \{g_1, g_2, \ldots g_n\}$. Betrachte die Abbildung:

$$f : G \to G, \ f(g) = ag.$$

Diese Abbildung ist eine Bijektion, da die Multiplikation mit a eine invertierbare Operation ist. D.h. aber

$$\{g_1, g_2, \ldots, g_n\} = \{ag_1, ag_2, \ldots, ag_n\}.$$

Hieraus folgt insbesondere

$$\prod_{i=1}^{n} g_i = \prod_{i=1}^{n} ag_i = a^n \prod_{i=1}^{n} g_i.$$

Daraus folgt $a^n = 1$.

Korollar 15 (Fermat) *Sei p prim. Dann gilt für alle $a \in Z_p^*$*

$$a^{p-1} = 1 \bmod p.$$

Wir haben zuvor in einem Beispiel gesehen, dass 11 das Inverse von 11 in der multiplikativen Gruppe Z_{15}^*. D.h. mit anderen Worten die lineare Gleichung $11x = 1$ hat die Lösung $x = 11$ in $\mathbb{Z}_{15}$. Wenn n eine zusammengesetzte Zahl ist und man statt einer linearen eine quadratische Gleichung in Z_n lösen will, dann kennt man keinen effizienten Algorithmus. D.h. man kennt keinen Algorithmus mit einer Laufzeit polynomiell in $\log n$ (polynomiell in der Bitlänge von n).

Wenn man allerdings die Primfaktorzerlegung von n kennt, kann man wie folgt vorgehen:

1. Man splittet n zunächst in seine Primzahlpotenzen auf: $n = \prod_i p_i^{e_i}$.
2. Daraufhin löst man die quadratische Gleichung modulo der einzelnen Primfaktoren $p_i^{e_i}$. Für dieses Problem kennt man effiziente probabilistische Verfahren.
3. Anschließend setzt man die einzelnen Lösungen dann wieder modulo n zusammen.

Der letzte Schritt ist effizient berechenbar dank des sogenannten Chinesischen Restsatzes. Die Konstruktion der Lösung modulo n aus den Lösungen modulo der Primteiler ist eine Anwendung des Erweiterten Euklidischen Algorithmus (Fakt 8).

Satz 16 (Chinesischer Restsatz) *Seien m, n teilerfremde natürliche Zahlen. Es existiert genau eine Lösung $x \bmod mn$ des Gleichungssystems*

$$\left| \begin{array}{rcl} x & = & a \bmod m \\ x & = & b \bmod n \end{array} \right| .$$

Beweis: Wegen $\mathrm{ggT}(m,n) = 1$ liefert der Erweiterte Euklidische Algorithmus u, v mit $um + vn = 1$. D.h. es gilt $um = 1 \bmod n$ und $vn = 1 \bmod m$. Sei $x = bum + avn$. Dann gilt offenbar

$$x = a \bmod m \quad \text{und} \quad x = b \bmod n.$$

Das heisst es existiert eine Lösung x. Wir müssen noch zeigen, dass diese Lösung modulo mn eindeutig ist. Angenommen x' wäre eine zweite Lösung. Dann gilt

$$x = a = x' \bmod m \quad \text{und} \quad x = b = x' \bmod n.$$

Damit teilen sowohl m als auch n die Differenz $x - x'$. Da m und n teilerfremd sind, teilt dann auch mn die Differenz $x - x'$, d.h. $x - x' = 0 \bmod mn$ bzw. $x = x' \bmod mn$.

Man beachte, dass der Chinesische Restsatz eine injektive Abbildung $\mathbb{Z}_m \times \mathbb{Z}_n \to \mathbb{Z}_{mn}$ liefert. Jedes Tupel $(a,b) \in \mathbb{Z}_m \times \mathbb{Z}_n$ entspricht genau einem $x \in \mathbb{Z}_{mn}$. Nun besitzt aber der Urbildbereich der Abbildung genauso wie der Bildbereich jeweils mn Elemente. Daher ist die Abbildung surjektiv und damit auch bijektiv. Das heißt jedem Tupel (a,b) entspricht eineindeutig ein x, bzw. die Abbildung definiert einen Isomorphismus $\mathbb{Z}_m \times \mathbb{Z}_n \cong \mathbb{Z}_{mn}$.

Dieser Isomorphismus bedeutet für uns, dass wir beliebig zwischen den Darstellungen als Element aus $\mathbb{Z}_{mn}$ bzw. als Element aus $\mathbb{Z}_m \times \mathbb{Z}_n$ wechseln können. Wir werden das in dem folgenden Beispiel veranschaulichen.

Beispiel: Wir wollen alle Lösungen der Gleichung $x^2 = 1 \mod 15$ bestimmen. Da $15 = 3 \cdot 5$, gilt $x^2 = 1 \mod 15$ genau dann wenn

$$\left| \begin{array}{rcl} x^2 & = & 1 \bmod 3 \\ x^2 & = & 1 \bmod 5 \end{array} \right| .$$

Wir wechseln also zunächst von der Darstellung in $\mathbb{Z}_{15}$ auf die Darstellung in $\mathbb{Z}_3 \times Z_5$. Es ist leicht zu sehen, dass beide Gleichungen ± 1 als Lösung besitzen und das dies die einzigen Lösungen sind. Jeder Kombination dieser einzelnen Lösungen ist eine Lösung des Gleichungssystem, d.h. wir erhalten die vier Lösungen

$$\left|\begin{array}{lll} x_1 = 1 & \bmod 3 \\ x_1 = 1 & \bmod 5 \end{array}\right|, \left|\begin{array}{lll} x_2 = -1 & \bmod 3 \\ x_2 = -1 & \bmod 5 \end{array}\right|, \left|\begin{array}{lll} x_3 = 1 & \bmod 3 \\ x_3 = -1 & \bmod 5 \end{array}\right|, \left|\begin{array}{lll} x_4 = -1 & \bmod 3 \\ x_4 = 1 & \bmod 5 \end{array}\right|.$$

Nun wechseln wir wieder von der Darstellung in $\mathbb{Z}_3 \times \mathbb{Z}_5$ zurück zu der Darstellung in $\mathbb{Z}_{15}$ und erhalten mit Hilfe des Chinesischen Restsatzes die 4 Lösungen $x_1 = 1$, $x_2 = 14$, $x_3 = 4$ und $x_4 = 11$.

Es ist eine Übungsaufgabe, die folgende verallgemeinerte Form des Chinesischen Restsatzes zu beweisen.

Übung 17 (Verallg. Chinesischer Restsatz) *Seien $m_1, m_2, \ldots, m_n$ teilerfremde natürliche Zahlen. Es existiert genau eine Lösung $x \bmod m_1 m_2 \ldots m_n$ des Gleichungssystems*

$$\left|\begin{array}{rcl} x & = & a_1 \bmod m_1 \\ x & = & a_2 \bmod m_2 \\ & \vdots & \\ x & = & a_n \bmod m_n \end{array}\right|.$$

3 RSA & Meet-in-the-Middle Angriffe

3.1 Das RSA Kryptosystem

Jetzt können wir das RSA Public-Key Kryptosystem beschreiben.

RSA Kryptosystem

Sei $N = pq$, wobei p und q Primzahlen sind. Sei $\mathcal{P} = \mathcal{C} = \mathbb{Z}_N$. Wie definieren den Schlüsselraum $\mathcal{K}$ als

$$\mathcal{K} := \{(N, e, d) \mid ed = 1 \bmod \phi(N)\}.$$

Die öffentliche Verschlüsselungsfunktion $e_k : \mathbb{Z}_N \to \mathbb{Z}_N$ ist definiert als

$$e_k(m) := m^e \bmod N.$$

Die geheime Entschlüsselungsfunktion $d_k : \mathbb{Z}_N \to \mathbb{Z}_N$ ist analog definiert als

$$d_k(c) := c^d \bmod N.$$

Da e_k durch die Parameter N und e definiert ist, nennen wir das Tupel (N, e) den *öffentlichen Schlüssel*. N wird dabei als *RSA Modul* bezeichnet und e als *öffentlicher Exponent*. Den geheimen Parameter d bezeichnen wir als den *geheimen Schlüssel* oder auch den *geheimen Exponenten*.

Man beachte, dass man die Primzahlen p und q nicht zur Entschlüsselung benötigt. Trotzdem werden wir später eine Variante der RSA-Entschlüsselung (sog. CRT-RSA) kennenlernen, bei der man die Primfaktorzerlegung benutzt, um effizient zu entschlüsseln.

Zunächst wollen wir aber zeigen, dass die Ver- und Entschlüsselungsfunktionen effizient berechnet werden können.

Satz 18 *Gegeben $b, n \in \mathbb{N}$ und $a \in \mathbb{Z}_n$. Dann kann man $a^b \bmod n$ in Zeit $\log(b)\log^2(n)$ berechnen.*

Beweis: Sei b eine m-Bit Zahl. Wir schreiben $b = \sum_{i=0}^{m-1} b_i 2^i$ in der Binärdarstellung $b_{m-1} \dots b_0$. Nun verwenden wir den folgenden Algorithmus, um $a^b \bmod n$ zu berechnen.

Algorithmus Repeated Squaring

EINGABE: $n, a, b = b_{m-1} \dots b_0$

1. Setze $z := 1$.
2. FOR $i = 0$ TO $m - 1$
   a) IF $(b_i = 1)$ setze $z := z \cdot a \bmod n$.
   b) IF $(i < m - 1)$ setze $a := a^2 \bmod n$.

AUSGABE: $z = a^b \bmod n$

Korrektheit: Schreibe

$$a^b = a^{\sum_{i=0}^{m-1} b_i 2^i} = \prod_{i=0}^{m-1} a^{b_i 2^i} = \prod_{i=0}^{m-1} (a^{2^i})^{b_i}.$$

In der Variablen a steht zu Beginn der i-ten Iteration der Wert a^{2^i}. Dieser wird zum Zwischenergebnis z multipliziert, falls $b_i = 1$ ist.

Es werden $m = \mathcal{O}(\log b)$ Iterationen der Schleife durchlaufen, wobei jede Iteration Laufzeit $\mathcal{O}(\log^2 n)$ benötigt. Daraus folgt die Gesamtlaufzeit.

Korollar 19 *Die Berechnung der RSA-Verschlüsselungsfunktion e_k benötigt Laufzeit $\mathcal{O}(\log e \log^2 N)$, die Berechnung der RSA-Entschlüsselungsfunktion d_k benötigt Laufzeit $\mathcal{O}(\log d \log^2 N)$.*

In der Praxis wählt man oft einen kleinen Verschlüsselungsexponenten e, um die Laufzeit der Verschlüsselung zu minimieren. Beliebt ist z.B. die Wahl $e = 3$ oder $e = 2^{16} + 1$. Wir werden sehen, dass dies in gewissen Situationen zu Angriffen führen kann.

Analog könnte man auch ein kleines d verwenden. Wir werden die Sicherheit von RSA mit kleinem geheimen d ausgiebig untersuchen. Mit Hilfe von sogenannten Gitterangriffen werden wir zeigen, dass Entschlüsselungsexponenten, die kleiner als $1/4$ der Bitlänge von N sind, aus den öffentlichen Parametern in Zeit polynomiell in $\mathcal{O}(\log N)$ berechnet werden können.

Wir müssen noch zeigen, dass RSA wirklich ein Public-Key System gemäß Definition 1 ist. Dazu bleibt zu zeigen, dass für alle $m \in Z_n$ gilt $d_k(e_k(m)) = m$. Wir zeigen dies hier

zunächst für alle $m \in Z_N^*$. Wie wir aus Kapitel 2 wissen, hat die multiplikative Gruppe Z_N^* die Ordnung $\text{ord}(\mathbb{Z}_N^*) = \phi(N) = (p-1)(q-1)$. Wir schreiben $ed = 1 \bmod \phi(N)$ als $ed = 1 + k\phi(N)$ für ein unbekanntes $k \in \mathbb{N}$. Nun gilt für alle $m \in \mathbb{Z}_N^*$:

$$d_k(e_k(m)) = (m^e)^d = m^{ed} = m^{1+k\phi(N)} = m \cdot (m^{\phi(N)})^k = m \bmod N,$$

wobei die letzte Gleichung aus dem Satz von Euler folgt (Satz 14). Es bleibt noch zu zeigen, dass die obige Identität auch für $m \in \mathbb{Z}_N \setminus \mathbb{Z}_N^*$ gilt. Dies ist eine Übungsaufgabe.

RSA Signaturen

Man beachte, dass die Funktionen e_k und d_k kommutativ sind. Das bedeutet es gilt für alle $m \in \mathbb{Z}_N$ auch:

$$e_k(d_k(m)) = m.$$

Daher kann man RSA als Signaturschema im Sinne von Definition 3 verwenden. Wir definieren $\mathcal{P} = \mathcal{U} = \mathbb{Z}_N$ und den Schlüsselraum $\mathcal{K}$ genau wie beim RSA Kryptosystem.

Die Signierfunktion ist definiert als $\text{sig}_k(m) := m^d \bmod N$. Die korrespondierende Verifikationsfunktion ist definiert als

$$\text{ver}_k(m, \text{sig}_k(m)) := \begin{cases} \text{wahr} & \text{für } \text{sig}_k(m)^e = m \bmod N \\ \text{falsch} & \text{sonst} \end{cases}$$

Man beachte, dass RSA *multiplikativ* ist, d.h.

$$m_1^e \cdot m_2^e = (m_1 m_2)^e \bmod N.$$

Dies führt dazu, dass das RSA Public-Key Kryptosystem nicht sicher ist gegen Chosen Ciphertext Angriffe und das RSA Signaturverfahren nicht sicher ist gegen Chosen Message Angriffe.

Übung 20 *Gegeben seien ein RSA-Chiffretext $c = m^e \bmod N$ und ein Entschlüsselungsorakel für alle $c' \neq c$. Zeigen Sie, dass dann der zugrundeliegende Klartext m effizient berechnet werden kann.*

Übung 21 *Gegeben sei ein RSA-Signierorakel, das bei Eingabe $m' \neq m$ die RSA-Signatur von m' zurückliefert. Zeigen Sie, dass man dann effizient die Signatur von m berechnen kann, d.h. man kann RSA-Signaturen* universell *fälschen.*

3.1.1 Brute Force und Meet-in-the-Middle Angriffe auf d

Bei RSA ist die Entschlüsselungsfunktion d_k identisch mit der Signierfunktion sig_k. Nach Korollar 19 ist die Laufzeit zum Berechnen dieser Funktion $\mathcal{O}(\log d \log^2 N)$.

Angenommen, man verwendet in einem RSA-System die Entschlüsselungs-/Signierfunktion d_k wesentlich häufiger als die Verschlüsselungs-/Verifikationsfunktion e_k. Dann wäre es

vorteilhaft, ein kleines d zu wählen, um das Signieren/Entschlüsseln möglichst effizient durchführen zu können. Dies wäre insbesondere dann wünschenswert, wenn d_k auf einem resourcenarmen Gerät (z.B. einer Smartcard) berechnet wird und e_k auf einem schnellen Rechner.

Natürlich darf man d nicht so klein wählen, dass es von einem Angreifer geraten werden kann. Heutzutage geht man davon aus, dass ein Angreifer nicht in der Lage ist, 2^{80} Operationen durchzuführen. D.h. wenn wir d als zufällige 80-Bit Zahl wählen würden, dann wäre dieses d sicher gegen die einfachste Art eines Angriffs; eines *Brute Force Angriffs* (auch Exhaustive Search bzw. Erschöpfende Suche genannt).

Angenommen wir wissen, dass d klein ist. Bei einem Brute Force Angriff durchsucht man den Raum aller möglichen Zahlen von 1 bis d. Für jede Wahl eines Kandidaten d' führt man einen Test durch, ob dieses d' der gesuchte geheime Exponent ist. Bei einem Public-Key Kryptosystem ist der einfachste Test die Äquivalenz $d_k(e_k(m)) = m$.

Brute Force Angriff auf d

EINGABE: N, e

1. Wähle $m \in \mathbb{Z}_N$ zufällig.
2. Berechne $c = m^e \bmod N$
3. FOR $d = 1$ TO N
   a) IF $(c^d = m \bmod N)$ EXIT

AUSGABE: d

Technische Anmerkung: Beim obigen Angriff erhält man $d \bmod \mathrm{ord}(m)$. Man hätte aber gerne $d \bmod \phi(N)$. Dazu verwenden wir einen kleinen technischen Trick. Wir nehmen an, dass $N = pq$ ein Produkt zweier *starker* Primzahlen $p = 2p' + 1$ und $q = 2q' + 1$ mit p', q' prim ist. Dann ist $\phi(N) = (p-1)(q-1) = 4p'q'$. Man kann zeigen, dass ein zufälliges $m \in \mathbb{Z}_N$ mit überwältigender Wahrscheinlichkeit $(1 - \mathrm{poly}(\frac{1}{N}))$ Ordnung $p'q'$ oder $2p'q'$ hat. Ferner kann man zeigen, dass für alle $m \in \mathbb{Z}_N^*$ gilt: $m^{2p'q'} = 1$ (d.h. $\mathbb{Z}_N^*$ ist nicht zyklisch, Übungsaufgabe). Daher kann $d \bmod 2p'q'$ zum Entschlüsseln eingesetzt werden. Wenn wir andererseits $d = x \bmod p'q'$ kennen, dann ist entweder $d = x \bmod 2p'q'$ oder $d = x + p'q' \bmod 2p'q'$. Da wir wissen, dass d ungerade ist, können wir das korrekte $d \bmod 2p'q'$ ermitteln (x falls x ungerade, sonst $x + p'q'$).

Wir wollen im folgenden die $\tilde{\mathcal{O}}$-Notation (weiche $\mathcal{O}$-Notation) verwenden, bei der man polylogarithmische Faktoren vernachlässigt, z.B. ist $\mathcal{O}(N \log^2 N) = \tilde{\mathcal{O}}(N)$.

Satz 22 (Brute Force auf d) *Sei d der geheime RSA-Schlüssel. Dann kann d in Zeit $\tilde{\mathcal{O}}(d)$ und Platz $\tilde{\mathcal{O}}(1)$ bestimmt werden.*

Ein Brute Force Angriff ist für jedes Public-Key Kryptosystem und jedes Signaturschema möglich. Daher müssen die Designer eines Kryptosystems/Signaturschemas als Mindestanforderung an ihr System sicherstellen, dass es eine genügend hohe Komplexität gegenüber Brute Force Angriffen besitzt. In unserem Fall müsste man also zumindest $d \geq 2^{80}$ fordern. Wir werden allerdings sehen, dass diese Schranke bei RSA nicht genügt.

Interessanterweise ist ein Brute Force Angriff, obwohl er so simpel ist, für viele kryptographische Systeme der beste bekannte Angriff. Z.B. ist eine Brute Force Attacke der beste bekannte Angriff auf das noch weitverbreitete symmetrische Kryptosystem DES. DES gilt heutzutage lediglich als unsicher, weil sein Schlüsselraum der Größe 2^{56} schlicht zu klein ist und eine erschöpfende Suche auf diesem Raum mit Hilfe heutiger Rechner kein Problem mehr darstellt.

Nun begnügen sich aber Angreifer im allgemeinen nicht mit einem Brute Force Angriff. Das folgende Angriffsschema wird in der Literatur als *Meet-in-the-Middle* Angriff bezeichnet (Vorsicht: Es gibt bei kryptographischen Protokollen einen anderen Angriff, der auch Meet-in-the-Middle Angriff genannt wird). Der Name Meet-in-the-Middle kommt daher, dass man den Suchraum in zwei gleichgroße Hälften aufteilt. Auch hier benötigt man natürlich wieder einen Test, um zu entscheiden, ob ein Schlüsselkandidat der gesuchte geheime Schlüssel ist. Schematisch kann man einen Meet-in-the-Middle Angriff wie folgt beschreiben:

- Teile den gesuchten Schlüssel in zwei gleich große Hälften auf.
- Führe eine Brute Force Suche auf der ersten Schlüsselhälfte durch. Berechne für jeden Kandidaten einen Test und speichere alle Ergebnisse in einer Liste L. Eventuell muss L sortiert werden.
- Führe eine Brute Force Suche auf der zweiten Schlüsselhälfte durch. Berechne für jeden Kandidaten einen Test und prüfe, ob das Testergebnis mit einem Element aus L übereinstimmt (z.B. mittels binärer Suche). Falls ja, gib die beiden Schlüsselhälften aus.

Wir wollen nun einen Meet-in-the-Middle Angriff auf den geheimen RSA-Schlüssel durchführen. Dazu nehmen wir an, wir wissen dass $d \leq B$ gilt und definieren $A = \lceil \sqrt{B} \rceil$. Unser Test beim Brute Force Angriff war die Identität $c^d = (m^e)^d = m \bmod N$. Wir teilen d in der Mitte auf und schreiben $d = d_0 A + d_1$ mit $d_0, d_1 < A$. Es gilt

$$c^d = c^{d_0 A + d_1} = (c^A)^{d_0} \cdot c^{d_1} = m \bmod N.$$

Wir können daher testen, ob $(c^A)^{d_0} = m(c^{-1})^{d_1} \bmod N$ gilt. Diesen Test verwenden wir in unserem Meet-in-the-Middle Angriff.

Meet-in-the-Middle Angriff auf d

EINGABE: N, e, B

1. Wähle $m \in \mathbb{Z}_N$ zufällig und setze $c = m^e \bmod N$.
2. Setze $A = \lceil \sqrt{B} \rceil$.
3. FOR $d_0 = 0$ TO $A - 1$
   a) Speichere $(d_0, (c^A)^{d_0} \bmod N)$ in einer nach der zweiten Komponente sortierten Liste.
4. FOR $d_1 = 0$ TO $A - 1$
   a) Schaue mit Hilfe binärer Suche, ob in L ein Wert $(d_0, m(c^{-1})^{d_1} \bmod N)$ auftaucht. Falls ja, EXIT.

AUSGABE: $d = d_0 A + d_1$

Satz 23 (Meet-in-the-Middle Angriff auf d) *Sei d der geheime RSA-Schlüssel. Dann kann d in Zeit $\tilde{\mathcal{O}}(\sqrt{d})$ mit Speicherbedarf $\tilde{\mathcal{O}}(\sqrt{d})$ bestimmt werden.*

Wir haben hier zwischen dem Brute Force Angriff und der Meet-in-the-Middle Attacke auf d einen sogenannten *Time-Memory* Tradeoff. D.h. wir können zwar die Laufzeit von $\tilde{\mathcal{O}}(d)$ auf $\tilde{\mathcal{O}}(\sqrt{d})$ reduzieren, gleichzeitig steigt aber der Speicherbedarf von $\tilde{\mathcal{O}}(1)$ auf $\tilde{\mathcal{O}}(\sqrt{d})$. Wir werden später Angriffe auf den diskreten Logarithmus kennenlernen, die eine Laufzeit von der Größenordnung der Wurzel des Suchraums haben aber nur Speicherplatz $\tilde{\mathcal{O}}(1)$ benötigen. Ob es einen solchen Angriff auf kleines d gibt, ist ein offenes Problem.

Bei Meet-in-the-Middle Angriffen muss man stets eine geeignete Aufteilung des Geheimnisses und einen geeigneten Test für die zwei Schlüsselhälften finden. Beide Probleme können nicht-triviale Lösungen erfordern. Wir werden im nächsten Abschnitt einen Angriff auf kleine CRT-Exponenten bei RSA betrachten, bei dem der Test einen algorithmischen Trick zum gleichzeitigen Auswerten von Polynomen an mehreren Punkten erfordert. Danach betrachten wir einen Angriff von Boneh, Joux und Nguyen auf kleine Klartexte m, bei denen die Aufteilung nicht mehr additiv sondern multiplikativ erfolgt.

3.1.2 Meet-in-the-Middle Angriff auf kleine CRT-Exponenten

Quisquater und Couvreur schlugen die folgende Methode zur schnellen Berechnung der RSA-Entschlüsselungs-/Signierfunktion vor.

RSA-Entschlüsselung mittels Chinesischem Restsatz

EINGABE: N, d, c

1. Berechne $d_p = d \bmod p-1$ und $d_q = d \bmod q-1$.
2. Berechne $c^{d_p} \bmod p$ und $c^{d_q} \bmod q$.
3. Setze die beiden Teillösungen mit Hilfe des Chinesischen Restsatzes zusammen.

AUSGABE: $c^d \bmod N$

Man beachte, dass im zweiten Schritt der Quisquater-Couvreur Methode $c^d \bmod p$ und $c^d \bmod q$ berechnet werden. Da aber die Gruppenordnungen der multiplikativen Gruppen $\mathbb{Z}_p^*$ und $\mathbb{Z}_q^*$ die Größen $p-1$ und $q-1$ haben, können wir d modulo dieser Gruppenordnung reduzieren.

Nehmen wir an, d hat dieselbe Bitlänge wie $N = pq$, und p, q haben gleiche Bitgröße. Vergleichen wir nun die direkte Berechnung von $c^d \bmod N$ mit der Quisquater-Couvreur Methode. Nach Satz 19 benötigt man zur Berechnung von $c^d \bmod N$ Laufzeit $\mathcal{O}(\log d \log^2 N)$. Bei der Berechnung im zweiten Schritt haben sowohl d als auch die Primzahlen p und q nur noch die halbe Bitgröße von N. Jede der Berechnungen ist also etwa 8-mal so schnell wie die direkte Berechnung $c^d \bmod N$. Da die Laufzeit von Schritt 1 & 3 im Vergleich zum zweiten Schritt vernachlässigt werden kann, ist die Quisquater-Couvreur Methode insgesamt etwa um den Faktor 4 schneller. Aus diesem Grund findet die Methode in der Praxis häufig Verwendung und ist bei resourcenarmen Chips, wie man sie z.B. auf Smartcards findet, Standard.

Offenbar kann man die Quisqater-Couvreur Methode noch weiter beschleunigen, wenn man kleine Exponenten $d_p = d \bmod p-1$ und $d_q = d \bmod q-1$ wählt. Solche Exponenten nennt man kleine CRT-Exponenten (CRT = Chinese Remainder Theorem). Man verwendet für die Quisquater-Couvreur Methode in der Literatur auch die Bezeichnung CRT-RSA.

Überlegen wir uns nun, wie man kleine CRT-Exponenten erzeugt. Angenommen wir wählen $d_p, d_q \leq B$. Nun müssten wir $d_p \bmod p-1$ und $d_q \bmod q-1$ mit dem Chinesischen Restsatz zu $d \bmod \phi(N)$ zusammensetzen und aus d den öffentlichen Schlüssel e generieren. Das Problem ist jedoch, dass $p-1$ und $q-1$ nicht teilerfremd sind. Daher konstruieren wir p und q so, dass $\gcd(p-1, q-1) = 2$ und verwenden folgenden Trick.

Generierung kleiner CRT-Exponenten

EINGABE: B, p, q mit $\gcd(p-1, q-1) = 2$

1. Wähle zufällige $d_p, d_q \leq B$ mit $d_p = d_q = 1 \bmod 2$
2. Berechne mit dem Chinesischen Restsatz d' mod $\frac{(p-1)(q-1)}{4}$ mit
$$d' = \frac{d_p - 1}{2} \bmod \frac{p-1}{2} \text{ und } d' = \frac{d_q - 1}{2} \bmod \frac{q-1}{2}$$
3. Setze $d = 2d' + 1$ und berechne mit dem EEA $e = d^{-1} \bmod \phi(N)$.

AUSGABE: e, d_p, d_q

Man beachte, dass die Gleichung im zweiten Schritt wohldefiniert sind, da $d_p - 1$ und $d_q - 1$ gerade sind. Weiterhin gilt

$$d = 2d' + 1 = d_p \bmod p - 1 \text{ und } d = 2d' + 1 = d_q \bmod q - 1$$

Daraus folgt die Korrektheit des Algorithmus.

Nun wollen wir uns wieder in die Perspektive eines Angreifers versetzen und überlegen, wie man RSA mit kleinen CRT-Exponenten attackieren kann. Die Entwicklung einer Brute-Force Attacke ist eine Übungsaufgabe:

Übung 24 *Sei (N, e) ein öffentlicher RSA Schlüssel mit zugehörigen CRT-Exponenten $d_p \neq d_q$. Dann kann die Faktorisierung von N in Zeit $\tilde{\mathcal{O}}(\min\{d_p, d_q\})$ und Platz $\tilde{\mathcal{O}}(1)$ berechnet werden.*

Wir wollen uns nun einen Meet-in-the-Middle auf kleine CRT-Exponenten überlegen. Dazu gehen wir zunächst analog zum Meet-in-the-Middle Angriff auf d in Abschnitt 3.1.1 vor.

Wir nehmen oBdA an, dass $d_p = \min\{d_p, d_q\} \leq B$. Weiterhin können wir annehmen, dass $d_p \neq d_q$ (Was passiert sonst? Kleine Übungsaufgabe). Setze $A = \lceil \sqrt{B} \rceil$ und teile d_p auf in $d_p = d_0 A + d_1$ mit $d_0, d_1 < A$. Wir machen nun eine erschöpfende Suche über alle Kandidaten für d_0 und d_1. Wir wissen, dass d_p die Identität $m^{ed_p} = m \bmod p$ für ein beliebiges $m \in \mathbb{Z}_N$ erfüllt. Da wir p nicht kennen, können wir diese Identität nicht direkt überprüfen. Allerdings können wir testen, ob $\gcd(m^{ed_p} - m, N) = p$ gilt. (Es ist $\gcd(m^{ed_p} - m) \neq q$ mit großer Wahrscheinlickeit über die zufällige Wahl von m.)

Analog zu Abschnitt 3.1.1 schreiben wir

$$m^{ed_p} = m^{e(d_0 A + d_1)} = (m^{eA})^{d_0} \cdot (m^e)^{d_1} = m \bmod p$$

Wir können wieder eine Liste L mit allen möglichen Kandidaten $(d_0, (m^{eA})^{d_0} \bmod N)$ erstellen. Dann testen wir im nächsten Schritt, ob $m(m^{-e})^{d_1} \bmod N$ der zweiten Komponente eines Kandidaten aus L modulo p entspricht. Das Problem hier ist aber, dass wir die Identität modulo p überprüfen müssen. D.h. eine Sortierung von L nach der zweiten Komponente nutzt uns hier nichts. Wir müssen für jeden Kandidaten für d_1 für jeden Eintrag in L prüfen, ob $\gcd((m^{eA})^{d_0} \bmod N - m(m^{-e})^{d_1} \bmod N, N) = p$ gilt.

Damit müssen wir für jeweils $\mathcal{O}(\sqrt{d_p})$ Kandidaten für d_1 jeweils $\mathcal{O}(\sqrt{d_p})$ Listeneinträge getestet werden. D.h. unsere Gesamtlaufzeit ist $\tilde{\mathcal{O}}(d_p)$, und wir haben nichts gegenüber dem Brute Force Angriff gewonnen.

Wir verwenden nun den folgenden Trick. Zunächst berechnen wir wieder $m^{eAi} \bmod N$ für alle möglichen Kandidaten $i = 0, \ldots, A-1$, die für d_0 in Frage kommen. Für die d_1-Kandidaten führen wir eine Variable x ein. Danach multiplizieren wir die einzelnen Terme auf:

$$g(x) = \prod_{i=0}^{A-1} (m^{eAi} x - m) \bmod N.$$

Man beachte, dass $g(x)$ ein Polynom vom Grad A ist. Zudem wissen wir, dass m^{ed_1} eine Nullstelle des Polynoms modulo p ist:

$$g(m^{ed_1}) = 0 \bmod p,$$

denn für $i = d_0$ wird der i-te Faktor kongruent 0 modulo p und damit das ganze Produkt. Daher werten wir nun $g(x)$ an den Stellen $(m^e)^j \bmod N$ für $j = 0, \ldots, A-1$ aus. Wenn wir mit Hilfe des Horner-Schemas ein Polynom vom Grad A an A Stellen auswerten, benötigen wir Zeit $\Theta(A^2) = \Theta(d_p)$.

Heißt das, wir haben schon wieder nichts gegenüber der Brute-Force Methode gewonnen? Nein, denn glückerlicherweise kann man ein Polynom effizient simultan an mehreren Stellen auswerten. Dazu benutzen wir den folgenden Satz, der in den Übungen mit Hilfe einer Divide & Conquer Strategie bewiesen wird.

Satz 25 *Sei $p(x)$ ein Polynom vom Grad n. Dann kann $p(x)$ an n Stellen in Zeit $n \log^2 n$ ausgewertet werten, wobei der Speicherbedarf $\tilde{\mathcal{O}}(n)$ ist.*

Zusammenfassend erhaltend wir die folgende Meet-in-the-Middle Attacke.

Meet-in-the-Middle Angriff auf kleine CRT-Exponenten

EINGABE: N, e, B mit $d_p \leq B$

1. Wähle $m \in \mathbb{Z}_N$ zufällig und setze $c := m^e \bmod N$.
2. Setze $A := \lceil \sqrt{B} \rceil$.
3. Berechne $g(x) = \prod_{i=0}^{A-1}(c^{Ai}x - m) \bmod N$.
4. Werte $g(x)$ an den Stellen c^j für $j = 0, \ldots, A-1$ aus.
5. FOR $j = 0$ TO $A - 1$
   a) Falls $\gcd(g(c^j), N) = p$, EXIT.

AUSGABE: p, $q = \frac{N}{p}$

Satz 26 *Sei (N, e) ein öffentlicher Schlüssel mit zugehörigen geheimen CRT-Exponenten d_p und d_q. Dann kann die Faktorisierung von N in Zeit $\tilde{\mathcal{O}}(\min(\sqrt{d_p}, \sqrt{d_q}))$ mit Speicherbedarf $\tilde{\mathcal{O}}(\min(\sqrt{d_p}, \sqrt{d_q}))$ berechnet werden.*

3.1.3 Meet-in-the-Middle Attacke auf m

Angenommen Alice und Bob wollen Nachrichten sicher austauschen, aber aus Effizienzgründen nicht für jede Nachricht ein Public-Key Verfahren verwenden. Dazu schickt Alice an Bob einen geheimen 128-Bit Sitzungsschlüssel m und verschlüsselt m mit Bob's öffentlichem RSA-Schlüssel. Zuvor hängt Alice an die Nachricht noch eine Signatur, um Integrität und Authentizität zu gewährleisten. Nun können Alice und Bob den Sitzungsschlüssel verwenden, um Nachrichten mittels eines schnellen symmetrischen Verfahrens wie AES auszutauschen.

Man beachte, dass die verschlüsselte Nachricht im obigen Szenario deutlich kürzer ist als ein RSA-Modul, der heutzutage üblicherweise mindestens 1024 Bit hat. Dies eröffnet Angriffsmöglichkeiten. Offensichtlich ist ein Brute-Force Angriff auf m. In den Übungen werden wir einen weiteren einfachen Angriff für RSA mit kleinem e kennenlernen.

Übung 27 *Sei $c = m^e \bmod N$ ein RSA-Chiffretext. Zeige, dass m leicht aus c berechnet werden kann, falls $m < N^{1/e}$.*

Später werden wir mit Hilfe von Gitterangriffen eine stärke Variante dieses Angriffs kennenlernen. Wir werden dazu zeigen: Wenn man schon einen Teil von m kennt und der unbekannte Teil von m höchstens $N^{1/e}$ groß ist, dann kann man den unbekannten

Teil effizient berechnen. Dies ist eine inhomogene Variante des Angriffs aus Übung 27; bei dieser neuen Variante sind die obersten Bits von m nicht Null sondern bekannt.

Nun wollen wir aber zunächst untersuchen, ob es einen Meet-in-the-Middle Angriff gibt, der uns m in Zeit $\tilde{\mathcal{O}}(m)$ liefert. Ein solcher Angriff wurde 2000 von Boneh, Joux und Nguyen vorgestellt. Wir wollen den Angriff hier nicht im Detail analysieren, sondern nur die Idee präsentieren.

Der Unterschied dieser Attacke im Vergleich zu den vorigen wird sein, dass sie nur mit einer gewissen Wahrscheinlichkeit funktioniert, die von der Wahl von m abhängt. Wir verwenden hier statt einer additiven Aufteilung von m eine multiplikative Aufteilung. Beginnen wir mit einem abgefangenen Chiffretext $c = m^e \bmod N$, zu dem wir m bestimmen wollen. Nun versuchen wir, m als Produkt zweier etwa gleichgroßer Faktoren zu schreiben: $m = m_0 \cdot m_1$ mit $m_0, m_1 \approx \sqrt{m}$.

Boneh, Joux und Nguyen geben in ihrer Arbeit Wahrscheinlichkeit an, wann m als Produkt zweier etwa gleich großer Faktoren geschrieben werden kann. Wir geben hier drei Beispiele:

Wenn m eine 40-Bit Zahl ist, dann kann sie mit einer Wahrscheinlichkeit von 18% als Produkt zweier 20-Bit Zahlen geschrieben werden. Mit einer Wahrscheinlichkeit von 32% kann sie als Produkt $m_0 m_1$ geschrieben werden, wobei $m_0 m_1$ höchstens 21-Bit Zahlen sind. Sind 22-Bit Zahlen zugelassen, so erhöht sich die Wahrscheinlichkeit auf 39%.

Der Angriff selbst ist analog zum Angriff auf kleines d. Unser Test besteht aus der Identität

$$m_0^e = c m_1^{-e} \bmod N$$

Meet-in-the-Middle Angriff auf kleines m

EINGABE: N, e, c mit $c = m^e \bmod N$, $m \leq B$

1. Setze $A := 2\lceil\sqrt{B}\rceil$.
2. Berechne $i^e \bmod N$ für $i = 0, \ldots, A-1$ und speichere $(i, i^e \bmod N)$ in einer Liste L. Sortiere L nach der zweiten Komponente.
3. FOR $j = 0$ TO $A - 1$
   a) Binäre Suche: Falls es in L ein Element $(i, cj^{-e} \bmod N)$ gibt, EXIT.
4. Entweder setze A := 2A und gehe zu Schritt 2 oder Ausgabe "Angriff nicht erfolgreich".

AUSGABE: $m = i \cdot j$

Der Angriff von Boneh, Joux und Nguyen zeigt, dass man aus dem Chiffretext $c =$

m^e mod N mit signifikanter Wahrscheinlichkeit (über die zufällige Wahl von m) den zugrundeliegenden Klartext m in Zeit $\tilde{\mathcal{O}}(\sqrt{m})$ mit Speicherplatzbedarf $\tilde{\mathcal{O}}(\sqrt{m})$ bestimmen kann.

4 ElGamal & Pollard's Rho-Methode

4.1 Der diskrete Logarithmus und das ElGamal Kryptosystem

Definition 28 (DL-Problem) *Sei G eine multiplikative endliche Gruppe und $\alpha \in G$. Sei β in der von α erzeugten Untergruppe, d.h. $\beta \in \langle\alpha\rangle$. Gesucht ist das eindeutige $a \bmod ord(\alpha)$ mit $\alpha^a = \beta$. Wir verwenden die Notation $a = dlog_\alpha(\beta)$. Das diskrete Logarithmusproblem bezeichnen wir auch abkürzend als* DL-Problem.

Man beachte, dass das diskrete Logarithmusproblem nicht in allen Gruppen schwer ist. Z.B. ist in der additiven Gruppe $(\mathbb{Z}_n, +)$ mit Generator $\alpha = 1$ das diskrete Logarithmusproblem trivial: $\mathrm{dlog}_1\beta = \beta$.

Wir nehmen an, dass das DL-Problem in der Gruppe $\mathbb{Z}_p^*$, p prim, schwer ist, sofern $\mathrm{ord}(\alpha)$ groß genug ist. (Anmerkung: $\mathrm{ord}(\alpha)$ darf nicht in kleine Primfaktoren zerfallen, sonst kann man den sogenannten Pohlig-Hellman Algorithmus verwenden). D.h. wir nehmen an, dass es keinen effizienten Algorithmus gibt, der bei Eingabe α, β den Wert $a = \mathrm{dlog}_\alpha(\beta)$ berechnet.

Das ElGamal Kryptosystem basiert auf der Schwierigkeit des DL-Problems in $\mathbb{Z}_p^*$. Es wurde 1984 von Taher El Gamal vorgestellt.

ElGamal Kryptosystem

Sei p eine Primzahl (512 Bit), wobei $p-1$ einen großen Primfaktor hat (> 160 Bit). Ferner sei α ein Generator von $\mathbb{Z}_p^*$.
Wir definieren

(1) $\mathcal{P} = \mathbb{Z}_p$

(2) $\mathcal{C} = \mathbb{Z}_p^* \times Z_p$

(3) $\mathcal{K} = \{(p, \alpha, \beta, a) \mid \beta = \alpha^a \bmod p, a \in \mathbb{Z}_{p-1}\}$, wobei (p, α, β) der öffentliche Teil des Schlüssels ist und a der geheime Schlüssel.

(4) Verschlüsselungsfunktion: $e_k(m) = (\alpha^r \bmod p, m \cdot \beta^r \bmod p)$ mit $r \in \mathbb{Z}_{p-1}$, Entschlüsselungsfunktion: $d_k(\gamma, \delta) = \gamma^{-a}\delta \bmod p$.

Korrektheit: Wir müssen zeigen, dass $d_k(e_k(m)) = m$ für alle $m \in \mathcal{P}$:

$$d_k(e_k(m)) = d_k(\alpha^r, m\beta^r) = (\alpha^r)^{-a} m\beta^r = \alpha^{-ar+ar} m = m \bmod p.$$

Nach Satz 18 kann e_k in Zeit $\mathcal{O}(\log r \log^2 p) = \mathcal{O}(\log^3 p)$ und d_k in Zeit $\mathcal{O}(\log a \log^2 p) = \mathcal{O}(\log^3 p)$ berechnet werden.

Ein Nachteil des ElGamal Kryptosystems ist eine Nachrichtenexpansion um den Faktor 2, d.h. dass Chiffretexte doppelt so lange sind wie die zugrundeliegenden Klartexte.

Wenn man das DL-Problem in $\mathbb{Z}_p^*$ effizient lösen kann, dann kann man aus dem öffentlichen Parameter $\beta = \alpha^a$ das geheime a ermitteln. Es ist als Übungsaufgabe zu zeigen, dass ein Angreifer zum Entschlüsseln eines ElGamal Chiffretextes aus den Parametern α^a und α^r den Wert α^{ar} berechnen muss. Das Problem, α^{ar} aus (α^a, α^r) zu berechnen, bezeichnet man als *Diffie-Hellman Problem*. Es ist bekannt, dass man das Diffie-Hellman Problem lösen kann, wenn man das DL-Problem lösen kann. Man vermutet, dass die Umkehrung ebenfalls gilt (man kann die Umkehrung bisher nur für Gruppen mit besonderen Eigenschaften zeigen).

Übung 29 *Die Notation $A \Rightarrow B$ bedeutet in dieser Übung, dass es einen effizienten Algorithmus für B gibt, wenn es einen effizienten Algorithmus für A gibt. Zeigen Sie:*

$$\textit{ElGamal Chiffretexte entschlüsseln} \Leftrightarrow \textit{Diffie-Hellman Problem} \Leftarrow \textit{DL-Problem}$$

4.2 Das ElGamal Signaturverfahren

Von ElGamal wurde zusätzlich zum Public-Key Kryptosystem noch ein Signaturverfahren vorgeschlagen.

ElGamal Signaturverfahren

Sei p eine Primzahl (512 Bit), wobei $p-1$ einen großen Primfaktor hat (> 160 Bit). Ferner sei α ein Generator von $\mathbb{Z}_p^*$.
Wir definieren

(1) $\mathcal{P} = \mathbb{Z}_{p-1}$

(2) $\mathcal{U} = \mathbb{Z}_p^* \times Z_{p-1}$

(3) $\mathcal{K} = \{(p, \alpha, \beta, a) \mid \beta = \alpha^a \bmod p, a \in \mathbb{Z}_{p-1}\}$, wobei (p, α, β) der öffentliche Teil des Schlüssels ist und a der geheime Schlüssel.

(4) Signierfunktion: $\text{sig}_k(x) = (\gamma, \delta) = (\alpha^r \bmod p, r^{-1}(x - a\gamma) \bmod p - 1)$ mit $r \in \mathbb{Z}_{p-1}^*$,

Verifikationsfunktion: $\text{ver}_k(x, (\gamma, \delta)) = \begin{cases} \text{wahr} & \text{für } \beta^\gamma \gamma^\delta = \alpha^x \bmod p \\ \text{falsch} & \text{sonst} \end{cases}$

Korrektheit: Wir zeigen $\text{ver}_k(x, y) = \text{ wahr } \Leftrightarrow y = \text{sig}_k(x)$:

$$\mathrm{ver}_k(x,y) = \text{ wahr } \Leftrightarrow \beta^\gamma \gamma^\delta = \alpha^x \Leftrightarrow \alpha^{a\gamma}\alpha^{rr^{-1}(x-a\gamma)} = \alpha^{a\gamma - a\gamma + x} = \alpha^x \bmod p$$

Mit Satz 18 können sig_k und ver_k in Zeit $\mathcal{O}(\log^3 p)$ berechnet werden.

Ebenso wie das ElGamal Kryptosystem basiert das ElGamal Unterschriftenverfahren auf der Schwierigkeit des DL-Problems.

Wir sehen, dass der Nachrichtenraum bei ElGamal Unterschriften $\mathcal{P} = \mathbb{Z}_{p-1}$ ist. Erinnern wir uns, dass er bei RSA $\mathcal{P} = \mathbb{Z}_N$ ist. Was macht man nun wenn man eine Unterschrift zu einer beliebig langen Nachricht $x \in \{0,1\}^*$ leisten will? Dann verwendet man Hashfunktionen, die beliebig lange Bitstrings auf den Nachrichtenraum abbilden, d.h. also $h : \{0,1\}^* \to \mathbb{Z}_{p-1}$ bzw. $h' : \{0,1\}^* \to \mathbb{Z}_N$. Diese Hashfunktionen sollten kollisionsresistent sein, d.h. es sollte nicht effizient möglich sein $m \neq m'$ mit $h(m) = h(m')$ zu berechnen.

Die Verwendung einer Hashfunktion, die Nachrichten beliebiger Länge auf den Klartextraum abbildet, ist Standard in der modernen Kryptographie. Daher ist die einzige Anforderung an den Nachrichtenraum eines Unterschriftenverfahrens, dass er groß genug ist, um die Kollisionsresistenz einer Hashfunktion zu gewährleisten.

4.3 DSA Unterschriften

ElGamal und RSA Unterschriften sind relativ lang (beide etwa 1024 Bit). Das DSA Verfahren (DSA=Digital Signature Algorithm) ist eine Variante des ElGamal-Verfahrens, das eine Unterschriftenlänge von nur 320 Bit erlaubt. DSA ist im wesentlichen eine kleine Modifikation eines von Schnorr 1989 entwickelten Unterschriftenverfahrens. Es ist heutzutage das Standardverfahren für Unterschriften.

DSA Signaturverfahren

Sei p eine Primzahl (512 Bit), wobei $p-1$ einen großen Primfaktor q hat (160 Bit). Ferner sei $\alpha \in Z_p^*$ mit $\text{ord}(\alpha) = q$.
Wir definieren

(1) $\mathcal{P} = \mathbb{Z}_q$

(2) $\mathcal{U} = \mathbb{Z}_q^* \times Z_q^*$

(3) $\mathcal{K} = \{(p, \alpha, \beta, a) \mid \beta = \alpha^a \bmod p, a \in \mathbb{Z}_q\}$, wobei (p, α, β) der öffentliche Teil des Schlüssels ist und a der geheime Schlüssel.

(4) Signierfunktion: $\text{sig}_k(x) = (\gamma, \delta) = ((\alpha^r \bmod p) \bmod q, r^{-1}(x + a\gamma) \bmod q)$ mit $r \in \mathbb{Z}_q^*$,
Verifikationsfunktion:

$$\text{ver}_k(x, (\gamma, \delta)) = \begin{cases} \text{wahr} & \text{für } ((\alpha^x \beta^\gamma)^{\delta^{-1} \bmod q} \bmod p) \bmod q = \gamma \bmod q \\ \text{falsch} & \text{sonst} \end{cases}$$

Korrektheit: Aus der Formel für δ erhalten wir die Identität $r = \delta^{-1}(x + a\gamma) \bmod q$. Damit gilt:

$$\begin{aligned} \text{ver}_k(x, (\gamma, \delta)) = \text{ wahr} \quad \Leftrightarrow \quad & ((\alpha^x \beta^\gamma)^{\delta^{-1} \bmod q} \bmod p) \bmod q = (\alpha^{\delta^{-1}x} \alpha^{\delta^{-1}a\gamma} \bmod p) \bmod q \\ & = (\alpha^{\delta^{-1}(x+a\gamma)} \bmod p) \bmod q = (\alpha^r \bmod p) \bmod q = \gamma. \end{aligned}$$

Eine Anwendung von Satz 18 zeigt, das die Signierfunktion und die Verifikationsfunktion in Zeit $\mathcal{O}(\log q \log^2 p)$ berechenbar sind.

Die Sicherheit des ElGamal Public Kryptosystems, des ElGamal Signaturverfahrens und des DSA Signaturverfahrens beruht auf der Annahme, dass aus dem öffentlichen Schlüssel (p, α, β) nicht der geheime Schlüssel $a = \text{dlog}_\alpha \beta$ berechnet werden kann. Wir wollen uns nun Algorithmen betrachten, die das DL-Problem lösen. Ein Brute-Force Angriff löst das Problem in Zeit $\tilde{\mathcal{O}}(a)$ mit Speicherplatz $\tilde{\mathcal{O}}(1)$ (Übungsaufgabe).

Wie in Kapitel 3 können wir auch hier wieder Time-Memory Tradeoffs verwenden, um die Laufzeit des Algorithmus auf $\tilde{\mathcal{O}}(\sqrt{a})$ zu drücken.

Übung 30 *Konstruieren Sie einen Algorithmus, der bei Eingabe (p, α, β) die Ausgabe $a = dlog_\alpha \beta$ bezüglich der Gruppe $\mathbb{Z}_p^*$ in Zeit $\tilde{\mathcal{O}}(\sqrt{a})$ mit Speicherplatz $\tilde{\theta}(\sqrt{a})$ liefert.*

Da a modulo $\text{ord}(\alpha)$ definiert ist, hat der obige Algorithmus Laufzeit $\tilde{\mathcal{O}}(\sqrt{\text{ord}(\alpha)})$. Wir werden nun einen Algorithmus kennenlernen, der den diskreten Logarithmus ebenfalls in Zeit $\tilde{\mathcal{O}}(\sqrt{\text{ord}(\alpha)})$ aber mit konstantem Speicherplatz berechnet.

4.4 Pollard's Rho-Methode

Gegeben sei (p, α, β), gesucht ist $a = \text{dlog}_\alpha \beta \bmod \text{ord}(\alpha)$ bezüglich der Gruppe $\mathbb{Z}_p^*$. Dabei nehmen wir an, dass die Ordnung von α gegeben ist, sei $q = \text{ord}(\alpha)$.

Die Methode von Pollard basiert auf der folgenden Überlegung. Angenommen man findet Tupel (x_1, y_1) und (x_2, y_2) mit der Eigenschaft

$$\alpha^{x_1}\beta^{y_1} = \alpha^{x_2}\beta^{y_2} \bmod p. \tag{4.1}$$

Dann gilt $\beta^{y_1-y_2} = \alpha^{x_2-x_1} \bmod p$. Nehmen wir an, dass $y_1 - y_2$ modulo der Ordnung von α invertierbar ist. Daraus folgt

$$\beta = \alpha^{\frac{x_2-x_1}{y_1-y_2}} \bmod p \quad \Leftrightarrow \quad \text{dlog}_\alpha \beta = \frac{x_2 - x_1}{y_1 - y_2} \bmod q.$$

Wir müssen eine Identität der Form $\alpha^{x_1}\beta^{y_1} = \alpha^{x_2}\beta^{y_2} \bmod p$ konstruieren. Wenn $x_1, x_2 \in_R \mathbb{Z}_q$, dann stehen auf der rechten und linken Seite der Identität zufällige Elemente aus der von α erzeugten Untergruppe der Ordnung q.

Angenommen wir erzeugen zufällige Elemente aus einer Menge der Größe q. Da die Menge endlich ist, müssen wir irgendwann ein Element zum zweiten Mal erzeugen. Sei $S = (s_1, s_2, \ldots, s_n)$ die Sequenz der von uns erzeugten Elemente. Wir nennen Elemente s_i, s_j mit $s_i = s_j$, $i \neq j$ eine Kollision. Wie groß muss unsere Sequenz sein, damit wir eine Kollision erhalten? Da wir Elemente aus einer Menge der Größe q zufällig erzeugen, gilt für alle i, j:

$$\Pr(s_i = s_j) = \frac{1}{q}.$$

Sei die Zufallsvariable $X_{i,j}$ wie folgt definiert:

$$X_{i,j} = \begin{cases} 1 & \text{für } s_i = s_j \\ 0 & \text{sonst} \end{cases}$$

Dann ist die erwartete Anzahl der Kollisionen gerade:

$$E(\#\text{Kollisionen}) = \sum_{1 \leq i < j \leq n} \Pr(X_{i,j} = 1) = \sum_{1 \leq i < j \leq n} \Pr(s_i = s_j) = \frac{\binom{n}{2}}{q} = \Theta\left(\frac{n^2}{q}\right).$$

Damit unsere Sequenz also eine konstante erwartete Anzahl von Kollisionen enthält, muss man $n = \Theta(\sqrt{q})$ Elemente zufällig erzeugen. Dieses Resultat nennt man auch das *Geburtstagsparadoxon* (wenn mehr als $\sqrt{365}$ Leute im Raum sitzen, dann kann man erwarten, dass mindestens zwei am selben Tag Geburtstag haben).

Angenommen wir erzeugen zufällige Elemente $s_i = \alpha^{x_i}\beta^{y_i} \bmod p$ aus der von α generierten Gruppe. Zusätzlich zu diesen Elementen speichern wir uns noch die Exponenten x_i und y_i. Wenn wir $\Theta(\sqrt{q})$ Elemente generiert haben, dann erwarten wir eine Kollision,

d.h. eine Identität wie in Gleichung (4.1). Mit Hilfe der gespeicherten x_i und y_i können wir das DL-Problem wie zuvor beschrieben lösen.

Angenommen wir gehen bei unserem Angriff wie folgt vor: Wir generieren die Sequenz S sukzessive und schauen für jedes neu erzeugte Element, ob Identität (4.1) gilt. Damit müssen wir $\mathcal{O}(\sqrt{q})$ Elemente speichern und $\mathcal{O}(q)$ Identitäten prüfen. Ein solcher Angriff ist schlechter als eine Brute-Force Attacke. D.h. wir möchten sowohl verhindern, jedes neu generierte Element mit allen Elementen der Sequenz zu vergleichen, als auch alle Elemente der Sequenz zu speichern.

Wir wollen nun einen eleganten Trick verwenden, um die obigen Probleme mit Hilfe von *Random Walks* zu lösen. Sei $G = \langle\alpha\rangle$ die von α erzeugte Untergruppe der Größe q. Angenommen wir haben eine zufällige Abbildung $f : G \to G$, d.h. wir wählen f zufällig aus der Menge aller Abbildungen $G \to G$.

Wir starten mit einem zufälligen Wert $s_0 \in G$ und berechnen die Sequenz

$$s_i = f(s_i) \quad \text{für } i \geq 0.$$

Da G endlich ist, erhalten wir irgendwann ein Element doppelt. Angenommen s_i und s_j seien unsere erste Kollision. Da f eine deterministische Funktion ist, gilt dann ebenfalls

$$s_{i+1} = f(s_i) = f(s_j) = s_{j+1} \ , \quad s_{i+2} = s_{j+2} \ , \quad \text{usw.}$$

D.h. aber wir haben einen Kreis der Länge $j - i$. Durch die Verwendung einer deterministischen Funktion f haben wir immer dann eine Kollision, wenn man von einem Punkt des Kreises startet und einmal den Kreis durchläuft. Die entstehende Sequenz sieht aus wie der griechische Buchstabe ρ, daher kommt der Name *Rho-Methode*. Wir nennen die Sequenz $s_1, s_2, \ldots, s_i$ das Anfangsstück und die Sequenz $s_i, s_{i+1}, \ldots, s_{j-1}, s_i$ den Kreis.

Wie findet man nun eine Kollision? Dazu verwendet man einen Algorithmus von Floyd zum Finden von Kreisen. Angenommen, wir lassen zwei Känguruhs im Punkt s_0 starten. Känguruh 1 ist langsam und springt jeweils einen Schritt weit, d.h. es durchläuft die Sequenz $s_1, s_2, s_3, \ldots$ und Känguruh 2 ist flink und springt jeweils zwei Schritte $s_2, s_4, s_6, \ldots$. Die Frage ist: Wann landen die beiden auf dem gleichen Punkt?

Im Anfangsstück können sich die Känguruhs nicht treffen. Sei nun Känguruh 1 beim Punkt s_i. Falls Känguruh 2 ebenfalls in s_i ist, dann sind wir fertig. Ansonsten laufen beide Känguruhs im Kreis der Länge $j - i$. In jedem Schritt verringert das schnelle Känguruh seinen Rückstand auf das langsame Känguruh um 1. Da sein Abstand zu Beginn kleiner als die Kreislänge gewesen ist, hat es das langsame Känguruh nach weniger als der Gesamtzahl von $i + (j - i)$ Schritten eingeholt.

D.h. wir finden eine Kollision nach weniger als j Schritten. Wie wir bereits zuvor gesehen haben, ist $j = \Theta(\sqrt{q})$ und damit ist unsere Laufzeit durch $\mathcal{O}(\sqrt{q})$ Schritte beschränkt. Andererseits müssen wir uns lediglich die beiden Positionen der Känguruhs merken, d.h. wir benötigen lediglich $\tilde{\mathcal{O}}(1)$ Speicherplatz.

Abbildung 4.1: Pollard's Rho-Methode

Übung 31 *In unserer Rho-Methode habe das Anfangsstück Länge i und der Kreis Länge $j - i$. Zeige, dass sich die beiden Känguruhs im Punkt $s_m = s_{2m}$ mit*

$$m = (j - i) \cdot \left\lceil \frac{i}{j - i} \right\rceil$$

treffen.

Wir müssen noch unser f spezifizieren, dass unseren Random Walk definiert. Dazu partitionieren wir unsere Gruppe G in drei etwa gleichgroße Mengen S_1, S_2 und S_3. Wenn G eine Untergruppe von Z_p^* ist, dann können wir auch Z_p^* anstatt G partitionieren, da wir G im allgemeinen nicht kennen.

Als Startwert für unsere Sequenz wählen wir $s_0 = \alpha^0\beta^0$ mit $x_0 = y_0 = 0$. Nun definieren wir unseren Random Walk als

$$s_{i+1} = f(s_i) = \begin{cases} \alpha \cdot s_i & \text{für } s_i \in S_1 \\ \beta \cdot s_i & \text{für } s_i \in S_2 \\ s_i^2 & \text{für } s_i \in S_3 \end{cases}$$

Entsprechend müssen wir unsere Zusatzwerte x_i und y_i während des Random Walks anpassen.

$$x_{i+1} = \begin{cases} x_i + 1 \bmod q & \text{für } s_i \in S_1 \\ x_i & \text{für } s_i \in S_2 \\ 2x_i \bmod q & \text{für } s_i \in S_3 \end{cases} \quad \text{bzw.} \quad y_{i+1} = \begin{cases} y_i & \text{für } s_i \in S_1 \\ y_i + 1 \bmod q & \text{für } s_i \in S_2 \\ 2y_i \bmod q & \text{für } s_i \in S_3 \end{cases}.$$

Zusammenfassend erhalten wir folgenden Algorithmus für das DL-Problem.

Pollard's Rho-Methode für das DL-Problem

EINGABE: p, α, β mit $\beta = \alpha^a \bmod p$, $\text{ord}(\alpha) = q$.

1. Partitioniere $\mathbb{Z}_p^*$ in S_1, S_2 und S_3.
2. Setze $K_1 = K_2 = s_0 = 1$ und $x_0 = y_0 = 0$.
3. FOR $i = 1$ TO q
   a) Berechne $K_1 = s_i = f(K_1)$, $K_2 = s_{2i} = f(f(K_2))$ und passe die Informationen für x_i, y_i bzw. x_{2i}, y_{2i} entsprechend an.
   b) Falls $K_1 = K_2$, EXIT

AUSGABE: $a = \frac{x_{2i} - x_i}{y_i - y_{2i}} \bmod q$

Satz 32 *Unter der Annahme, dass sich f wie eine zufällige Funktion verhält und $y_i \neq y_{2i} \bmod q$ gilt: Der obige Algorithmus löst das DL-Problem in Zeit $\tilde{\mathcal{O}}(\sqrt{q})$ mit Speicherplatz $\tilde{\mathcal{O}}(1)$.*

5 Seitenkanalangriffe

In diesem Kapitel beschäftigen wir uns mit sogenannten *Seitenkanalangriffen* auf Public-Key Kryptosysteme. Diese Seitenkanalangriffe unterscheiden sich grundlegend von den sonstigen von uns betrachteten Angriffen. Normalerweise versuchen wir, den kryptographischen Algorithmus selbst anzugreifen, die Verschlüsselungsfunktion zu invertieren oder ein zugrundeliegendes mathematisch komplexes Problem wie das Faktorisierungsproblem oder das DL-Problem zu lösen.

Bei *Seitenkanalangriffen* greift man dagegen nicht den kryptographischen Algorithmus direkt an, sondern die Implementierung des Algorithmus auf einem Computer. Dabei kann man verschiedene Eigenschaften, die sogenannten *Seitenkanäle*, analysieren:

Zeit: Wie lange dauert es, bis eine Berechnung der Entschlüsselungsfunktion für verschiedene Nachrichten beendet ist? Da der geheime Schlüssel in diese Berechnung eingeht, kann man aus der Dauer Rückschlüsse auf den geheimen Schlüssel ziehen. Die Berechnungsdauer wurde von Kocher 1996 als Seitenkanal vorgeschlagen und war der Startpunkt für weitere Seitenkanalangriffe. Wir werden im folgenden Abschnitt Kochers Idee beschreiben.

Fehler: Was passiert, wenn während der Berechnung der Ver-/Entschlüsselungsfunktion ein Fehler bei der Berechnung auftritt? Kann man dann Rückschlüsse auf den zugrundeliegenden Klartext oder den geheimen Schlüssel ziehen? Boneh, Durfee und Frankel zeigten 1997, dass Fehler in der Berechnung die geheimen Parameter preisgeben können. Wir werden den Angriff von Boneh, Durfee und Frankel auf CRT-RSA in diesem Kapitel kennenlernen.

Stromverbrauch: Wie ändert sich der Stromverbrauch eines Rechners/einer Smartcard während der Berechnung der Entschlüsselungsfunktion? Wie kann man aus dem Stromverbrauchs-Profil den geheimen Schlüssel ermitteln?

Abstrahlung: Welches elektromagnetische Profil kann man während der Berechnung der Entschlüsselungsfunktion messen? Welches akustische Profil kann man vom Prozessor erstellen? Wie kann man aus diesen Informationen die geheimen Parameter ermitteln?

Die obige Liste ist nur eine Zusammenstellung der gebräuchlichsten Charakteristika, die man misst, um die geheimen Parameter zu bestimmen. Der Phantasie sind allerdings wenig Grenzen gesetzt, welche Faktoren in der Praxis noch eine Rolle spielen können,

damit aus einem in der Theorie mathematisch sicheren System in der realen Welt ein komplett unsicheres System wird. Hierbei ist noch nicht berücksichtigt, dass in der Praxis fehlerhafte Implementierungen der Kryptoalgorithmen einem Angreifer seine Arbeit erleichtern können (z.B. Verwendung eines schlechten Zufallszahlengenerators, mehrfache Verwendung desselben Zufallsstrings, etc.).

Man beachte, dass alle Seitenkanal-Angriffe voraussetzen, dass man Kontrolle über die Geräte hat, die die geheime Information speichern. D.h. wir sind in einem Szenario, in dem ein Angreifer Eve nicht nur verschickte Nachrichten abhört, sondern aktiv auf die bei der Berechnung beteiligten Rechner Zugriff hat. Z.B. könnte man sich vorstellen, dass Eve Zugriff auf Alices Rechner hat, aber nicht den geheimen Schlüssel auslesen kann. Oder Eve hat Alices Smartcard gestohlen und versucht, die darauf gespeicherten geheimen Parameter zu ermitteln, ohne dass sie diese direkt auslesen kann.

Da Seitenkanalangriffe Eve mit weitreichenden Möglichkeiten ausstatten, gibt es derzeit keine Public-Key System, die grundsätzlich sicher gegenüber Seitenkanalangriffen sind.

5.1 Kochers Timing Angriff

Kochers Angriff von 1996 gilt als die Pionierarbeit im Bereich der Seitenkanalangriffe. Er zeigt, dass man aus der Berechnungszeit der Exponentiation in Signaturverfahren wie RSA und DSA den geheimen Schlüssel ermitteln kann. Wir zeigen seinen Angriff hier exemplarisch am Beispiel der RSA-Exponentiation. Dazu erinnern wir uns an den Repeated-Squaring Algorithmus aus dem Beweis zu Satz 18. Hier ist der Algorithmus für die Berechnung einer RSA-Signatur $\text{sig}_k(x) = x^d \bmod N$ dargestellt.

Algorithmus Repeated Squaring für RSA-Signaturen

EINGABE: $N, x, d = d_{m-1} \dots d_0$ mit $d = \sum_{i=0}^{m-1} d_i 2^i$

1. Setze $z := 1$.
2. FOR $i = 0$ TO $m - 1$
   a) IF $(d_i = 1)$ setze $z := z \cdot x \bmod N$.
   b) IF $(i < m - 1)$ setze $x := x^2 \bmod N$.

AUSGABE: $z = x^d \bmod N$

Nehmen wir an, dass ein Angreifer sich beliebige Nachrichten x signieren lassen kann. Zusätzlich kennt er Hardwarespezifikationen des Rechners, aus denen er ermitteln kann, wieviel Zeit eine modulare Multiplikation mit gegebenen Parametern benötigt.

Ein Timing Angriff bestimmt nun den geheimen Schlüssel d bitweise. Wir wissen zu Beginn des Angriffs, dass $d_0 = 1$ (Warum?). Angenommen ein Angreifer Eve kennt bereits die Bits $d_0, \ldots, d_{j-1}$ und will im nächsten Schritt das Bit d_j ermitteln.

Dazu generiert er eine große Anzahl Nachrichten $x_1, x_2, \ldots, x_n$, lässt jede der Nachrichten signieren und misst die Zeit. Sei t_k die Zeit, die man zum Signieren von m_k benötigt. Die t_k unterscheiden sich im allgemeinen, da die durchgeführten Zwischenoperationen verschieden viele modulare Reduktionen benötigen.

Da wir annehmen, dass Eve bereits j Bits von d kennt, kann sie die ersten j Schleifendurchläufe im Repeated Squaring Algorithmus selbst berechnen. Damit kennt sie auch die Werte von z und x vor dem Schleifendurchlauf mit $i = j$.

Falls $d_j = 1$ ist, führt der Algorithmus eine Multiplikation $z := z \cdot x_k \bmod N$ aus. Im Falle $d_j = 0$ entfällt diese Operation. Da der Angreifer die Werte von z und x_k vor dem Schleifendurchlauf kennt, kann er für jedes der x_k die Zeit $t_k^{(j)}$ bestimmen, die eine Multiplikation $z := z \cdot x_k \bmod N$ auf dem Rechner benötigt. Die Zeit $t_k^{(j)}$ variiert dabei für unterschiedliche Werte von x_k.

Nun weist die Gesamtzeit t_k zur Berechnung der kompletten Exponentiation genau dann eine Korrelation zur Berechnungzeit $t_k^{(j)}$ in der Schleifeniteration $i = j$ auf, falls $d_j = 1$. Nur in diesem Fall macht $t_k^{(j)}$ einen Anteil zur Gesamtzeit der Berechnung aus. Falls $d_j = 0$ ist, dann ist die Gesamtzeit t_k unabhängig von $t_k^{(j)}$, da diese Multiplikation nicht durchgeführt wird. Einfach gesagt sollte im Fall $d_j = 1$ die Gesamtzeit t_k für große Werte von $t_k^{(j)}$ größer sein und umgekehrt. Im Fall $d_j = 0$ sollte kein Zusammenhang zwischen den Größen t_k und $t_k^{(j)}$ bestehen.

Die Korrelation der Werte t_k und $t_k^{(j)}$ kann man mittels statischer Methoden berechnen. Erhält man eine hohe Korrelation, so schließt man, dass $d_j = 1$ ist, ansonsten setzt man $d_j = 0$. Da man sich nicht sicher sein kann, dass d_j einen bestimmten Wert besitzt, bezeichnet man die ermittelte Sequenz $d_0 \ldots d_j$ auch als *Hypothese*. Nun versucht man, das Bit d_{j+1} mit Hilfe der zuvor aufgestellten Hypothese $d_0 \ldots d_j$ analog zu ermitteln, solange bis man alle Bits von d bestimmt hat. Bemerkt man an einer bestimmten Stelle im Algorithmus, dass eine Hypothese inkorrekt ist, so kann es erforderlich sein, den Angriff zu einer früheren Hypothese zurückzusetzen.

Übung 33 *Überlegen Sie sich einfache Gegenmaßnahmen gegen Kochers Timing Angriff. Welche Vor- bzw. Nachteile haben Ihre Gegenmaßnahmen?*

5.2 Der Bellcore Angriff

Der folgende Angriff wurde 1997 von Boneh, Lipton und deMillo vorgestellt und heißt auch Bellcore Attacke (da der Angriff bei der Firma Bellcore entwickelt wurde). Er beschreibt einen sogenannten Fehlerangriff auf das RSA Kryptosystem. Dabei nehmen wir

an, dass ein Angreifer die Berechnung einer Signatur derart stören kann, dass temporär ein Fehler (z.B. ein Bitflip im Ergebnis einer Berechnung) auftritt. Wenn die Berechnung auf einer Smartcard durchgeführt wird und der Angreifer im Besitz dieser Karte ist, dann hat er zum Erzeugen eines Fehlers verschiedenste Möglichkeit, wie z.B.

- Anlegen einer falschen Versorgungsspannung
- Magnetische Felder
- Ändern von Registern mittels Laser
- Hitze
- Chemikalien, etc.

Hier setzt man voraus, dass der Angreifer Kontrolle über das Medium hat, auf dem eine Berechnung mit dem geheimen Schlüssel durchgeführt wird. Es kann allerdings auch vorkommen, dass der Angreifer die Berechnung gar nicht stören muss, da sie entweder fehlerhaft implementiert ist oder die zugrundeliegende Hardware fehlerhaft ist. So war der Bellcore Angriff zunächst motiviert durch einen Mitte der 90er Jahre bekannt gewordenen Fehler des Pentium-Chips, der bei bestimmten Eingaben fehlerhaft rechnete.

Wir beschreiben hier den Bellcore Angriff auf CRT-RSA. Es gibt auch eine komplexere Variante des Angriffs auf das normale RSA System. Dazu erinnern wir uns an die Signaturgenerierung bei CRT-RSA wie in Kapitel 3 beschrieben.

RSA-Signaturen mittels Chinesischem Restsatz

EINGABE: N, d, x

1. Berechne $d_p = d \bmod p-1$ und $d_q = d \bmod q-1$.
2. Berechne $y_p = x^{d_p} \bmod p$.
3. Berechne $y_q = x^{d_q} \bmod q$.
4. Berechne die eindeutige Lösung $y \in \mathbb{Z}_N$ des Gleichungssystems

$$\left| \begin{array}{rcl} y & = & y_p \bmod p \\ y & = & y_q \bmod q \end{array} \right| .$$

AUSGABE: $y = x^d \bmod N$

Hier bezeichnen wir die Unterschrift einer Nachricht x mit $y = \mathrm{sig}_k(x)$.

Nehmen wir nun an, ein Angreifer stört die Berechnung in Schritt 3. D.h. y_p wird noch korrekt berechnet, aber bei der Berechnung von y_q tritt ein Fehler auf. Damit gilt für das im letzten Schritt berechnete y

$$\left|\begin{array}{lcl} y & = & x^d \bmod p \\ y & \neq & x^d \bmod q \end{array}\right| \quad \Leftrightarrow \quad \left|\begin{array}{lcl} y^e & = & x \bmod p \\ y^e & \neq & x \bmod q \end{array}\right| \quad \Leftrightarrow \quad \left|\begin{array}{lcl} y^e - x & = & 0 \bmod p \\ y^e - x & \neq & 0 \bmod q \end{array}\right|.$$

Daraus folgt für die Unterschrift y zur Nachricht x, dass $y^e - x$ ein Vielfaches von p ist aber nicht von q. Berechnen wir den ggT mit N, dann erhalten wir

$$\mathrm{ggT}(y^e - x, N) = p.$$

D.h. wir erhalten aus der falsch berechneten Unterschrift sofort die Faktorisierung des RSA-Moduls N.

Übung 34 *Überlegen Sie sich einen einfachen Test, den man nach der Berechnung des Wertes y durchführen kann, um den Bellcore Angriff zu verhindern. Was ist der Nachteil eines solchen Tests?*

5.3 Ein Fehlerangriff auf den öffentlichen Schlüssel e

Bisher haben wir bei unseren Seitenangriffen ausgenutzt, dass eine Berechnung mit dem geheimen Schlüssel Informationen über diesen preisgeben kann. Beim Kocher Timing Angriff nutzt man unterschiedliche Berechnungszeiten aus, je nach dem ob die Bits des geheimen Schlüssel 0 oder 1 sind. Bei der Bellcore Attacke berechnet man mit Hilfe des geheimen Schlüssels ein Vielfaches von p aber nicht von q.

Es scheint natürlich, dass Algorithmen die den geheimen Schlüssel verwenden, Charakteristika des Schlüssels preisgeben können. Was passiert aber, wenn man nicht die Entschlüsselungs/-Signierfunktion angreift sondern die Verschlüsselungs-/Verifikationsfunktion. Diese verwendet ja nur öffentliche Parameter. Den geheimen Schlüssel kann man hier nicht lernen, allerdings kann man unter gewissen Voraussetzungen die zugrundeliegende geheime Nachricht m ermitteln, wie wir bei dem folgenden Angriff exemplarisch sehen.

Angenommen Alice schickt eine Nachricht an Bob. Dazu will Alice ihre Nachricht m mit Bobs RSA-Schlüssel (N, e) verschlüsseln, der in einem öffentlichen Verzeichnis abgelegt ist. Zusätzlich soll gelten, dass e eine Primzahl ist. Nun gelingt es Eve, den öffentlichen Exponenten e temporär durch $e' \neq e$ zu ersetzen.

Daher schickt Alice an Bob die verschlüsselte Nachricht $c' = m^{e'} \bmod N$. Bob versucht, die Nachricht mit seinem geheimen Schlüssel d zu entschlüsseln. Er stellt fest, dass die entschlüsselte Nachricht keinen Sinn ergibt. Daher bittet er Alice, ihre Nachricht noch einmal an ihn zu senden, da bei der ersten Nachricht offenbar ein Fehler aufgetreten ist.

Diesmal greift Eve nicht in die Berechnung ein, und Alice kann den korrekten Chiffretext $c = m^e \bmod N$ senden.

Nun kann Eve aber aus den beiden Werten c und c', den zugrundeliegenden Klartext m effizient berechnen. Dieses Resultat stammt von Simmons und wird als Common-Modulus Angriff bezeichnet:

Satz 35 *Seien $c = m^e \bmod N$ und $c' = m^{e'} \bmod N$ gegeben mit $ggT(e, e') = 1$ und $\max\{e, e'\} \leq N$. Dann kann m in Zeit $\mathcal{O}(\log^3 N)$ berechnet werden.*

Beweis: Berechne mit Hilfe des EEA (Fakt 8) ganze Zahlen u, v mit

$$eu + e'v = \mathrm{ggT}(e, e') = 1.$$

Damit gilt

$$c^u \cdot (c')^v = (m^e)^u \cdot (m^{e'})^v = m^{eu+e'v} = m^1 \bmod N.$$

Es ist nicht schwer zu zeigen, dass u, v die Ungleichungen $|u| < e$ und $|v| < e'$ erfüllen. Daher gilt $|u|, |v| < N$. Nach Satz 18 kann m daher in Zeit $\mathcal{O}(\log^3 N)$ berechnet werden. □

6 Gitter und ganzzahlige lineare Gleichungssysteme

In der Public-Key Kryptographie hat man sehr häufig modulare Gleichungssysteme, die geheime Parameter und öffentliche Parameter beinhalten. Jedes Gleichungssystem kann man linearisieren, indem man für alle unbekannten Terme neue Variablen einführt. Wir werden in diesem Kapitel zeigen, dass die Unbekannten häufig in Polynomialzeit berechnet werden können, wenn ihr Produkt kleiner als der Modul der linearen Gleichung ist. Dazu verwenden wir Methoden aus der Theorie ganzzahliger Gitter.

6.1 Einführung in Gittertheorie

Wir werden im folgenden zwei äquivalente Definitionen von Gittern geben.

Definition 36 (Gitter1) *Ein Gitter L ist eine diskrete additive abelsche Untergruppe des $\mathbb{R}^n$.*

Dies bedeutet, ein Gitter besteht aus einer Menge von Punkten (bzw. Vektoren) im Euklidischen Raum und es gelten die Bedingungen aus Definition 4 für abelsche Gruppen:

Abgeschlossenheit: Für zwei Punkte des $u, v \in L$ ist auch $u + v$ ein Gitterpunkt.

Neutrales Element: Jedes Gitter enthält den Nullvektor 0^n.

Inverses Element: Für jedes u ist $-u$ in L.

Ferner gelten Assoziativität und Kommutativität der Vektoraddition. Dass L *diskret* ist, bedeutet dass es keinen Häufungspunkt in L gibt, d.h. dass es um jeden Punkt u des Gitters einen offenen Ball gibt, der nur u enthält. Den Raum $\mathbb{R}^n$ bezeichnet man auch als Einbettungsraum oder etwas lax als den *Raum, in dem L lebt.* Wir werden als Einbettungsraum bei unseren Angriffen später stets den $\mathbb{Z}^n$ verwenden. In diesem Raum ist die Diskretheitsbedingung trivialerweise erfüllt.

Beispiel 37 *(1) $\mathbb{Z} \subset \mathbb{R}$ ist ein Gitter.*

(2) $\mathbb{Z}^d \subset \mathbb{R}^n$ mit $d \leq n$ ist ein Gitter.

(3) $(k\mathbb{Z})^d \subset \mathbb{R}^n$ mit $d \leq n, k > 1$ ist ein Gitter.

Der Unterschied zu den Gruppen mit denen wir uns in den vorigen Kapiteln beschäftigt haben ist, dass Gitter eine unendliche Menge von Punkten besitzen. Wir können ein Gitter L also nicht darstellen, indem wir alle Punkte aufzählen. Bei der Darstellung von L hilft uns die zweite Definition eines Gitters.

Definition 38 (Gitter2) *: Seien $b_1, b_2, \ldots, b_d \in \mathbb{R}^n$, $d \leq n$ linear unabhängig. Dann ist die Menge*

$$L = \left\{ v \in \mathbb{R}^n \;\middle|\; v = \sum_{i=1}^{d} a_i b_i, a_i \in \mathbb{Z} \right\}$$

ein Gitter.

D.h. wir können jedes Gitter L durch eine Menge $B = \{b_1, \ldots, b_d\}$ von Vektoren repräsentiert. Wir nennen B eine Basis des Gitters L. Das Gitter L ist dann die Menge der ganzzahligen Linearkombinationen der Basisvektoren. Man sagt auch, dass L von den Basisvektoren aufgespannt wird. Die Basis B schreiben wir stets in Form einer $d \times n$ Matrix, wobei der Vektor b_i der i-te Zeilenvektor in B ist.

Den Parameter $\dim(L) := d$ nennt man die *Gitterdimension* und n die *Einbettungsdimension*. Wir werden sehr häufig Gitter betrachten mit $d = n$. Dies bezeichnet man als Gitter mit *vollem Rang*. Gitter mit vollem Rang haben eine quadratische Basismatrix.

Nun besitzen wir eine Darstellungsform von Gittern in Form von Basen. Leider ist diese Darstellung nicht eindeutig. Jede Basis eines Gitters L kann in eine andere Basis von L umgewandelt werden, indem man

- Basisvektoren permutiert oder
- zu einem Basisvektor ein Vielfaches eines anderen Vektors addiert.

Man bezeichnet diese Umwandlungsschritte auch als unimodulare Transformationen. In der Matrixschreibweise entspricht dies der Multiplikation der Basismatrix mit einer $d \times d$-Matrix mit Determinante ± 1 (Übungsaufgabe). Betrachten wir nun ein Gitter L mit

Abbildung 6.1: 2-dimensionales Gitter L mit verschiedenen Basen

vollem Rang $d = n$. Dies Gitter wird von einer Basis mit einer quadratischen Basismatrix

B erzeugt. Daher können wir leicht die Determinante $\det(B)$ der Basismatrix berechnen. Den Absolutbetrag dieser Determinante bezeichnen wir als die Determinante von L, d.h. $\det(L) = |\det(B)|$. Man kann die Determinante auch für Gitter definieren, die keinen vollen Rang haben (über die sogenannte *Gram-Schmidt Orthogonalisierung*), aber wir werden i. allg. nur Gitter mit vollem Rang betrachten.

Da unimodulare Transformationen durch Multiplikationen mit einer Matrix T mit $\det(T) = \pm 1$ beschrieben werden können, ändert sich die Determinante bei einem Basiswechsel nicht. Sie ist eine *Invariante* des Gitters. Die Gitterdeterminante hat eine geometrische Interpretation; sie entspricht dem Volumen des von den Vektoren der Basis B aufgespannten *Parallelepipeds*

$$P(B) := \left\{ v \in \mathbb{R}^n \;\middle|\; \sum_{i=1}^{n} x_i b_i, x_i \in \mathbb{R} \text{ mit } 0 \leq x \leq 1 \right\}.$$

Man nennt $P(B)$ auch die *Grundmasche* einer Basis. Wir werden im folgenden Beispiel

Abbildung 6.2: Das Volumen der Grundmaschen ist invariant.

sehen, dass die zweite Definition eines Gitters nicht immer die bessere Definition ist. Wie schon eingangs erwähnt, ist es unser Ziel, mit Hilfe der Gittertheorie modulare lineare Gleichungen zu lösen. Sei

$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = 0 \bmod N$$

eine homogene Gleichung mit den Unbekannten x_i. Mit Hilfe unserer ersten Definition sehen wir unmittelbar, dass die Menge der Lösungen $(x_1, \ldots, x_n)$ ein n-dimensionales Gitter L ist. Es gilt $0^n \in L$ und für $u, v \in L$ ist $u - v \in L$. Dagegen ist die Konstruktion einer Basis für L gemäß der zweiten Definition nicht offensichtlich (es gibt aber einen Polynomialzeitalgorithmus, der eine Basis konstruiert).

Analog bildet die Lösungsmenge eines homogenen linearen Gleichungssystems ein Gitter.

Übung 39 *Sei $m \geq n$ und $A \in Z_{n \times m}$ eine ganzzahlige $n \times m$ Matrix, wobei die Zeilenvektoren linear unabhängig sind. Zeigen Sie, dass die Menge $\{x \in \mathbb{Z}^m \mid Ax = 0\}$ ein Gitter L mit Gitterdimension $\dim(L) = m - n$ ist.*

Da es unendliche viele unimodulare Transformationen gibt, ist auch die Anzahl verschiedener Basen eines Gitters L unendlich. Welches ist aber eine gute Basis? Einfach gesagt bestehen die zur Lösung wichtiger algorithmischer Probleme interessanten Basen aus Vektoren die *kurz* und *paarweise orthogonal* sind.

Man bezeichnet die Länge der kürzesten Vektoren eines Gitters als *sukzessive Minima.*

Definition 40 (Sukzessive Minima) *Sei L ein d-dimensionales Gitter. Für $i \leq d$ bezeichnen wir mit $\lambda_i(L)$ den minimalen Radius eines Balls um den Nullpunkt in L, so dass der Ball i linear unabhängige Vektoren enthält. Man nennt $\lambda_i(L)$ das* i-te sukzessive Minimum.

Abbildung 6.3: Sukzessive Minima $\lambda_1 = \|b_1\|, \lambda_2 = \|b_2\|$

Man beachte, dass ein kürzester Vektor $v = (v_1, \ldots, v_n)$ mit Euklidischer Norm $\|v\| = \sqrt{\sum_{i=1}^{n} v_i^2}$ nicht eindeutig ist, da $-v \in L$ dieselbe Länge hat.

Man könnte intuitiv vermuten, dass n linear unabhängige Vektoren mit Längen $\lambda_1, \ldots, \lambda_n$ eine optimale Basis für ein Gitter definieren. Leider bilden diese Vektoren für $d \geq 5$ nicht immer eine Basis. Betrachten wir die folgende Basis

$$B = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Es gilt $\lambda_i = 2$ für $i = 1, \ldots, 5$. Diese sukzessiven Minima werden von den ersten vier Basisvektoren und dem Vektor $(0, 0, 0, 0, 2) \in L$ angenommen. Die fünf kürzesten Vektoren bilden also eine Basis des $(2\mathbb{Z})^5$, in dem der Vektor $(1, 1, 1, 1, 1)$ nicht enthalten ist. Man darf also den letzten Basisvektoren nicht durch $(0, 0, 0, 0, 2)$ ersetzen, obwohl er Euklidische Norm $\sqrt{5} > \lambda_5$ hat.

Die Basisvektoren erreichen daher i. allg. nicht die sukzessiven Minima. Es existiert allerings stets eine Basis, deren kürzester Vektor die Länge λ_1 hat. Für die Länge eines kürzesten Vektors in L liefert der 1. Satz von Minkowski eine obere Schranke, die nur von den Gitterinvarianten $\det(L)$ und $\dim(L) = d$ abhängt.

Überlegen wir uns zunächst, was wir für die Länge eines kürzesten Vektors erwarten. Wir wissen, das die Determinante dem Volumen des von den Basisvektoren aufgespannten Parallelepipeds entspricht. Damit sollte die Länge der kürzesten Seite eines Parallelepipeds von einer Größenordnung höchstens der d-ten Wurzel der Determinante sein. Dies entspricht bis auf einen kleinen Faktor der Schranke im 1.Satz von Minkowski.

Fakt 41 (1.Satz von Minkowski) *In jedem d-dimensionalen Gitter L gilt*

$$\lambda_1 \leq \sqrt{d}\det(L)^{\frac{1}{d}}.$$

Minkowskis zweiter Satz liefert eine Schranke für das Produkt aller sukzessiven Minima.

Fakt 42 (2.Satz von Minkowski) *In jedem d-dimensionalen Gitter L gilt*

$$\lambda_1 \cdot \lambda_2 \cdot \cdots \cdot \lambda_d \leq \sqrt{d^d}\det(L).$$

Für ein "zufälliges" Gitter erwartet man, dass

$$\lambda_1 \approx \lambda_2 \approx \cdots \approx \lambda_d \approx \det(L)^{\frac{1}{d}}.$$

Wir werden oft eine Heuristik verwenden, die eine Umkehrung des ersten Satzes von Minkowski darstellt. D.h. wir werden annehmen, dass es nur einen einzigen Vektor in einem Gitter gibt, der eine Norm kleiner als die Minkowski-Schranke $\sqrt{d}\det(L)^{\frac{1}{d}}$ aus Satz 41 hat. D.h. wenn es uns gelingt einen derart kurzen Vektor zu berechnen, dann nehmen wir an, dass er ein kürzester Vektor des Gitters ist.

Dies führt uns direkt zum algorithmischen Problem, in einem Gitter einen kürzesten Vektor zu bestimmen.

Definition 43 (SVP) *Beim* Kürzester-Vektor Problem (Shortest-Vector Problem = SVP) *erhält man als Eingabe eine Basis B für ein d-dimensionales Gitter L im $\mathbb{R}^n$. Gesucht ist ein Vektor mit Länge λ_1 in L.*

Beim γ-SVP ist ein Vektor mit Länge höchstens $\gamma \cdot \lambda_1$ gesucht.

Man beachte, dass das γ-SVP ein Approximationsproblem ist; gesucht ist ein kürzester Vektor bis auf einen Faktor von γ. Eng verwandt mit dem Kürzester-Vektor Problem ist das Problem, einen *nächsten Gittervektor* zu finden.

Definition 44 (CVP) *Beim* Nächster-Vektor Problem (Closest-Vector Problem = CVP) *erhält man als Eingabe eine Basis für ein d-dimensionales Gitter L im R^n und zusätzlich einen (Target-)Vektor $t \in R^n$. Gesucht ist ein Gittervektor u, der möglichst nah an t liegt, d.h.*

$$\|u - t\| = \min_{v \in L} \|v - t\|.$$

Analog zum γ-SVP sucht man bei der Approximationsvariante γ-CVP einen Vektor u mit

$$\|u - t\| \leq \gamma \cdot \min_{v \in L} \|v - t\|.$$

Abbildung 6.4: Nächster-Vektor Problem (CVP)

Wir wollen nun untersuchen, welche algorithmische Komplexität die Probleme SVP und CVP besitzen. Dazu werden wir zunächst zeigen, dass man das Subset-Sum Problem auf CVP reduzieren kann. Die Berechnungsvariante des Subset-Sub Problems ist folgendermaßen definiert:

- Gegeben: $a_1, a_2, \ldots, a_n, s \in \mathbb{N}$
- Gesucht: $x_1, x_2, \ldots, x_n \in \{0, 1\}$ mit $\sum_{i=1}^{n} x_i a_i = s$

Bei der Sprachvariante des Subset-Sum Problems muss man im Gegensatz zur Berechnungsvariante lediglich entscheiden, ob es einen n-dimensionalen 0,1-Vektor x gibt mit der obigen Eigenschaft, ohne dass man x selbst berechnen muss. Von der Sprachvariante des Subset-Sums Problem weiß man, dass sie NP-vollständig ist.

Eine Reduktion von Subset-Sum auf CVP zeigt, dass das Nächste-Gittervektor Problem NP-hart ist, d.h. wir können nicht erwarten, dass wir einen Polynomialzeit Algorithmus für das CVP finden können.

Satz 45 *SubsetSum* $\leq_p$ *CVP*

Beweis: Sei $\sum_{i=1}^{n} x_i a_i = s$ eine Subset-Sum Instanz. Mit Hilfe des Erweiterten Euklidischen Algorithmus kann man ganze Zahlen $y_1, \ldots, y_n$ berechnen, so dass $s = \sum_{i=1}^{n} y_i a_i$ (Übungsaufgabe).

Betrachten wir nun das $(n-1)$-dimensionale Gitter L der Lösungen $(z_1, \ldots, z_n)$, die die folgende homogene Gleichung erfüllen:

$$z_1 a_1 + \cdots + z_n a_n = 0.$$

Man beachte, dass der Vektor $d = (y_1 - x_1, \ldots, y_n - x_n)$ für jeden Lösungsvektor $(x_1, \ldots, x_n)$ des Subset-Sum Problems zum Gitter L gehört, denn

$$(y_1 - x_1)a_1 + \cdots + (y_n - x_n)a_n = \sum_{i=1}^{n} y_i a_i - \sum_{i=1}^{n} x_i a_i = s - s = 0.$$

Nun ist der bekannte Vektor $y = (y_1, \ldots, y_n)$ aber sehr nah zum Vektor d. Seine Distanz zum Gittervektor d ist $\|x\| \leq \sqrt{n}$. Wenn die Nullen und Einsen in x gleichverteilt sind, dann ist der erwartete Abstand $\sqrt{n/2}$.

Da x ein $0,1$-Vektor ist, beträgt der Abstand von d zum Vektor $y' = (y_1 - 1/2, \ldots, y_n - 1/2)$ sogar nur exakt $\sqrt{n/4}$. Es ist nicht schwer (und daher prädestiniert für eine Übungsaufgabe), die folgenden beiden Aussagen zu zeigen:

- d ist ein nächster Gittervektor zum Targetvektor y'.
- Jeder Gittervektor in L, der Abstand exakt $\sqrt{n/4}$ zum Targetvektor y' hat, ist von der Form $(y_1 - x'_1, \ldots, y_n - x'_n)$ mit $s = \sum_{i=1}^{n} x'_i a_i$ und $x'_i \in \{0, 1\}$.

D.h. jeder nächste Vektor in L zum Targetvektor y' liefert uns einen $0,1$-Vektor $x' = (x'_1, \ldots, x'_n)$, der das Subset-Sum Problem löst. □

Dass das Nächster-Vektor Problem NP-hart ist, wurde bereits 1981 von van Emde Boas gezeigt. Ajtai zeigte 1996, dass auch dass Kürzeste Vektor Problem NP-hart ist (unter sogenannten randomisierten Reduktionen).

Wir können also nicht hoffen, dass wir einen Polynomialzeit Algorithmus finden, der eines der beiden Probleme löst. Polynomialzeit bedeutet bei SVP und CVP: Angenommen man bekommt als Eingabe eine Gitterbasis B eines d-dimensionalen Gitters im $\mathbb{Q}^n$ mit maximalem Eintrag $b_{max} = |\max_{i,j} b_{i,j}|$. Dann ist ein polynomieller Algorithmus ein Algorithmus, dessen Laufzeit polynomiell in den Parametern d, n und $\log(b_{max})$ ist. Man beachte, dass man Computern mit Gleitkomma-Arithmetik nur rationale Gitterbasen in $\mathbb{Q}^n$ als Eingabe geben kann.

Obwohl es unter der Annahme P≠NP für SVP und CVP keinen Polynomialzeit Algorithmus gibt, kann man beide Probleme in Gittern konstanter Dimension in Polynomialzeit lösen. Wenn die Gitterdimension $d = 2$ ist, dann findet der *Gauß-Algorithmus* einen kürzesten Vektor in polynomieller Zeit.

Fakt 46 (Gauß-Algorithmus) *Gegeben eine Basis $B \in \mathbb{Z}^{2\times 2}$ eines 2-dimensionalen Gitters L. Dann berechnet der Gaußalgorithmus in Zeit $\mathcal{O}(\log^2(b_{max}))$ eine Basis, deren Basisvektoren Länge $\lambda_1(L)$ und $\lambda_2(L)$ besitzen.*

Der Gauß-Algorithmus ist eine vektorielle Generalisierung des uns bekannten Euklidischen Algorithmus. Er iteriert dabei die folgenden beiden Schritte bis die Basisvektoren nicht mehr verkürzt werden können.

- Ziehe vom längeren Basisvektor ein Vielfaches des kürzeren Basisvektor ab, so dass der resultierende Basisvektor minimale Länge hat. Man beachte, dass dies eine unimodulare Transformation ist.
- Vertausche die beiden Basisvektoren. Auch dies ist eine unimodulare Transformation.

Man kann zeigen, dass der Gauß-Algorithmus zwei Vektoren ausgibt, deren Längen den sukzessiven Minima des Gitters entsprechen. Wir wollen den Gauß-Algorithmus anhand eines Beispiels illustrieren.

Beispiel 47 *Wir starten den Gauß-Algorithmus mit der Eingabe* $a = (11, 6)$ *und* $b = (8, 4)$.

$$\|(11, 6) - c \cdot (8, 4)\| \text{ ist minimal für } c = 1.$$

Daher erhalten wir in der nächsten Iteration $a = (8, 4)$ *und* $b = (3, 2)$.

$$\|(8, 4) - c \cdot (3, 2)\| \text{ ist minimal für } c = 2.$$

Wir erhalten $a = (3, 2)$ *und* $b = (2, 0)$.

$$\|(3, 2) - c \cdot (2, 0)\| \text{ ist minimal für } c = 1.$$

Wir erhalten als neue Basisvektoren $a = (2, 0)$ *und* $b = (1, 2)$. *Diese Basis kann durch weitere Iterationen nicht mehr verkürzt werden. Daher sind die sukzessiven Minima des Gitters* $\lambda_1 = 2$ *und* $\lambda_2 = \sqrt{5}$.

Alle Gitter, die wir im folgenden Kapitel für Angriffe betrachten werden, haben Dimension 2 oder 3. Wir können in diesen Gitter das SVP (und auch das CVP) lösen.

Fakt 48 *In jedem Gitter* L *konstanter Dimension* d *mit Basis* $B \in \mathbb{Z}^{d \times n}$ *können das SVP und das CVP in Zeit polynomiell in* b_{max} *und* n *gelöst werden.*

In den folgenden Kapiteln werden wir allerdings auch Gitter mit nicht-konstanter Dimension betrachten. In diesen Gittern können SVP und CVP in Zeit exponentiell in der Gitterdimension gelöst werden. Da wir als (angehende) Kryptanalysten aber hauptsächlich an Polynomialzeit Angriffen interessiert sind, verwenden wir Polynomialzeit Approximationsalgorithmen für γ-SVP und γ-CVP.

Sehr oft genügt es statt eines kürzesten Vektors nur einen relativen kurzen Vektor zu berechnen, also einen Vektor, der bis auf einen Faktor γ die Länge von λ_1 hat. Der Algorithmus, den wir dabei einsetzen werden, wird *LLL-Algorithmus* (oder auch L^3-Algorithmus) genannt und ist nach seinen Erfindern Lenstra, Lenstra und Lovász benannt. Der LLL-Algorithmus hat zahlreiche Anwendungen in der Mathematik und Theoretischen Informatik, wie z.B. Faktorisierung von Polynomen und ganzzahlige Programmierung. Wir werden diesen Algorithmus im nächsten Abschnitt verwenden, um Nullstellen von ganzzahligen Polynomgleichungen zu berechnen.

Der LLL-Algorithmus kann dazu genutzt werden, um die Probleme γ-SVP und γ-CVP mit einem Approximationsfaktor der Größenordnung $\gamma = 2^d$ zu lösen. Obwohl dieser Approximationsfaktor exponentiell schlecht in der Gitterdimension ist, genügt eine Approximation dieser Güte für viele interessante Probleme.

Der LLL-Algorithmus kann als eine Generalisierung des Gauß-Algorithmus für beliebige Dimensionen angesehen werden, insbesondere verwendet er den Gaußalgorithmus als Subroutine. Für unsere Angriffe genügt der folgende Fakt, dass der LLL-Algorithmus eine kurze Basis in polynomieller Zeit berechnet.

Fakt 49 (LLL 1982:) *Sei L ein Gitter mit Basisvektoren $v_1, v_2, \ldots, v_d \in \mathbb{Z}^n$. Dann berechnet der LLL-Algorithmus eine* LLL-reduzierte Basis *$b_1, b_2, \ldots, b_d$ mit*

1. $\|b_1\| \leq c^d \cdot \det(L)^{\frac{1}{d}}$,
2. $\|b_i\| \leq c^{2d} \cdot \lambda_i(L)$,

wobei $c = (\frac{4}{3})^{\frac{1}{4}} \approx 1.075$. Die Laufzeit des LLL-Algorithmus ist $\mathcal{O}(d^5 n \log^3 b_{max})$

Nun haben wir die notwendigen Werkzeuge, um das von uns gewünschte kryptanalytische Problem modularer ganzzahliger linearer Gleichungen zu lösen.

Wir werden nun zunächst ein n-dimensionales Gitter L konstruieren, in dem jede Lösung $x = (x_1, x_2, \ldots, x_n)$ eines linearen Gleichungssystems einen Gittervektor darstellt. Wir werden dann zeigen, dass ein Lösungsvektor x die Minkowski-Schranke $\|x\| \leq \sqrt{n}\det(L)^{\frac{1}{n}}$ aus Satz 41 erfüllt, wenn das Produkt der x_i hinreichend klein ist. D.h. aber dass x in diesem Fall zu den kürzesten Vektoren des Gitters gehört. Daher können wir hoffen, dass eine Lösung des SVP-Problems in L uns den Vektor x liefert. Für konstantes n können wir das SVP in L effizient berechnen.

Satz 50 *Seien $a_1, a_2, \ldots, a_n \in \mathbb{Z}$, $N \in \mathbb{N}$ mit $ggT(a_i, N) = 1$ für ein $i \in \{1, \ldots, n\}$. Ferner seien obere Schranken $X_i \in \mathbb{Q}$, $i = 1, \ldots, n$ mit*

$$\prod_{i=1}^{n} X_i \leq N$$

gegeben. Seien $x_1, x_2, \ldots, x_n \in Z$ mit $|x_i| \leq X_i$ eine Lösung der linearen Gleichung

$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = 0 \bmod N.$$

Dann kann man ein n-dimensionales Gitter L' konstruieren, in dem es einen Vektor $x' \in L'$ gibt mit:

(1) x' erfüllt die Minkowski-Schranke $\|x'\| \leq \sqrt{n}\det(L')^{\frac{1}{n}}$.

(2) $x = (x_1, \ldots, x_n)$ kann leicht aus x' berechnet werden.

Unter der Annahme, dass x' ein kürzester Vektor in L' ist, kann x durch Lösen eines SVP bestimmt werden.

Beweis: Nach Voraussetzung gibt es ein a_i, das teilerfremd zu N ist. ObdA sei dies a_n. Mit Hilfe des EEA können wir das Inverse a_n^{-1} von a_n modulo N bestimmen. Wir multiplizieren unsere Ausgangsgleichung mit $-a_n^{-1}$ und erhalten

$$b_1x_1 + b_2x_2 + \cdots + b_{n-1}x_{n-1} = x_n \bmod N, \tag{6.1}$$

wobei $b_i = -a_i a_n^{-1}$. Man beachte, dass Multiplikation mit einer zu N teilerfremden Zahl die Lösungsmenge des Gleichungssystems unverändert lässt.

Nun definieren wir zunächst das n-dimensionale Gitter L, das von den Zeilenvektoren der folgenden Basismatrix aufgespannt wird:

$$B = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 & b_1 \\ 0 & 1 & 0 & \dots & 0 & b_2 \\ & & \ddots & & & \vdots \\ 0 & 0 & 0 & \dots & 1 & b_{n-1} \\ 0 & 0 & 0 & \dots & 0 & N \end{pmatrix}.$$

D.h. die ersten $n-1$ Spaltenvektoren sind Einheitsvektoren und der letzte Spaltenvektor besteht aus den b_i und N. Seien die x_i nun eine Lösung der modularen Gleichung (6.1) mit

$$\sum_{i=1}^{n-1} b_i x_i = x_n - yN$$

für ein $y \in \mathbb{Z}$. Wir wissen, dass jede Linearkombination von Zeilenvektoren aus B ein Gittervektor in L ist. Betrachten wir nun die Linearkombination mit Koeffizienten $(x_1, x_2, \dots, x_{n-1}, y)$. Für diese gilt

$$(x_1, x_2, \dots, x_{n-1}, y) \cdot B = (x_1, x_2, \dots, x_{n-1}, \sum_{i=1}^{n-1} b_i x_i + yN) = (x_1, x_2, \dots, x_n).$$

D.h. unser gesuchter Vektor $x = (x_1, x_2, \dots, x_n)$ ist ein Gittervektor in L. Umgekehrt ist jeder Vektor in L eine Lösung der linearen Gleichung (6.1) und damit auch der Ausgangsgleichung mit den Koeffizienten a_i. Leider ist unser Vektor x i. allg. kein kurzer Vektor in L. Seine Norm kann durch $\sqrt{n}\max_i\{x_i\}$ abgeschätzt werden.

Da die Basis B obere Dreiecksgestalt hat, ist die Determinante von B das Produkt der Diagonaleinträge in B. Damit ist $\det(L) = N$. Die Minkowski-Schranke sagt, dass das erste sukzessive Minimum λ_1 kleiner ist als $\sqrt{n}\det(L)^{\frac{1}{n}} = \sqrt{n}N^{\frac{1}{n}}$.

Wir wissen, dass das Produkt der x_i nach Voraussetzung durch N beschränkt ist. D.h. also wenn alle x_i von der gleichen Größenordnung wären, $x_1 \approx \cdots \approx x_n \approx N^{\frac{1}{n}}$, dann wäre unser Vektor x ein kurzer Vektor in L, denn er hätte etwa Norm $\sqrt{n}N^{\frac{1}{n}}$.

Wir haben jedoch nicht vorausgesetzt, dass alle x_i gleiche Größe haben. Wir wollen nun unsere Basismatrix derart abändern, dass der gesuchte Vektor in jeder Komponente etwa gleich große Einträge hat.

Wir können oBdA annehmen, dass das Produkt der oberen Schranke X_i gleich N ist. Wenn das Produkt kleiner ist, erhöhen wir die Schranken entsprechend. Nun definieren wir $Y_i = \frac{N}{X_i}$. Wir multiplizieren jetzt in B den i-ten Spaltenvektor mit Y_i. Die entstehende Basismatrix

$$B' = \begin{pmatrix} Y_1 & 0 & 0 & \dots & 0 & Y_n b_1 \\ 0 & Y_2 & 0 & \dots & 0 & Y_n b_2 \\ & & \ddots & & & \vdots \\ 0 & 0 & 0 & \dots & Y_{n-1} & Y_n b_{n-1} \\ 0 & 0 & 0 & \dots & 0 & Y_n N \end{pmatrix}$$

definiert unser neues Gitter L'. Betrachten wir jetzt dieselbe Linearkombination $(x_1, x_2, \dots, x_{n-1}, y)$ von Basisvektoren aus B' wie zuvor

$$(x_1, x_2, \dots, x_{n-1}, y) \cdot B = (x_1 Y_1, x_2 Y_2, \dots, Y_n x_n) = x'.$$

Es gilt $x_i Y_i = \frac{x_i}{X_i} N \leq N$, wegen $x_i \leq X_i$. Man beachte, dass man aus dem Vektor x' leicht den Vektor x mit den Komponenten x_i bestimmen kann. Für den Vektor x' gilt:

$$\|x'\| \leq \sqrt{n} N.$$

Wir wollen nun zeigen, dass die rechte Seite der Ungleichung genau der Minkowski-Schranke für das erste sukzessive Minimum in L' entspricht. Für die Determinante von L' erhalten wir

$$\det(L') = N \prod_{i=1}^{n} Y_i = N \prod_{i=1}^{n} \frac{N}{X_i} = N^{n+1} \prod_{i=1}^{n} \frac{1}{X_i} = N^n.$$

Damit gilt für das erste sukzessive Minimum $\lambda_1(L')$ die Minkowski-Schranke $\lambda_1(L') \leq \sqrt{n} \det(L')^{\frac{1}{n}} = \sqrt{n} N$. Dies entspricht der oberen Schranke unseres Zielvektors $x' \in L'$, was zu zeigen war. □

Falls das Produkt der x_i durch N nach oben beschränkt ist, dann ist wie gezeigt $x' = (Nx_1/X_1, \dots, Nx_n/X_n)$ ein kurzer Vektor im zuvor definierten Gitter L'. Wenn das zweite sukzessive Minima größer als die Minkowski Schranke $\sqrt{n} \det(L')^{\frac{1}{n}}$ ist, dann liefert das SVP in L' diesen Vektor x' und damit auch x.

Nun wollen wir uns kurz überlegen, dass wir mit Hilfe unserer Methode auch inhomogene lineare Gleichungen

$$a_1 x_1 + a_2 x_2 + \dots + a_n x_n = b \bmod N$$

lösen können. Ein Lösungsansatz wäre ein Gitter aufzustellen, in dem alle Vektoren von der Form $(x_1, \dots, x_n, \sum_{i=1}^{n-1} x_i a_i - yN)$ sind. Dann würde man in diesem Gitter einen

nächsten Gittervektor zum Targetvektor $t = (0, \ldots, 0, b)$ suchen. D.h. man würde eine CVP-Instanz lösen.

Man kann allerdings das Lösen inhomogener linearer Gleichungen auf Satz 50 zurückführen und damit eine SVP-Instanz lösen. Dies ist eine Übungsaufgabe.

Übung 51 *Zeigen Sie ein Analogon von Satz 50 für inhomogene Gleichungssysteme*

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = b \bmod N$$

Hinweis: *Verwenden Sie ein* $(n+1)$*-dimensionales Gitter.*

Im folgenden werden wir auch oft mit einer ganzzahligen nichtmodularen Gleichung starten:

$$a_1x_1 + a_2x_2 + \cdots + a_xx_n = b.$$

Diese kann man natürlich immer in eine modulare Gleichung umwandeln, indem man sie modulo eines der Parameter a_i oder b reduziert.

7 Linearisierungsangriffe auf Kryptosysteme

Wir werden nun zeigen, wie man die im letzten Kapitel vorgestellten Gittermethoden zum Lösen linearer modularer Gleichungen dazu verwenden kann, RSA und ElGamal anzugreifen.

7.1 Der Wiener-Angriff

Der 1990 von Michael Wiener vorgestellte Angriff auf RSA mit kleinem d gilt als einer der wichtigsten Angriffe auf das RSA Kryptosystem. Wie wir schon in Kapitel 3 diskutiert haben, ist es in manchen Situationen vorteilhaft, einen kleinen geheimen Exponenten d zu wählen, da die Entschlüsselungs-/Signierfunktion d_k bei RSA die Laufzeit $\mathcal{O}(\log d \log^2 N)$ hat.

Der Angriff von Wiener zeigt jedoch, dass geheime Exponenten d, die kleiner als ein Viertel der Bitlänge von N sind, dazu führen, dass man aus den öffentlichen Parametern (N, e) in Polynomialzeit die Faktorisierung von N berechnen kann. Wir werden hier eine heuristische Variante von Wiener's Angriff zeigen. Es gibt aber auch eine beweisbare Variante des Angriffs.

Satz 52 (Wiener) *Sei (N, e) ein öffentlicher RSA-Schlüssel mit $N = pq$, wobei p und q dieselbe Bitgröße haben. Für den geheimen Exponenten d mit $ed = 1 \bmod \phi(N)$ gelte*

$$d \leq \frac{1}{3} N^{\frac{1}{4}}.$$

Dann kann die Faktorisierung von N in Zeit $\mathcal{O}(\log^2 N)$ gefunden werden.

Beweis: Wir schreiben die RSA-Gleichung $ed = 1 \bmod \phi(N)$ in der Form

$$ed = 1 + k(p-1)(q-1) = 1 + k(N - (p + q - 1))$$

für ein unbekanntes $k \in \mathbb{N}$. Man beachte, dass

$$k = \frac{ed - 1}{\phi(N)} < \frac{e}{\phi(N)} \cdot d < d$$

Unsere RSA-Gleichung können wir auch als

$$ed + k(p + q - 1) - 1 = kN \tag{7.1}$$

schreiben. Nun nehmen wir die Gleichung modulo N und linearisieren, d.h. wir weisen allen unbekannten Termen eine Variable zu:

$$ex_1 + x_2 = 0 \bmod N,$$

mit $(x_1, x_2) = (d, k(p+q-1)-1)$. Um nun Satz 50 anwenden zu können, müssen wir noch obere Schranken X_1, X_2 für x_1 und x_2 definieren. Sei $X_1 = \frac{1}{3}N^{\frac{1}{4}}$, dann gilt $x_1 \leq X_1$ nach Voraussetzung. Berechnen wir nun eine obere Schranke für x_2. ObdA gelte $p < q$. Damit ist $p < N^{\frac{1}{2}}$ und da p und q dieselbe Bitgröße haben, folgt

$$q \leq 2p \leq 2N^{\frac{1}{2}}.$$

Damit gilt insgesamt $p + q < 3N^{\frac{1}{2}}$. Wie wir zuvor gezeigt haben ist $k < d$. Daraus folgt für x_2:

$$x_2 < k(p+q) < N^{\frac{3}{4}}.$$

Wir setzen $X_2 = N^{\frac{3}{4}}$. Damit ist $X_1 X_2 < N$ und die Bedingungen aus Satz 50 sind erfüllt. D.h. wir erhalten ein 2-dimensionales Gitter L, in dem $x = (x_1, x_2)$ ein Vektor mit Norm kleiner als die Minkowski-Schranke ist. Unter der (heuristischen) Annahme, dass x ein kürzester Vektor in L ist, finden wir x_1 und x_2, indem wir mit Hilfe des Gauß-Algorithmus das SVP in L lösen.

Der Parameter $x_1 = d$ liefert uns bereits den geheimen Schlüssel. Man kann aber sogar leicht zeigen, dass man auch N faktorisieren kann. Mit Hilfe von Gleichung (7.1) sehen wir, dass man

$$k = \frac{ex_1 + x_2}{N}.$$

berechnen kann. Nun liefern k und x_1 zusammen den Term $\phi(N)$. Wie wir bereits in einer Übungsaufgabe gezeigt haben, kann man mit Hilfe von $\phi(N)$ leicht die Faktorisierung von N berechnen.

Der laufzeitbestimmende Schritt ist die Anwendung des Gaußalgorithmus. Damit erhalten wir als Gesamtlaufzeit $\mathcal{O}(\log^2 N)$. □

Die Wiener-Schranke $d \leq N^{\frac{1}{4}}$ wurde 1999 von Boneh und Durfee auf $d \leq N^{0.292}$ verbessert. Dazu benutzt man Gittermethoden zum Lösen nicht-linearer Gleichungen, mit denen wir uns später beschäftigen werden.

7.2 Selektives Fälschen von RSA Unterschriften

Der folgende Angriff auf RSA Unterschriften wurde 1997 von Girault und Misarsky vorgeschlagen. Wie wir bereits gesehen haben, hat das RSA Kryptosystem eine multiplikative

Struktur. Dies führt dazu, dass man leicht selektiv fälschen kann. Sei $m_1 = m_2 m \bmod N$, dann gilt

$$\text{sig}_k(m_1) = m_1^d = (m_2 \cdot m)^d = m_2^d \cdot m^d = \text{sig}_k(m_2) \cdot \text{sig}_k(m) \bmod N.$$

Nun kann man die Unterschrift eines beliebigen $m \in Z_N$ fälschen, indem man m als Quotient der Form $\frac{m_1}{m_2}$ darstellt und die Unterschriften von m_1, m_2 mit Hilfe eines Chosen Message Angriffs ermittelt. Der Quotient der Unterschriften ist dann eine gültige Unterschrift für m. Um für vorgegebenes m die Parameter m_1 und m_2 zu konstruieren, wähle man z.B. ein beliebiges $m_2 \in Z_N^*, m_2 \neq 1$ und berechne $m_1 = m_2 m \bmod N$. (Anmerkung: Wir haben bereits in einer Übungsaufgabe gezeigt, dass sogar das Anfordern einer einzigen Unterschrift genügt, um selektiv zu fälschen.)

Um den oben beschriebenen Angriff zu verhindern, hasht man heutzutage die Nachrichten zuerst bevor man sie unterschreibt. Aber 1978, in der Geburtsstunde von RSA, kannte man noch keine kryptographisch sicheren Hashfunktionen. Daher schlug man zum Verhindern der obigen Chosen Message Attacke ein einfaches Padding mit einem öffentlichen P vor, d.h. $\text{sig}_k = (P||m)^d \bmod N$. Dies können wir auch als $(P + m)^d \bmod N$ schreiben, wobei P von 2^k geteilt wird und $m < 2^k$. Die Frage ist nun, wie groß das Padding P – und damit der Parameter k – sein muss, damit das resultierende Unterschriftenverfahren sicher ist.

Das Problem eines selektiven Fälschers ist nun, für ein vorgegebenes $m < 2^k$ zwei Nachrichten $m_1, m_2 < 2^k$ zu finden mit

$$(P + m_1) = (P + m_2)(P + m) \bmod N.$$

Wir werden nun zeigen, dass man für $m < N^{\frac{1}{2}}$ immer noch eine selektive Fälschung mit Hilfe eines Chosen Message Angriffs durchführen kann. D.h. das Padding muss bereits größer sein als die Hälfte der Bitlänge von N, um diesen Angriff zu verhindern. Dabei werden wir zur Analyse des Angriffs wieder Satz 50 verwenden, d.h. wir wollen den Angriff auf ein SVP zurückführen.

Satz 53 (Girault, Misarsky) *Sei $sig_k(m) = (P + m)^d \bmod N$ eine RSA Unterschrift mit bekanntem Padding P und $m \leq N^{\frac{1}{2}}$. Sei $\mathcal{O}_{sig}(m)$ ein Signier-Orakel, das bei Eingabe einer Nachricht $m < N^{\frac{1}{2}}$ die Unterschrift $sig_k(m)$ ausgibt.*

Man kann eine RSA-Unterschrift einer vorgegebenen Nachricht m effizient selektiv fälschen mit Hilfe von zwei Aufrufen $m_1, m_2 \neq m$ an das Signierorakel.

Beweis: Wie zuvor gezeigt, wollen wir eine Identität der Form

$$(P + m_1) = (P + m_2)(P + m) \bmod N$$

erzeugen. D.h. wir müssen $m_1, m_2 < N^{\frac{1}{2}}$ berechnen, die die obige Gleichung erfüllen. Da es N mögliche Tupel (m_1, m_2) gibt, können wir erwarten, dass ein Tupel die Gleichung modulo N erfüllt.

Wir schreiben unsere Identität wie folgt

$$m_1 - m_2(P+m) = P(P+m-1).$$

Dies ist eine inhomogene lineare Gleichung mit den Unbekannten $m_1, m_2 < N^{\frac{1}{2}}$, d.h. das Produkt $m_1 m_2$ ist kleiner als N. Mit Hilfe des Satzes 50 und seiner Erweiterung aus Übung 51 können wir m_1 und m_2 bestimmen. Dazu müssen wir ein SVP in einem 3-dimensionalen Gitter lösen, was Laufzeit polynomiell in $\log N$ benötigt. □

7.3 Angriff auf GnuPG ElGamal Signaturen

Der folgende Angriff wurde 2004 von Nguyen vorgestellt. Er ist ein Angriff auf die Implementierungen von ElGamal Signaturen im Gnu Privacy Guard, einer Open Source Alternative zu PGP. Der Angriff berechnet aus einer Unterschrift in Polynomialzeit den geheimen Schlüssel a und kann auf einem PC in weniger als einer Sekunde durchgeführt werden.

Erinnern wir uns an das ElGamal Unterschriften-Verfahren aus Kapitel 4.

ElGamal Signaturverfahren

Sei p eine Primzahl (512 Bit), wobei $p-1$ einen großen Primfaktor hat (> 160 Bit). Ferner sei α ein Generator von $\mathbb{Z}_p^*$.
Wir definieren

(1) $\mathcal{P} = \mathbb{Z}_{p-1}$

(2) $\mathcal{U} = \mathbb{Z}_p^* \times Z_{p-1}$

(3) $\mathcal{K} = \{(p, \alpha, \beta, a) \mid \beta = \alpha^a \bmod p, a \in \mathbb{Z}_{p-1}\}$, wobei (p, α, β) der öffentliche Teil des Schlüssels ist und a der geheime Schlüssel.

(4) Signierfunktion: $\text{sig}_k(x) = (\gamma, \delta) = (\alpha^r \bmod p, r^{-1}(x - a\gamma) \bmod p - 1)$ mit $r \in \mathbb{Z}_{p-1}^*$,
Verifikationsfunktion: $\text{ver}_k(x, (\gamma, \delta)) = \begin{cases} \text{wahr} & \text{für } \beta^\gamma \gamma^\delta = \alpha^x \bmod p \\ \text{falsch} & \text{sonst} \end{cases}$

Aus Effizienzgründen wurden im GnuPG der geheime Schlüssel a und der Randomisierungsparameter r so gewählt, dass $a, r \leq p^{\frac{3}{8}}$. Man beachte, dass durch die Wahl kleiner a, r die Exponentiationen α^a und α^r beschleunigt werden kann. Leider bricht aber auch die Sicherheit des Systems mit dieser Parameterwahl komplett zusammen.

Satz 54 (Nguyen) *Gegeben eine ElGamal-Unterschrift*

$$sig_k(x) = (\gamma, \delta) = (\alpha^r \bmod p, r^{-1}(x - a\gamma) \bmod p - 1),$$

wobei $r, a \leq p^{\frac{3}{8}}$. *Dann kann der geheime Schlüssel* a *effizient bestimmt werden.*

Beweis: Betrachten wir die Gleichung

$$r^{-1}(x - a\gamma) = \delta \bmod p - 1$$

mit den Unbekannten a und r. Dies können wir als lineare modulare Gleichung schreiben:

$$\delta r + \gamma a = x \bmod p - 1.$$

Unter der Annahme, dass δ oder γ zu $p-1$ teilerfremd ist, sind die Voraussetzungen von Satz 50 mit seiner inhomogenen Erweiterung aus Übung 51 erfüllt. Das Produkt ar ist nach Voraussetzung kleiner als $p^{\frac{3}{4}} << p - 1$. Die Lösung eines SVPs in einem 3-dimensionalen Gitter liefert uns die geheimen Parameter r und a in Zeit polynomiell in $\log p$. □

7.4 Angriff auf den Pollard-Generator

In der Informatik benötigt man für viele Anwendungen Zufallszahlen. Da echte Zufallszahlen schwer zu erhalten sind, setzt man in der Praxis sogenannte *Generatoren für Pseudo-Zufallszahlen* ein. Diesen Generatoren gibt man als Eingabe eine Zahl x_0, die Saat (engl: seed), und sie erzeugen durch iterierte Anwendung einer Funktion f eine deterministische Folge von Zahlen $x_1, x_2, \ldots$, die möglichst zufällig sein soll. Z.B. verwendet die C-Funktion $rand()$ die Funktion

$$x_{i+1} = f(x_i) = ax_i + b \bmod N,$$

für bekanntes N und geheime Parameter a, b. Da die Funktion f hier eine lineare Funktion ist, nennt man solche Generatoren auch *lineare Kongruenzgeneratoren.*

Nun ist eine der Anforderungen an einen Pseudo-Zufallszahlengenerator, dass man von einer Ausgabe $x_1, \ldots, x_n$ nicht auf die $(n+1)$-te Ausgabe x_{n+1} schließen kann. Man kann leicht sehen, dass man beim obigen Generator aus x_1, x_2, x_3 leicht den Wert x_4 berechnen kann.

Übung 55 *Bei Verwendung des linearen Kongruenzgenerators* $x_{i+1} = ax_i + b \bmod N$ *kann aus* x_1, x_2, x_3 *effizient der Wert* x_4 *berechnet werden.*

Um zu verhindern, dass man aus einer Folge von Ausgabewerten effizient weitere Folgenelemente berechnen kann, gibt man beim linearen Kongruenzgenerator nicht die kompletten x_i sondern nur einen Bruchteil der Bits der x_i aus. 1988 zeigten Frieze, Hastad, Kannan, Lagarias und Shamir mit Hilfe von Gittermethoden, dass auch dies nicht zu einem sicheren Generator führt.

Da also lineare Kongruenzen nicht zur Erzeugung von geeigneten Zufallszahlen geeignet sind, verwendet man heute oft nicht-lineare Generator. Ein bekannter Vertreter dieser Gattung ist der *Pollard-Generator*, der durch die Rekurrenz

$$x_{i+1} = f(x_i) = x_i^2 + c \bmod p$$

bestimmt ist. Hierbei setzen wir voraus, dass p eine öffentlich bekannte Primzahl und c geheim ist. Als Übungsaufgabe kann man leicht zeigen, dass man auch beim Pollard-Generator die x_i nicht komplett ausgeben darf. Die Frage ist nun, welchen Bruchteil der x_i man ausgeben darf.

Das folgende Resultat von Blackburn, Gomez-Perez, Gutierrez und Shparlinski von 2005 zeigt, dass man weniger als $\frac{3}{4}$ der obersten Bits der x_i ausgeben darf. Ansonsten können die x_i komplett rekonstruiert werden und damit auch der geheime Parameter c.

Satz 56 *Sei r_1, r_2, r_3 eine Ausgabe des Pollard-Generators mit*

$$|x_i - r_i| \leq \frac{1}{2} p^{\frac{1}{4}}.$$

Dann können die Folgenwerte x_1, x_2, x_3 und damit auch das geheime c effizient berechnet werden.

Beweis: Sei $x_i = r_i + y_i$ mit unbekannten $|y_i| \leq \frac{1}{2} p^{\frac{1}{4}}$. Unser Ziel ist es, die unbekannten y_i zu rekonstruieren. Aus der Definition des Pollard-Generators gilt

$$x_2 = x_1^2 + c \bmod p \quad \text{und} \quad x_3 = x_2^2 + c \bmod p.$$

Die Differenz beider Gleichungen liefert

$$x_2 - x_3 = x_1^2 - x_2^2 \bmod p.$$

Wir schreiben dies als

$$r_2 + y_2 - r_3 - y_3 = (r_1 + y_1)^2 - (r_2 + y_2)^2 \bmod p.$$

Nun bringen wir alle unbekannten Terme auf die linke Seite

$$(y_2^2 - y_1^2 + y_2 - y_3) + 2r_2 y_2 - 2r_1 y_1 = r_1^2 - r_2^2 + r_3 - r_2 \bmod p.$$

Linearisierung liefert eine lineare, inhomogene modulare Gleichung

$$z + 2r_2y_2 - 2r_1y_1 = c \bmod p,$$

mit $z = y_2^2 - y_1^2 + y_2 - y_3$ und konstantem Term $c = r_1^2 - r_2^2 + r_3 - r_2$. Da der Koeffizient von z Eins ist (und daher teilerfremd zu p), können wir Satz 50 mit seiner Erweiterung aus Übung 51 anwenden, sofern der Absolutbetrag des Produkts zy_1y_2 kleiner als der Modul p ist. Für $|y_1|, |y_2|$ kennen wir schon die obere Schranke $Y = \frac{1}{2}p^{\frac{1}{4}}$. Für z gilt

$$|z| \leq |y_2^2 - y_1^2| + |y_2 - y_3| \leq \frac{1}{2}p^{\frac{1}{2}} + p^{\frac{1}{4}} \leq p^{\frac{1}{2}}.$$

Damit können wir $Z = p^{\frac{1}{2}}$ als obere Schranke für z definieren. Es gilt $YZ < p$ und daher können wir (heuristisch) die Unbekannten z, y_1, y_2 mit Hilfe eines SVPs in einem 4-dimensionalen Gitter bestimmen. Aus y_1, y_2 erhält man x_1, x_2 und damit

$$c = x_2 - x_1^2 \bmod p.$$

□

8 Lösen polynomieller modularer Gleichungen

In Kapitel 6 haben wir eine heuristische Methode zum Lösen linearer Gleichungen kennengelernt. In Kapitel 7 haben wir vier verschiedene Anwendungen der Methode kennengelernt. Dabei haben wir stets eine Gleichung aufgestellt, diese bei Bedarf linearisiert und dann die Methode aus Kapitel 6 verwendet. Bei dieser Methode haben wir bei unseren Angriffen das Lösen der Gleichung auf ein SVP in einem Gitter mit konstanter Dimension zurückgeführt (heuristische Methode). Für einige der Angriffe aus Kapitel 7 hat man nicht nur die experimentelle Evidenz, dass die Heuristik in der Praxis sehr gut funktioniert, sondern man kann die Angriffe auch beweisbar machen (z.B. unter Annahmen an die Verteilung der Koeffizienten in der linearen Gleichung).

8.1 Lösen univariater Polynomgleichungen

In diesem Abschnitt wollen wir untersuchen, unter welcher Bedingung wir für den Fall von polynomiellen modularen Gleichungen eine Lösung erhalten können. Sei

$$f(x) = a_n x^n + a_{n_1} x^{n-1} + \cdots + a_1 x + a_0$$

ein univariates Polynom (d.h. ein Polynom in einer Variable) vom Grad n. Die Terme x^i bezeichnen wir als *Monome von* $f(x)$. Für ein gegebenes $M \in \mathbb{Z}$ mit unbekannter Faktorisierung wollen wir nun die Nullstellen von $f(x)$ modulo M bestimmen. Ferner werden wir im folgenden stets annehmen, dass $f(x)$ monisch ist, d.h. dass $a_n = 1$. Dies kann man gegebenenfalls dadurch erreichen, indem man das Polynom mit dem Inversen von a_n^{-1} modulo M multipliziert (Übung: Was passiert, wenn das Inverse nicht existiert?).

Man beachte, dass das Problem des Lösens modularer Polynomgleichungen für die Kryptographie sehr bedeutend ist. Sei c ein Chiffretext, der mit einem öffentlichen RSA-Schlüssel (N, e) generiert wurde. Dann liefert die Nullstelle des Polynoms

$$f(x) = x^e - c \bmod N$$

das eindeutige m mit $m^e = c \bmod N$, d.h. den zugrundeliegenden Klartext.

Wenn man die Methode aus Kapitel 6 verwendet erhält man als Bedingung für die Größe der Nullstelle

$$\prod_{i=1}^{n} x^i = x^{\frac{n(n+1)}{2}} \leq M \quad \Leftrightarrow \quad x \leq M^{\frac{2}{n(n+1)}}.$$

Damit erhalten wir eine heuristische Methode, die das SVP in einem $(n+1)$-dimensionalen Gitter löst. Wir wollen nun zeigen, dass man eine verbesserte Schranke $x \leq M^{\frac{1}{n}}$ mit einer modifizierten Gittermethode erreichen kann, die zwei weitere Vorzüge bietet:

(a) Wir brauchen kein SVP in einem $(n+1)$-dimensionalen Gitter zu lösen. Uns genügt die Approximationsgüte des L^3-Algorithmus. Damit benötigt die Methode auch für *nicht-konstante* n nur polynomielle Laufzeit.

(b) Die Methode ist *beweisbar*. Sie liefert *alle* Nullstellen des Polynoms $f(x)$ modulo M, die hinreichend klein sind.

Die folgende Methode zum Lösen univariater Polynomgleichung wurde 1996 von Coppersmith vorgestellt. Sei wird auch als Coppersmith-Methode bezeichnet. Wir geben hier zunächst die zugrundeliegende Idee der Coppersmith-Methode wieder.

Idee der Coppersmith-Methode

Gegeben sei ein Polynom $f(x) \in Z[x]$ und $M \in \mathbb{Z}$. Gesucht sind die Lösungen von

$$f(x_0) = 0 \bmod M$$

für alle $|x_0| \leq X$. Unser Ziel ist es, die Schranke X zu maximieren.

- Wähle ein geeignetes $m \in \mathbb{N}$. Definiere eine Kollektion von Polynomen $f_1(x), f_2(x), \ldots, f_k(x)$, die alle die kleinen Nullstellen $|x_0| \leq X$ modulo M^m besitzen. *Beispiel:* Wähle $f_i(x) = x^i f^m(x)$.

- Konstruiere ein Polynom $g(x) = \sum_{i=1}^{k} a_i f_i(x)$, $a_i \in \mathbb{Z}$ mit der Eigenschaft

  $$g(x_0) = 0 \quad \text{über } \mathbb{Z}$$

  für alle $|x_0| \leq X$. Dazu genügt es, die folgende hinreichende Bedingung zu erfüllen:

  $$|g(x_0)| < M^m.$$

  Sei x_0 eine beliebige Nullstelle mit $|x_0| \leq X$. Da alle $f_i(x)$ die Nullstelle x_0 modulo M^m besitzen, gilt auch

  $$g(x_0) = a_1 f_1(x_0) + \cdots + a_k f_k(x_0) = a_1 \cdot 0 + \cdots + a_k \cdot 0 = 0 \bmod M^m.$$

  Falls aber $g(x_0) = 0 \bmod M^m$ und $|g(x_0)| < M^m$, dann ist $g(x_0) = 0$ (ohne modulo M^m, das gilt in den natürlichen Zahlen!) wie gewünscht.

- Finde die Nullstellen von $g(x)$ über den ganzen Zahlen mit Standardmethoden (z.B. Sturm'sche Ketten oder Newton-Iteration).

Die Coppersmith-Methode ist eine Reduktion vom modularen Fall auf den ganzzahligen Fall; sie führt das Lösen von modularen Polynomgleichungen auf das Lösen ganzzahliger Polynomgleichungen zurück (wofür man effiziente Algorithmen hat).

Wir haben gesehen, dass man ein Polynom $g(x)$ als ganzzahlige Linearkombination anderer Polynome $f_i(x)$ konstruieren muss, so dass $|g(x_0)| < M^m$ gilt. Diese hinreichende Bedingung werden wir nun als eine Bedingung an einen Gittervektor formulieren.

Dazu stellen wir Polynome $g(x) = a_n x^n + \ldots a_0 x^0$ in Form ihres Koeffizientenvektoren $(a_n, \ldots, a_0)$ dar. Die Norm eines Polynoms ist dann als die Norm des Koeffizientenvektors definiert.

Definition 57 *Sei $g(x) = \sum_i a_i x^i \in \mathbb{Z}[x]$. Dann ist die* Norm von g *definiert als*

$$\|g(x)\| = \sqrt{\sum_i a_i^2}.$$

Der folgende Satz von Howgrave-Graham beschreibt eine hinreichende Bedingung an die Norm von g, damit $g(x_0) = 0$ gilt.

Satz 58 (Howgrave-Graham) *Sei $g(x) = \sum_i b_i x^i \in Z[x]$ ein Polynom mit n Monomen. Es sei ferner*

(1) $g(x_0) = 0 \bmod M^m$ *für* $|x_0| \leq X$ *und*

(2) $\|g(xX)\| < \frac{M^m}{\sqrt{n}}$.

Dann gilt $g(x_0) = 0$ über den ganzen Zahlen.

Beweis: Es gilt

$$\begin{aligned} |g(x_0)| &= \left|\sum_i b_i x_0^i\right| \leq \sum_i \left|b_i X^i \left(\frac{x_0}{X}\right)^i\right| \\ &\leq \sum_i \left|b_i X^i\right| \leq \sqrt{n} \cdot \|g(xX)\| < M^m. \end{aligned}$$

Damit ist $|g(x_0)| < M^m$ und wegen Bedingung (1) auch $g(x_0) = 0 \bmod M^m$. Daraus folgt $g(x_0) = 0$. □

Satz 59 (Coppersmith 1996) *Sei $\epsilon > 0$ beliebig. Für hinreichend große $M \in \mathbb{N}$ gilt: Sei $f(x)$ ein monisches Polynom mit Grad n. Dann kann man alle Nullstellen x_0 mit*

$$f(x_0) = 0 \bmod M \quad \text{und} \quad |x_0| \leq M^{\frac{1}{n} - \epsilon}$$

in Zeit polynomiell in $\log M, n$ und $\frac{1}{\epsilon}$ finden.

Beweis: Wir werden nun die zuvor beschriebene Coppersmith Methode Schritt für Schritt anwenden.

Zunächst fixieren wir ein $m \in \mathbb{N}$ (eine exakte Analyse zeigt, dass $m = \lceil \frac{1}{n\epsilon} \rceil$ eine geeignete Wahl ist). Als nächstes müssen wir eine Kollektion von Polynomen angeben, die alle die Nullstellen x_0 von $f(x)$ modulo M^m besitzen (anstatt modulo M). Dazu definieren wir

$$f_{i,j}(x) = M^{m-i} x^j f^i(x) \quad \text{für } i = 0, \ldots, m-1 \text{ and } j = 0, \ldots, n-1$$

Falls $f(x_0) = 0 \bmod M$, dann folgt für jedes der $f_{i,j}$ dass $f_{i,j}(x_0) = 0 \bmod M^m$. Dies gilt wegen $f^i(x_0) = 0 \bmod M^i$ und daher $M^{m-i} f^i(x_0) = 0 \bmod M^m$.

Daher gilt für jede ganzzahlige Linearkombination $g(x) = \sum_{i,j} a_{i,j} f_{i,j}(x) = \sum_k b_k x^k$ ebenfalls $g(x_0) = 0 \bmod M^m$. D.h. die Bedingung (1) aus Satz 58 ist erfüllt. Nun wollen wir mit Gitterreduktion eine Linearkombination $g(x)$ finden, die ebenfalls Bedingung (2) erfüllt.

Sei X eine obere Schranke für die Nullstellen x_0 von $f(x)$. Dazu betrachten wir die Koeffizientenvektoren der Polynome $f_{i,j}(xX)$ und die Polynome in der folgenden Reihenfolge

$$\begin{array}{ccc} f_{0,0}(xX), f_{0,1}(xX), & \ldots & , f_{0,n-1}(xX), \\ f_{1,0}(xX), f_{1,1}(xX), & \ldots & , f_{1,n-1}(xX), \\ \vdots & \vdots & \vdots \\ f_{m-1,0}(xX), f_{m-1,1}(xX), & \ldots & , f_{m-1,n-1}(xX) \end{array}$$

Man beachte, dass wir die Polynome in der Reihenfolge aufsteigenden Grads geordnet haben, beginnend bei $f_{0,0}(xX)$ mit Grad 0 und abschließend mit $f_{m-1,n-1}(xX)$ mit Grad $(m-1)n + n - 1 = mn - 1$. Ordnen wir die Koeffizientenvektoren in einer Basismatrix B eines Gitters L an, so hat diese Basismatrix Dreiecksgestalt:

$$\begin{pmatrix} M^m & & & & & & & & & \\ & M^m X & & & & & & & & \\ & & \ddots & & & & & & & \\ & & & M^m X^{n-1} & & & & & & \\ & \ddots & \ddots & \ddots & \ddots & & & & & \\ - & - & \ldots & - & \ldots & MX^{(m-1)n} & & & & \\ & - & \ldots & - & \ldots & - & MX^{(m-1)n+1} & & & \\ & & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & & \\ & & & - & \ldots & - & - & \ldots & MX^{mn-1} \end{pmatrix}.$$

Da unsere Kollektion mn Polynome hat, folgt $\dim(L) = mn$. Weiterhin gilt

$$\det(L) = \prod_{i=1}^{m} M^{in} \prod_{i=0}^{mn-1} X^i = M^{\frac{m(m+1)n}{2}} X^{\frac{(mn-1)mn}{2}} \approx M^{\frac{m^2n}{2}} X^{\frac{m^2n^2}{2}}.$$

Man beachte, dass wir bei der letzten Approximation Terme niederer Ordnung weggelassen haben. Diese Terme fließen in den Fehlerterm ϵ ein.

Nun wenden wir den L^3-Reduktionsalgorithmus auf unsere Basis B an. Gemäß Satz 49 liefert dieser uns einen Vektor v mit

$$\|v\| \leq c^{\dim(L)} \det(L)^{\frac{1}{\dim(L)}}.$$

in Zeit polynomiell in m, n und $\log M$.

Man beachte, dass der Vektor v eine Linearkombination von Koeffizientenvektoren der $f_{i,j}(xX)$ ist, d.h. v ist der Koeffizientenvektor eines Polynoms $g(xX)$ mit $\dim(L)$ vielen Monomen. Nach Satz 58 benötigen wir, dass $\|v\| = \|g(xX)\| \leq \frac{M^m}{\sqrt{\dim(L)}}$.

Kombinieren wir dies mit unserer Schranke aus dem L^3-Algorithmus, so müssen wir die Bedingung

$$c^{\dim(L)} \det(L)^{\frac{1}{\dim(L)}} \leq \frac{M^m}{\sqrt{\dim(L)}}$$

erfüllen. Da die Faktoren $c^{\dim(L)}$ und $\sqrt{\dim(L)}$ nicht von M abhängen, können wir sie für hinreichend große M vernachlässigen.
Damit erhalten wir eine vereinfachte Bedingung

$$\det(L) \leq M^{m \dim(L)}.$$

Einsetzen von $\dim(L)$ und $\det(L)$ liefert

$$M^{\frac{m^2n}{2}} X^{\frac{m^2n^2}{2}} \leq M^{m^2n} \;\Leftrightarrow\; X^{\frac{m^2n^2}{2}} \leq M^{\frac{m^2n}{2}} \;\Leftrightarrow\; X \leq M^{\frac{1}{n}}.$$

Unter Berücksichtigung des Fehlerterms ϵ, erhalten wir eine obere Schranke von $|x_0| \leq X = M^{\frac{1}{n}-\epsilon}$ für die gesuchten kleinen Nullstellen von $f(x)$.

Die Laufzeit wird von der L^3-Reduktion der Gitterbasis B bestimmt. Da B Dimension mn hat und die Bitgrößen der Einträge in B durch $\mathcal{O}(mn \log M)$ beschränkt sind, ist die L^3-Laufzeit nach Satz 49 polynomiell in $\log M, m$ und n. Man beachte, dass m selbst polynomiell in $\frac{1}{\epsilon}$ ist. □

Durch eine exakte aber aufwändige Analyse des Fehlerterms ϵ in Satz 59 kann gezeigt werden, das man mit Hilfe elementarer Brute-Force Techniken sogar auf ϵ ganz verzichten kann. Dabei erhöht sich der Aufwand in der Laufzeit nur um einen konstanten Faktor.

Satz 60 (Coppersmith 1996) *Sei $f(x)$ ein monisches Polynom mit Grad n. Dann kann man alle Nullstellen x_0 mit*

$$f(x_0) = 0 \bmod M \quad \text{und} \quad |x_0| \leq M^{\frac{1}{n}}$$

in Zeit polynomiell in $\log M$ und n finden.

Im folgenden werden wir diese vereinfachte Variante des Satzes von Coppersmith anwenden.

8.2 Anwendungen der Coppersmith-Methode

Wir werden in diesem Abschnitt zwei Anwendungen von Satz 60 bei Angriffen auf das RSA Kryptosytem kennenlernen:

- Angriff auf stereotype Nachrichten
- Angriff auf RSA mit zufälligem Padding

8.2.1 Angriff auf stereotype Nachrichten

Beim Angriff auf stereotype RSA Nachrichten nimmt man an, dass einem Angreifer ein Teil des zugrundeliegenden Klartextes bekannt ist. Ziel des Angreifers ist es, den unbekannten Rest effizient zu berechnen. Nehmen wir an, ein Angreifer weiß, dass die verschlüsselte Nachricht mit der folgenden stereotypen Phrase beginnt: „Guten Morgen. Ihr Passwort für den heutigen Tag lautet:“. Danach folgt jeweils das dem Angreifer unbekannte Tagespasswort x. D.h. der RSA-Chiffretext ist von der Form

$$c = (S + x)^e \bmod N,$$

wobei S der bekannte stereotype Teil der Nachricht ist.

Der folgende Satz ist eine direkte Anwendung des Satzes von Coppersmith 60.

Satz 61 *Sei $m = S + x$ eine stereotype Nachricht mit bekanntem S. Dann kann x aus dem RSA-Chiffretext $c = m^e \bmod N$ in Zeit polynomiell in $\log N$ und e berechnet werden, sofern*

$$|x| \leq N^{\frac{1}{e}}.$$

Beweis: Ein Angreifer muss die Nullstellen des Polynoms

$$f(x) = (x + S)^e - c \bmod N$$

finden. Dies ist ein monisches Polynom vom Grad e. Mit Hilfe von Satz 60 kann die Nullstelle dieses Polynoms in polynomieller Zeit gefunden werden, sofern $|x| \leq N^{\frac{1}{e}}$. □

8.2.2 RSA mit zufälligem Padding

Der folgende Angriff wurde von Franklin und Reiter 1996 vorgestellt. Angenommen zwei Nachrichten m und m' erfüllen eine affine Relation

$$m' = m + r \bmod N$$

mit *bekanntem* r.

Seien c und c' die jeweiligen RSA-Verschlüsselungen von m, m' mit öffentlichem Exponenten 3, d.h.

$$\begin{aligned} c &= m^3 \bmod N \text{ und} \\ c' &= (m+r)^3 = m^3 + 3m^2r + 3mr^2 + r^3 \bmod N \end{aligned}$$

Wir erhalten als nette Knobelaufgabe:

Übung 62 *Zeigen Sie, dass man m mit Hilfe von c, c', r und N effizient berechnen kann.* Hinweis: *Die Lösung verwendet nur elementare Arithmetik (Addition, Subtraktion, Multiplikation, Division) modulo N.*

Was passiert, wenn r *unbekannt aber klein* ist. Können wir dann aus c und c' immer noch m effizient berechnen? Wir werden nun einen Angriff kennenlernen, der r bestimmt sofern r hinreichend klein ist. Damit kann anschließend der Algorithmus aus Übung 62 verwendet werden, um m zu ermitteln.

Ein solcher Angriff hat direkte Auswirkung auf die Sicherheit von RSA Nachrichten, bei denen man zur Randomisierung der Nachricht einen zufälligen Bitstring R anhängt (sogenanntes Random Padding). Wie schon in vorigen Kapiteln gesehen, kann ein Anhängen von Zufallsbits manche Angriffe auf RSA verhindern, daher wurde das Random Padding in der Praxis häufig eingesetzt.

Sei das zufällige Padding R ein k-Bit String. Wir wollen eine Nachricht M verschicken. Dazu shiften wir M um k Stellen nach links und hängen R an, d.h. unser Klartext ist von der Form $m = M2^k + R$. Verschlüsseln wir dies mit dem öffentlichen Exponenten 3 so erhalten wir

$$c = m^3 = (M2^k + R)^3 \bmod N.$$

Angenommen, man verschlüsselt dieselbe Nachricht M nochmal, aber mit einem anderen Padding $R' = R + r$. Dann sind wir in derselben Situation wie zuvor, d.h. wir haben ein m' von der Form $m' = M2^k + R' = m + r$.

Satz 63 *Seien $c = m^3 \bmod N$ und $c' = (m+r)^3 \bmod N$ zwei RSA-Chiffretexte. Dann kann man m in Zeit polynomiell in $\log N$ berechnen, sofern*

$$|r| \leq N^{\frac{1}{9}}.$$

Beweis: Aus den beiden Gleichungen

$$\begin{aligned} m^3 - c &= 0 \bmod N \\ (m+r)^3 - c' &= 0 \bmod N \end{aligned}$$

kann man das unbekannte m mit Hilfe einer Resultantenberechnung eliminieren. Auf die Theorie der Resultanten wollen wir an dieser Stelle nicht näher eingehen, daher geben wir nur das Resultat der Berechnung an. Resultanten werden häufig eingesetzt, um Unbekannte in Gleichungssystemen zu eliminieren.

$$\mathrm{res}_m(m^3 - c, (m+r)^3 - c') = r^9 + (3c - 3c')r^6 + 3c'^2 r^3 + (c - c')^3 \bmod N$$

Die Resultante liefert uns also ein univariates monisches Polynom $f(r)$ vom Grad 9. Mit Hilfe des Satzes von Coppersmith (Satz 60) können wir die Nullstelle r bestimmen, sofern $|r| \leq N^{\frac{1}{9}}$. Die Laufzeit für diesen Schritt ist polynomiell in $\log N$.

Haben wir aber r erfolgreich bestimmt, so können wir das Ergebnis aus Übung 62 verwenden, um die Nachricht m effizient zu berechnen. ◻

9 Lösen von Polynomgleichungen modulo Teilern

Im letzten Kapitel haben wir gesehen, dass man alle Nullstellen x_0 eines Polynoms $f(x)$ mit Grad n modulo M bestimmen kann, sofern $|x_0| \leq M^{\frac{1}{n}}$. Hierbei ist M ein Modul unbekannter Faktorisierung. Sei nun b ein Teiler von M, wobei b nicht notwendigerweise eine Primzahl sein muss. In diesem Kapitel wollen wir alle Nullstellen x_0 eines Polynoms

$$f(x) \bmod b$$

bestimmen. Dies erscheint zunächst widersinnig, da wir die Faktorisierung von M nicht kennen, also gar nicht modulo b rechnen können. Der Trick besteht darin, dass Gleichungen modulo M auch modulo b gelten, da b ein Teiler von M ist. Wir können also ausnutzen, dass wir ein Vielfaches M von b kennen.

Zudem finden wir mit Hilfe der Nullstellen modulo des unbekannten Faktors b i. allg. auch eine nichttriviale Faktorisierung von M. D.h. zu Beginn des Algorithmus kennen wir die Faktoren von M zwar nicht, aber unser Algorithmus kann dazu verwendet werden, M zu zerlegen.

Übung 64 *Sei $M \in \mathbb{N}$ mit unbekanntem Teiler b und $f(x) \in \mathbb{Z}[x]$ mit Grad n. Sei A ein Algorithmus, der bei Eingabe M und $f(x)$ eine Nullstellen x_0 von $f(x)$ modulo b berechnet, die keine Nullstelle von $f(x)$ modulo M ist, d.h.*

$$f(x_0) = 0 \bmod b \quad \textit{und} \quad f(x_0) \neq 0 \bmod M.$$

Dann kann man einen nicht-trivialen Faktor von M in Zeit polynomiell in n und $\log M$ bestimmen.

Da gemäß obiger Übung das Finden von Nullstellen modulo eines Teilers von M einen polynomiellen Faktorisierungsalgorithmus für M liefert, können wir nicht erwarten, alle Nullstellen eines Polynoms zu finden. Betrachten wir z.B. einen RSA-Modul $N = pq$, dann sind die Nullstellen des Polynoms

$$f(x) = x \bmod p$$

gerade die Vielfachen von p. Könnten wir also eine der Nullstellen kp mit $0 < k < q$ finden, so könnten wir anschließend in Polynomialzeit den RSA-Modul faktorisieren.

D.h. wir können nicht erwarten, Nullstellen beliebiger Größe effizient berechnen zu können, daher wollen wir uns wie in den Kapiteln zuvor mit dem Finden kleiner Nullstellen begnügen. Unsere Strategie zum Finden der kleinen Nullstellen wird die schon in Kapitel 8 vorgestellte gitterbasierte Coppersmith-Methode sein. Diesmal werden wir uns allerdings zunächst auf univariate Polynome vom Grad 1 beschränken und für diese eine Schranke X beweisen, bis zu der man effizient alle Nullstellen $|x_0| \leq X$ bestimmen kann.

In einer Übung zeigen wir zunächst, dass es genügt, sich auf monische Polynome zu beschränken.

Übung 65 *Sei $M \in \mathbb{N}$ mit unbekanntem Teiler b. Sei A ein Algorithmus, der bei Eingabe M und eines monischen Polynoms $f(x) \in \mathbb{Z}[x]$ die Nullstellen x_0 von $f(x)$ modulo b berechnet. Dann kann man daraus einen effizienten Algorithmus konstruieren, der entweder*

(a) die Nullstellen eines beliebigen (nicht-monischen) Polynoms $f'(x) \in \mathbb{Z}[x]$ oder

(b) einen nicht-trivialen Faktor von M

berechnet.

Wir zeigen nun eine Schranke für die Größe der Nullstellen, die wir noch mit unserem Algorithmus berechnen können. Diese Schranke hängt von der Größe des Teiler b ab. Sei $b \geq M^\beta$ $(0 < \beta \leq 1)$, dann können wir alle Nullstellen x_0 bis zur Größe $|x_0| \leq N^{\beta^2}$ berechnen. D.h. je größer unser Teiler ist, desto größer ist auch ist die Schranke für die Nullstellen.

Satz 66 *Sei $\epsilon > 0$ beliebig. Für hinreichend große $M \in \mathbb{N}$ gilt: Sei b ein Teiler von M mit $b \geq M^\beta$, $0 < \beta \leq 1$. Ferner sei $f(x) = x + a$. Dann kann man alle Nullstellen x_0 mit*

$$f(x_0) = 0 \bmod b \quad \text{und} \quad |x_0| \leq M^{\beta^2 - \epsilon}$$

in Zeit polynomiell in $\log M$, $\frac{1}{\beta}$ und $\frac{1}{\epsilon}$ finden.

Beweis: Wähle $m \in \mathbb{N}$ geeignet (eine exakte Analyse zeigt, dass $m = \left\lceil \frac{\beta^2}{\epsilon} \right\rceil$ eine geeignete Wahl ist). Wir definieren nun die folgende Kollektion von Polynomen:

$$\begin{aligned} f_i(x) &= M^{m-i} f^i(x) \quad \text{für } i = 0, \ldots, m \\ f_i(x) &= x^{i-m} f^m(x) \quad \text{für } i = m+1, \ldots, \frac{1}{\beta} m - 1. \end{aligned}$$

Falls $\frac{1}{\beta} m$ keine ganze Zahl ist, dann runden wir auf die nächste ganze Zahl. Man beachte, dass unsere $f_i(x)$ eine geeignete Kollektion für die Coppersmith-Methode darstellen: Falls $f(x_0) = 0 \bmod b$, dann ist $f_i(x_0) = 0 \bmod b^m$, denn $M^{m-i} = 0 \bmod b^{m-i}$. Damit erfüllt

jede ganzzahlige Linearkombination $g(x) = \sum_i b_i f_i(x)$ Bedingung (1) aus dem Satz von Howgrave-Graham (Satz 58). Wir wollen nun ein Polynom $g(x)$ konstruieren, das auch Bedingung (2) erfüllt, d.h. dass der Koeffizientenvektor von $g(xX)$ hinreichend kleine Norm hat.

Sei X eine obere Schranke an die Nullstellen von $f(x)$. Wir wollen zeigen, dass $X = M^{\beta^2-\epsilon}$. Man beachte, dass $f_i(x)$ ein Polynom vom Grad i ist. Wenn wir die Koeffizientenvektoren von $f_0(xX), f_1(xX), \ldots, f_{\frac{1}{\beta}m-1}(xX)$ als Zeilenvektoren einer Basismatrix B schreiben, so hat die resultierende Basismatrix daher untere Dreiecksgestalt.

$$B = \begin{pmatrix} M^m & & & & & & \\ - & M^{m-1}X & & & & & \\ - & - & M^{m-2}X^2 & & & & \\ \vdots & \ddots & \ddots & \ddots & & & \\ - & - & - & \ldots & X^m & & \\ & - & - & \ldots & - & X^{m+1} & \\ & & \ddots & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & \ddots & X^{\frac{1}{\beta}m-1} \end{pmatrix}$$

Das Gitter L, das von den Zeilenvektoren von B aufgespannt wird, hat Dimension $\dim(L) = \frac{1}{\beta}m$. Seine Determinante ist

$$\det(L) = \prod_{i=1}^{m} M^i \prod_{i=1}^{\frac{1}{\beta}m-1} X^i = M^{\frac{m(m+1)}{2}} X^{\frac{\left(\frac{1}{\beta}m-1\right)\left(\frac{1}{\beta}m\right)}{2}} \approx M^{\frac{m^2}{2}} X^{\frac{m^2}{2\beta^2}}.$$

Bei der letzten Umformung vernachlässigen wir Terme niederer Ordnung; diese gehen in den Fehlerterm ϵ ein.

Wenden wir den L^3-Algorithmus auf unsere Basis B an, so erhalten wir einen Vektor v, der dem Koeffizientenvektor eines Polynoms $g(xX)$ entspricht. Nach dem L^3-Satz (Satz 49) gilt, dass

$$\|v\| = \|g(xX)\| \leq c^{\dim(L)} \det(L)^{\frac{1}{\dim(L)}}.$$

Wir benötigen gemäß Bedingung (2) aus dem Satz von Howgrave-Graham, dass $\|g(xX)\| \leq \frac{b^m}{\sqrt{\dim(L)}}$. Zusammen mit der Schranke des L^3-Algorithmus erhalten wir die Bedingung

$$c^{\dim(L)} \det(L)^{\frac{1}{\dim(L)}} \leq \frac{b^m}{\sqrt{\dim(L)}}.$$

Da die Terme $c^{\dim(L)}$ und $\sqrt{\dim(L)}$ nicht vom Parameter M abhängen, vernachlässigen wir sie im folgenden. Für hinreichend große M erhalten wir daher die vereinfachte Bedingung

$$\det(L) \leq b^{m \dim(L)}.$$

Diese Bedingung können wir aber nicht überprüfen, da wir den Teiler b nicht kennen. Wir nutzen nun, dass $b \geq M^\beta$. Damit erhalten wir eine neue Bedingung

$$\det(L) \leq M^{\beta m \dim(L)}.$$

Man beachte: Wenn die neue Bedingung erfüllt ist, dann ist auch die Bedingung $\det(L) \leq b^{m \dim(L)}$ erfüllt. D.h. die neue Bedingung ist strikter. Bei dieser Bedingung sind nun alle Parameter bekannt und wir können die Terme für $\dim(L)$ und $\det(L)$ einfach einsetzen:

$$M^{\frac{m^2}{2}} X^{\frac{m^2}{2\beta^2}} \leq M^{\beta m \cdot \frac{1}{\beta} m} \quad \Leftrightarrow \quad M X^{\frac{1}{\beta}^2} \leq M^2 \quad \Leftrightarrow \quad X \leq M^{\beta^2}.$$

D.h. unter Einbeziehung des Fehlerterms ϵ können wir alle Nullstellen x_0 bestimmen mit $|x_0| \leq X = M^{\beta^2 - \epsilon}$.

Der laufzeitbestimmende Schritt ist die L^3-Reduktion der $\frac{1}{\beta}m$-dimensionalen Basis B mit Einträgen, deren Bitlänge durch $\mathcal{O}(\frac{1}{\beta} m \log M)$ beschränkt ist. D.h. die Laufzeit ist polynomiell in $\log M, m$ und $\frac{1}{\beta}$. Ferner ist m polynomiell in $\frac{1}{\epsilon}$. □

Mittels einer exakten Analyse des Fehlerterms ϵ und zusätzlicher Tricks kann man zeigen, dass man dieselbe Schranke auch ohne Fehlerterm in polynomieller Laufzeit erhalten kann.

Satz 67 *Sei b ein Teiler von M mit $b \geq M^\beta$, $0 < \beta \leq 1$. Ferner sei $f(x) = x + a$. Dann kann man alle Nullstellen x_0 mit*

$$f(x_0) = 0 \bmod b \quad \text{und} \quad |x_0| \leq M^{\beta^2}$$

in Zeit polynomiell in $\log M$ und $\frac{1}{\beta}$ finden.

Analog zu Satz 60 kann man das Lösen von Polynomgleichungen modulo Teilern auch für Polynome mit allgemeinem Grad n durchführen und erhält dann eine Schranke von $M^{\frac{\beta^2}{n}}$.

Satz 68 *Sei b ein Teiler von M mit $b \geq M^\beta$, $0 < \beta \leq 1$. Ferner sei $f(x) \in \mathbb{Z}[x]$ ein monisches Polynom mit Grad n. Dann kann man alle Nullstellen x_0 mit*

$$f(x_0) = 0 \bmod b \quad \text{und} \quad |x_0| \leq M^{\frac{\beta^2}{n}}$$

in Zeit polynomiell in $\log M$, n und $\frac{1}{\beta}$ finden.

9.1 Anwendungen von Polynomgleichungen modulo Teilern

Wir werden hier drei Anwendungen von Polynomgleichungen modulo Teilern kennenlernen. Dies sind:

- Faktorisierung des RSA-Moduln $N = pq$ mit bekannten Bits von p.
- Faktorisierung des RSA-Moduls $N = pq$ mit bekannten Bits des geheimen Schlüssels $d_p = d \bmod p - 1$.
- Deterministische Polynomialzeitäquivalenz des Faktorisierens von N und der Berechnung des geheimen Schlüssels d.

9.1.1 Faktorisierung mit bekannten Bits von p

Das nachfolgende Resultat wurde 1996 von Coppersmith gezeigt. Wir betrachten ein Szenario, in dem ein Angreifer obere Bits des geheimen Primteilers p eines RSA-Moduls $N = pq$ erhält. Z.B. werden im 1995 vorgeschlagenen Vanstone-Zuccherato Schema die obersten 264 von insgesamt 512 Bits von p dazu verwendet, die Identität eines Benutzers zu codieren. Sie sind also öffentlich bekannt. Alternativ kann man sich vorstellen, dass ein Angreifer in den Besitz von Bits von p mit Hilfe eines Seitenkanalangriffs gelangt (s. Kapitel 5).

Die Frage ist, wieviele Bits von p einem Angreifer genügen, um N effizient zu faktorisieren. Wir werden nun durch eine direkte Anwendung von Satz 67 sehen, dass es genügt, die Hälfte der Bits von p zu kennen. Daraus folgt direkt ein Polynomialzeitangriff auf das Vanstone-Zuccherato Kryptosystem.

Satz 69 (Coppersmith 1996) *Sei $N = pq$ ein RSA-Modul mit $p > q$. Sei eine Approximation $\tilde{p}$ von p gegeben mit*

$$|p - \tilde{p}\,| \leq N^{\frac{1}{4}}.$$

Dann kann die Faktorisierung von N in Zeit polynomiell in $\log N$ berechnet werden.

Beweis: Das Polynom

$$f(x) = \tilde{p} + x$$

hat die Nullstelle $x_0 = p - \tilde{p}$ modulo p, wobei $|x_0| \leq N^{\frac{1}{4}}$.

Wir wenden nun Satz 67 an. Wir haben ein univariates monisches Polynom $f(x)$ mit Grad 1 und wollen eine Nullstelle modulo p bestimmen. Da $p > q$, folgt dass $p > N^{\frac{1}{2}}$. Damit können wir $\beta = \frac{1}{2}$ setzen. Die Schranke aus Satz 67 liefert, dass wir x_0 in Zeit polynomiell in $\log N$ berechnen können, sofern

$$|x_0| \leq N^{\beta^2} = N^{\frac{1}{4}}.$$

Damit erhalten wir $x_0 = p - \tilde{p}$ und damit die Faktoren $p = x_0 + \tilde{p}$ und $q = \frac{N}{q}$. □

9.1.2 Faktorisieren mit bekannten Bits von $d_p = d \bmod p - 1$

Wir betrachten hier folgendes Szenario: Alice verwendet RSA mit kleinem Exponenten e und die geheimen Parameter $d_p = d \bmod p - 1$ und $d_q = d \bmod q - 1$ zum effizienten Entschlüsseln/Signieren. Einem Angreifer Eve gelingt es, obere Bits des geheimen Parameter d_p zu ermitteln (z.B. mit Hilfe eines Seitenkanalangriffs). Die Frage ist wie im letzten Abschnitt, wann ein Angreifer genügend Bits hat, um N effizient zu faktorisieren.

Im Unterschied zum letzten Abschnitt erhält der Angreifer Eve hier also Bits des geheimen Schlüssels d_p anstatt Bits des geheimen Primfaktors. In der Praxis ist das ein realistischeres Szenario, da man bei den meisten bekannten Seitenkanalangriffe Bits des geheimen Schlüssels erhält.

In unserem Angriff werden wir die Kenntnis von d_p dazu verwenden, um eine Approximation eines Vielfachen von p zu berechnen. Eine Verallgemeinerung von Satz 69 wird uns dann die Faktorisierung von N liefern. Diese Verallgemeinerung zu zeigen, ist eine Übungsaufgabe.

Übung 70 *Sei $N = pq$ ein RSA-Modul mit $p > q$. Sei $k \in \mathbb{N}$ eine unbekannte Zahl, die kein Vielfaches von q ist. Weiterhin sei eine Approximation $\widetilde{kp}$ von kp gegeben mit*

$$|kp - \widetilde{kp}| \leq N^{\frac{1}{4}}.$$

Dann kann die Faktorisierung von N in Zeit polynomiell in $\log N$ berechnet werden.

Der folgende Angriff wurde 2003 von Blömer und May vorgestellt. Falls p und q von derselben Größenordnung sind, dann hat d_p etwa halb soviele Bits wie N. Im Fall konstanter Exponenten e benötigt der Angriff nur die Hälfte der obersten Bits von d_p. Für größere Werte von e werden mehr Bits benötigt.

Satz 71 (Blömer, May 2005) *Sei $N = pq$, $p > q$ ein RSA-Modul und $e = N^\alpha$, $0 < \alpha \leq \frac{1}{4}$ der öffenliche Exponent. Sei eine Approximation $\widetilde{d}_p$ von d_p gegeben mit*

$$|d_p - \widetilde{d}_p| \leq N^{\frac{1}{4}-\alpha}.$$

Dann kann die Faktorisierung von N in Zeit polynomiell in $\log N$ berechnet werden.

Beweis: Wir wissen, dass $ed_p = 1 \bmod p - 1$. Daher gilt

$$ed_p - 1 = k(p-1)$$

für ein unbekanntes $k \in \mathbb{N}$ mit

$$k = \frac{ed_p - 1}{p-1} < e \cdot \frac{d_p}{p-1} < e.$$

Daher gilt $k \leq N^{\frac{1}{4}}$. Daraus folgt, dass k kein Vielfaches von q sein kann. Berechnen wir nun $\widetilde{kp} = e\tilde{d}_p - 1$ als Approximation von kp. Diese Approximation hat die Güte

$$|kp - \widetilde{kp}| = |ed_p + k - 1 - (e\tilde{d}_p - 1)| = |e(d_p - \tilde{d}_p) + k| \leq N^{\alpha} N^{\frac{1}{4}-\alpha} + N^{\frac{1}{4}} \leq 2N^{\frac{1}{4}}.$$

Um das Resultat aus Übung 70 anzuwenden, benötigen wir eine Approximation von kp mit Güte $N^{\frac{1}{4}}$. Einer der beiden Werte $\widetilde{kp} \pm N^{\frac{1}{4}}$ erfüllt diese Anforderung. Wir geben dem Übung 70 zugrundeliegenden Algorithmus einfach nacheinander beide Werte als Eingabe. Einer davon liefert die Faktorisierung von N in Zeit polynomiell in $\log N$. □

9.1.3 Faktorisieren von RSA-Moduln $N \equiv_{dp}$ Berechnen von d

Das größte ungelöste RSA-Problem ist die Frage, ob das Invertieren der Verschlüsselungsfunktion $e_k(m) = m^e \bmod N$ Polynomialzeit-äquivalent zur Berechnung der Faktorisierung von N ist. Man weiss, dass die Faktorisierung von N eine effiziente Methode zum Invertieren von e_k liefert. Unbekannt ist dagegen die Rückrichtung, d.h. ob jeder Algorithmus der e_k invertiert zum effizienten Faktorisieren von N verwendet werden kann.

Eine Möglichkeit, die Funktion e_k zu invertieren ist es, den geheimen Schlüssel d zu berechnen. Wir wollen nun zeigen, dass jeder effiziente Algorithmus, der das geheime $d \in \mathbb{Z}^*_{\phi(N)}$ mit $ed = 1 \bmod \phi(N)$ berechnet, dazu verwendet werden kann, die Faktorisierung von N in *deterministischer* Polynomialzeit zu berechnen[1].

Man beachte, dass dies nicht die Frage löst, ob die Invertierung der RSA-Funktion Polynomialzeit-äquivalent zum Faktorisierungsproblem ist. Es ist ja nicht klar, ob man zuerst d berechnen muss, um aus einem RSA-Chiffretext den zugrundeliegenden Klartext effizient zu berechnen. Wir haben bereits mehrere Angriffe kennengelernt, die z.B. Chiffretexte $c = m^e \bmod N$ für spezielle Parameter m und e entschlüsseln können, ohne dabei d zu berechnen.

Satz 72 (May 2005) *Sei $N = pq$ ein RSA-Modul mit p, q gleicher Bitgröße und e ein öffentlicher Exponent. Angenommen man hat einen Algorithmus A, der in Polynomialzeit d berechnet mit*

$$ed = 1 \bmod \phi(N) \quad \textit{und} \quad ed < \phi(N)^2.$$

Dann kann die Faktorisierung von N in Zeit polynomiell in $\log N$ berechnet werden.

[1] Eine *probabilistische* Lösung für dieses Problem wurde schon 1976 von Rivest, Shamir und Adleman basierend auf einem Algorithmus von Miller vorgeschlagen.

Beweis: Wir wenden Algorithmus A an, um d zu berechnen. Es gilt $ed-1 = k\phi(N)$ bzw.

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

Wir kennen $ed-1$, und damit ein Vielfaches von $\phi(N)$. Nun schreiben wir $\phi(N) = (p-1)(q-1) = N-(p+q-1)$. D.h. aber, wir können unseren RSA-Modul N als eine Approximation von $\phi(N)$ mit additivem unbekanntem Fehlerterm $p+q-1$ interpretieren. Für p, q gleicher Bitgröße wissen wir aus dem Beweis zum Satz von Wiener (Satz 52), dass

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

Wir folgern, dass einer der 6 Werte $N - \frac{i}{2}N^{\frac{1}{2}}$ für $i = 0, \ldots, 5$ eine Approximation von $\phi(N)$ bis auf einen Fehlerterm der Größe maximal $\frac{1}{2}N^{\frac{1}{2}}$ ist. Sei $\widetilde{\phi(N)}$ dieser Wert[2]. Für hinreichend große N gilt sicherlich $N \leq 2\phi(N)$. Daraus folgern wir

$$\widetilde{\phi(N)} - \phi(N) \leq \frac{1}{2}N^{\frac{1}{2}} < \phi(N)^{\frac{1}{2}}.$$

Nun definieren wir das Polynom

$$f(x) = \widetilde{\phi(N)} - x$$

mit einer Nullstelle $x_0 = \widetilde{\phi(N)} - \phi(N) \leq \phi(N)^{\frac{1}{2}}$ modulo $\phi(N)$. Wir wollen nun Satz 67 anwenden, um die Nullstelle x_0 zu bestimmen.

Wir wissen, dass $b = \phi(N)$ ein Teiler von $M = ed-1 \leq \phi(N)^2$ ist. Damit ist $b \geq M^{\frac{1}{2}}$, und wir können $\beta = \frac{1}{2}$ setzen. Mit Hilfe von Satz 67 können wir alle Nullstellen x_0 von $f(x)$ berechnen, sofern

$$|x_0| \leq M^{\beta^2} \leq (\phi(N)^2)^{\frac{1}{4}} = \phi(N)^{\frac{1}{2}}.$$

Dies Schranke wird aber von der gesuchten Nullstelle x_0 nach Konstruktion erfüllt. Die Laufzeit ist polynomiell in $\log M$ und damit auch polynomiell in $\log N$.

Wenn wir nun x_0 kennen, können wir einfach $\phi(N) = \widetilde{\phi(N)} - x_0$ berechnen. Wir haben aber bereits in einer Übungsaufgabe gezeigt, dass man mittels Kenntnis der Werte $\phi(N)$ und N die Faktorisierung von N effizient bestimmen kann. □

[2]Wir können oBdA davon ausgehen, den korrekten Wert zu kennen, da wir das folgende Verfahren einfach für $i = 0, \ldots, 5$ durchführen können.

9.2 Erweiterungen der Coppersmith-Methode auf multivariate Polynome

Die Coppersmith-Methode kann man auch für das Lösen modularer multivariater Polynome, d.h. Polynome in mehr als einer Variablen, verwenden. In der Verallgemeinerung für ein Polynom $f(x_1, \ldots, x_k)$ mit k Variablen verfährt man wie folgt:

- Konstruiere k Polynome $g_1(x_1, \ldots, x_k), \ldots, g_k(x_1, \ldots, x_k)$, die alle gesuchten kleinen Nullstelle über den ganzen Zahlen besitzen. Um solche Polynome zu konstruieren, kann man eine Verallgemeinerung des Satzes von Howgrave-Graham verwenden (Satz 58). Wir haben in einer Übung bereits eine Verallgemeinerung für bivariate Polynome kennengelernt.
- Berechne die Nullstellen mit Hilfe von Resultantenverfahren.

Der Nachteil der multivariaten modularen Variante von Coppersmith ist, dass die Methode nur noch heuristisch und nicht mehr beweisbar ist. Ein Problem ist, dass die Polynome g_i untereinander nicht-triviale gemeinsame Teiler besitzen können. Dann erhält man als Resultat einer Resultantenberechnung das Nullpolynom, aus dem man nicht die Nullstellen von $f(x_1, \ldots, x_k)$ berechnen kann. In der Praxis funktioniert die multivariate Coppersmith-Methode aber sehr gut.

Boneh und Durfee stellten 1999 einen Angriff für RSA mit kleinem geheimen Exponenten d vor, der eine bessere Schranke als die Wiener-Schranke $d \leq \frac{1}{3}N^{\frac{1}{4}}$ aus Satz 52 liefert. Dazu schreibt man die RSA-Gleichung $ed = 1 + k(N - (p + q - 1))$ als bivariates Polynom

$$f(x, y) = x(N - y) + 1$$

mit einer Nullstelle $(x_0, y_0) = (k, p + q - 1)$ modulo e. Boneh und Durfee zeigten, dass diese Nullstelle (x_0, y_0) in Zeit polynomiell in $\log N$ gefunden werden kann, falls

$$d \leq N^{1-\sqrt{\frac{1}{2}}} \approx N^{0.293}.$$

Dies ist derzeit die beste Schranke, die man in der RSA-Kryptanalyse für kleine d kennt.

10 Das Hidden Number Problem: Angriff auf DSA

Nachdem wir in den letzten Kapiteln einige gitterbasierte Angriffe auf das RSA Kryptosystem kennengelernt haben, werden wir in diesem Kapitel den derzeit stärksten gitterbasierten Angriff auf das DSA Signaturverfahren betrachten.

Das DSA Signaturverfahren haben wir bereits in Kapitel 4 kennengelernt. Zur Erinnerung: Der öffentliche DSA-Schlüssel besteht aus einer Primzahl p, einem Generator α einer Untergruppe mit primer Ordnung q und einem Element $\beta = \alpha^a$, wobei a der geheime Schlüssel ist. Signaturen haben die Form

$$\mathrm{sig}_k(x) = (\gamma, \delta) = ((\alpha^r \bmod p) \bmod q, r^{-1}(x + a\gamma) \bmod q)$$

für einen geheimen Randomisierungsparameter $r \in \mathbb{Z}_q^*$. Die zufällige Wahl von r ist entscheidend für die Sicherheit des Verfahrens. Man sieht leicht, dass ein Angreifer mit Kenntnis von r den geheimen Schlüssel effizient berechnen kann (Übungsaufgabe). Zudem muss r bei jeder Signatur stets neu gewählt werden, da eine zweimalige Verwendung desselben r ebenfalls zur Berechnung von a verwendet werden kann.

Übung 73 *Seien $sig_k(x)$, $sig_k(x')$ zwei Unterschriften unterschiedlicher Nachrichten $x \neq x' \bmod q$ unter Verwendung desselben r. Zeigen Sie, dass dann a effizient berechnet werden kann, sofern $\gamma \neq 0$.*

10.1 Angriff auf DSA mit bekannten Bits des Randomisierungsparameters r

Der folgende Angriff wurde 1999 von Nguyen vorgestellt. Er zeigt, dass man DSA mit Hilfe einer Lösung des sogenannten *Hidden Number Problems* angreifen kann. Ein gitterbasierter Algorithmus zum Lösen eines Hidden Number Problems wurde 1996 von Boneh und Venkatesan vorgeschlagen. Wir stellen hier zunächst den Angriff vor. Dann definieren wir das Hidden Number Problem und zeigen, dass der Angriff auf DSA ein Spezialfall des Hidden Number Problems ist. Abschließend geben wir einen Gitteralgorithmus zum Lösen des Hidden Number Problems.

Wir werden folgendes Szenario betrachten: Der Randomisierungsparameter r wird jedesmal mit Hilfe eines Pseudozufallszahlengenerators neu bestimmt. Ein Angreifer kann

zwar nicht r selbst ermitteln, aber eine gewisse Anzahl der unteren Bits von r. Ein solches Szenario ist in der Praxis durchaus realistisch, z.B. wird in der AT&T CryptoLib – aufgrund einer fehlerhaften Implementierung – stets ein ungerades r verwendet, so dass ein Angreifer stets das unterste Bit von r kennt. Ebenso kann man sich Seitenkanalangriffe als Quelle für Bits von r denken.

Die Frage ist nun, wieviele Bits von r und wieviele verschiedene Unterschriften ein Angreifer benötigt. Hier gibt es einen Trade-off: Je mehr Unterschriften ein Angreifer kennt, desto mehr Information besitzt er, d.h. desto kleiner sollte die Anzahl der von ihm benötigten Bits sein. Wir werden im folgenden stets annehmen, dass ein Angreifer polynomiell viele Unterschriften kennt. Unser Ziel ist es dann, die Anzahl der von ihm benötigten Bits zu minimieren.

Angenommen ein Angreifer kennt $d = \text{poly}(\log q)$ viele Signaturen $\text{sig}_k(x_i) = (\gamma_i, \delta_i)$ verschiedener Nachrichten x_i, $i = 1, \ldots, d$. Ferner kennt er für jede Signatur $\text{sig}(x_i)$ die ℓ untersten Bits des verwendeten Zufallswerts r_i. Sei $r_i = r_i^{(m)} 2^\ell + r_i^{(\ell)}$. D.h. der Angreifer kennt $r_i^{(\ell)}$ mit

$$r_i - r_i^{(\ell)} = 2^\ell r_i^{(m)}.$$

Schreiben wir nun $\delta_i = r_i^{-1}(x_i + a\gamma_i)$ in der Form

$$a\gamma_i = \delta_i r_i - x_i = \delta_i (r_i^{(m)} 2^\ell + r_i^{(\ell)}) - x_i \bmod q.$$

Multiplikation mit dem Inversen von $2^\ell \delta_i \bmod q$ liefert

$$a\gamma_i 2^{-\ell} \delta_i^{-1} = 2^{-\ell}(r_i^{(\ell)} - \delta^{-1} x_i) + r_i^{(m)} \bmod q.$$

(Übung: Was passiert, wenn $(2^\ell \delta_i)^{-1}$ nicht existiert?)
Die beiden Parameter a und $r_i^{(m)}$ sind die einzigen Unbekannten der Gleichung. Setze $t_i = \gamma_i 2^{-\ell} \delta_i^{-1}$. Man beachte, dass t_i bekannt ist. Der ebenfalls bekannte Term $\widetilde{a}t_i = 2^{-\ell}(r_i^{(\ell)} - \delta^{-1} x_i)$ liefert eine Approximation des Wertes at_i bis auf den Fehlerterm $r_i^{(m)}$. Die Größe des Fehlerterms lässt sich abschätzen durch

$$r_i^{(m)} = \frac{r_i - r_i^{(\ell)}}{2^\ell} < \frac{q}{2^\ell}.$$

Um den geheimen Parameter a zu finden, müssen wir das folgende Problem lösen, das auch *Hidden Number Problem* genannt wird.

Definition 74 (HNP) *Sei q prim. Beim* Hidden Number Problem (HNP) *erhält man als Eingabe Zahlen $t_1, \ldots, t_d \in \mathbb{Z}_q$ und zusätzlich $\widetilde{a}t_1, \ldots, \widetilde{a}t_d$ mit*

$$|(at_i \bmod q) - \widetilde{a}t_i| \leq \frac{q}{2^\ell}$$

für ein unbekanntes $a \in Z_q$. Gesucht ist der geheime Parameter a.

Intuitiv sollte klar sein, dass für hinreichend große d und l und uniform verteilte, unabhängig gewählte t_i das geheime a eindeutig bestimmt ist. Nun wollen wir a mit Hilfe eines Gitteralgorithmus auch berechnen.

10.2 Gitterbasierte Lösung des Hidden Number Problems

Wir betrachten die von Boneh und Venkatesan 1996 vorgestellte gitterbasierte Lösung des Hidden Number Problems. Dazu betrachten das Gitter L, das von den Zeilenvektoren der folgenden $(d \times (d+1))$-Basismatrix B erzeugt wird:

$$B = \begin{pmatrix} q & 0 & \dots & 0 & 0 \\ 0 & q & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \dots & 0 & q & 0 \\ t_1 & t_2 & \dots & t_d & \frac{1}{2^\ell} \end{pmatrix}$$

Multiplikation des letzten Basisvektors mit der geheimen Zahl a liefert den Gittervektor

$$(at_1, at_2, \dots at_d, \frac{a}{2^\ell}).$$

Da wir im i-ten Eintrag dieses Vektors durch Substraktion des i-ten Basisvektors von B beliebige Vielfache von q subtrahieren können, ist auch der Vektor

$$t = \left(at_1 \bmod q, at_2 \bmod q, \dots, at_d \bmod q, \frac{a}{2^\ell}\right)$$

ein Gittervektor in L. Wir bezeichnen diesen Vektor t als *Targetvektor*. Hat man den Targetvektor t bestimmt, so kann man aus t den geheimen Parameter a direkt ablesen.

Wir kennen aber bereits einen Vektor $\tilde{t} = (\widetilde{at}_1, \dots, \widetilde{at}_d, 0)$, der nah an unserem Targetvektor t ist. Der Abstand von $\tilde{t}$ zu t beträgt

$$\|t - \tilde{t}\| \leq \sqrt{d+1} \cdot \frac{q}{2^l},$$

denn für die ersten d Koordinaten der Differenz gilt $|(at_i \bmod q) - \widetilde{at}_i| \leq \frac{q}{2^\ell}$ nach Definition des HNP und für die letzte Koordinate gilt $\frac{a}{2^l} < \frac{q}{2^\ell}$.

Damit ist der Vektor $\tilde{t}$ sehr nah zum gesuchten Targetvektor $t \in L$ und wir können hoffen, den Vektor t durch Lösen eines CVPs in L zu finden. In der Tat kann man dies sogar beweisen.

Nguyen zeigte aufbauend auf der Methode von Boneh und Venkatesan, dass für unabhängig gleichverteilte t_i jeder Gittervektor $u \in L$ mit $\|u - \tilde{t}\| < \sqrt{d+1} \cdot \frac{q}{2^\ell}$ den geheimen Parameter a mit einer signifikanten Wahrscheinlichkeit liefert, die abhängig ist von l und d. Der Angriff funktioniert beweisbar mit konstanter Wahrscheinlichkeit, falls l von der Größenordnung $\log\log q$ und d von der Größenordnung $\log q$ ist. Eine explizite Auswertung der Wahrscheinlichkeitsfunktion liefert für 160-Bit Primzahlen q einen Angriff mit Erfolgswahrscheinlichkeit mindestens $\frac{1}{2}$ für die Parameterwahl $\ell = 6$ und $d = 100$.

D.h. mit anderen Worten: Gegeben 100 DSA-Unterschriften und zu diesen Unterschriften jeweils die untersten 6 Bits des Randomisierungsparameters r. Dann kann

durch Lösen eines CVPs in einem Gitter der Dimension 101 der geheime Parameter a mit Wahrscheinlichkeit mindestens $\frac{1}{2}$ bestimmt werden (unter der Annahme, dass die $t_i\gamma_i^{-1} = 2^{-\ell}\delta_i^{-1}$ bei den gegebenen DSA-Unterschriften unabhängig gleichverteilt sind).

In der Praxis liefert der Angriff sogar noch bessere Ergebnisse. Es zeigt sich, dass für 160-Bit Primzahlen q eine Anzahl von 100 DSA-Unterschriften mit nur jeweils 3 bekannten Bits der r_i genügen.

Dies sollte man als Warnung verstehen, in der Praxis sehr starke Pseudozufallsgeneratoren zur Erzeugung der r_i bei DSA zu verwenden, und die r_i gegenüber Seitenkanalangriffen physikalisch zu schützen.

10.3 Sicherheit von Diffie-Hellman Bits

Boneh und Venkatesan zeigten 1996 den folgenden Satz.

Fakt 75 *Das Hidden Number Problem kann mit Hilfe des L^3-Algorithmus für die Parameterwahl $\ell = \sqrt{\log q} + \log\log q$ und $d = 2\sqrt{\log q}$ in polynomieller Zeit gelöst werden.*

In diesem Abschnitt werden wir stets die Parameterwahl aus Fakt 75 für die Parameter ℓ und d voraussetzen.

Die polynomielle Lösbarkeit des Hidden Number Problems hat direkte Auswirkungen auf die Sicherheit einzelner Bits beim Diffie-Hellman Schlüsselaustausch-Verfahren.

Diffie Hellman Schlüsselaustausch

Sei q eine Primzahl (1024 Bit), α ein Generator der Gruppe $\mathbb{Z}_q^*$.

(1) Alice wählt $b \in \mathbb{Z}_{q-1}$ zufällig und sendet $\alpha^b \bmod q$ an Bob.

(2) Bob wählt $c \in \mathbb{Z}_{q-1}$ zufällig und sendet $\alpha^c \bmod q$ an Alice.

(3) Alice berechnet $K = (\alpha^c)^b = \alpha^{bc} \bmod q$.
Analog berechnet Bob $K = (\alpha^b)^c = \alpha^{bc} \bmod q$.

Wir sehen, dass Alice und Bob nach Beendigung des Protokolls den gemeinsamen Schlüssel $K = \alpha^{bc}$ besitzen. Ein lauschender Angreifer Eve müsste aus den abgefangenen Werten α^b und α^c den Schlüssel $K = \alpha^{bc}$ berechnen. Derzeit vermutet man, dass dieses Problem so schwer ist wie das Diskrete Logarithmus Problem. Wir nehmen also an, dass das Berechnen von $K = \alpha^{bc}$ ein schweres Problem ist.

Der Diffie Hellman Schlüsselaustausch ist ein heutzutage weitverbreitetes Protokoll, um sogenannte Session-Keys für ein symmetrisches Kryptosystem zu erzeugen. Diese Session-Keys haben aber im allgemeinen eine deutlich kleinere Bitlänge als K. Nehmen

wir an, man möchte aus einem 1024-Bit Schlüssel K einen 64-Bit Session-Key generieren. Die einfachste Möglichkeit wäre es, die obersten 64 Bits von K zu verwenden.

Nach Annahme ist das Berechnen von $K = \alpha^{bc}$ ein schweres Problem. Dies heißt aber noch nicht, dass auch das Berechnen eines Bruchteils der Bits von K ebenfalls ein schweres Problem ist. Interessanterweise können wir mit Hilfe des Hidden Number Problems zeigen, dass das Berechnen der obersten $\ell = \sqrt{\log q} + \log\log q$ Bits von K genauso schwer ist wie das Berechnen von K selbst. Wir werden nämlich zeigen, dass jeder Angreifer, der die obersten ℓ Bits in Polynomialzeit berechnen kann, ebenfalls das gesamte K in Polynomialzeit berechnen kann. Im Falle eines 1024-Bit q kann man damit mindestens die obersten $\sqrt{1024}+\log 1024 = 2^5+10 = 42$ Bits als Session-Key verwenden.

Satz 76 *Sei A ein polynomieller Algorithmus, der bei Eingabe α^b, α^c die obersten ℓ Bits von α^{bc} berechnet. Dann gibt es einen polynomiellen Algorithmus zur Berechnung von α^{bc}.*

Beweis: Sei d wie zuvor definiert. Wir wählen $x_i \in \mathbb{Z}_{q-1}$ zufällig für $i = 1, \ldots, d$ und berechnen $\alpha^b \cdot \alpha^{x_i} = \alpha^{b+x_i}$. Bei Eingabe von α^{b+x_i} und α^c liefert uns Algorithmus A die obersten ℓ Bits von

$$(\alpha^{b+x_i})^c = \alpha^{bc+x_i c} = \alpha^{bc} \cdot (\alpha^c)^{x_i}.$$

Wir setzen nun $a = \alpha^{bc}$ und $t_i = (\alpha^c)^{x_i}$. Damit haben wir gemäß Definition 74 ein Hidden Number Problem. Man beachte, dass man alle t_i berechnen kann. Weiterhin liefert Algorithmus A bei Eingabe α^{b+x_i} und α^c eine Zahl $\widetilde{at}_i$ mit

$$|(at_i \bmod q) - \widetilde{at}_i| < \frac{q}{2^\ell}.$$

Der polynomielle Algorithmus A wird d-mal aufgerufen. Da d selbst polynomiell in $\log q$ ist, benötigen diese Aufrufe insgesamt polynomielle Laufzeit. Danach hat man eine Hidden Number Problem-Instanz $t_1, \ldots, t_d$ und $\widetilde{at}_1, \ldots, \widetilde{at}_d$ generiert.

Wie zuvor erwähnt kann man für unsere Wahl von d und ℓ die Hidden Number Problem-Instanz in polynomieller Zeit lösen. Dies liefert den geheimen Schlüssel $a = \alpha^{bc}$. □

11 Klassische Faktorisierung & Faktorbasen

In diesem Kapitel wollen wir uns mit der klassischsten Art des Angriffs auf ein faktorisierungsbasierten Kryptoverfahrens beschäftigten: Der Primfaktorzerlegung eines zusammengesetzten Moduls N. Dazu wollen wir im folgenden stets voraussetzen, dass N keine Primzahlpotenz ist, d.h. wir können $N = pq$ für zwei teilerfremde $p, q > 2$ schreiben, wobei p, q nicht notwendigerweise prim sein müssen. Desweiteren setzen wir immer voraus, dass N eine ungerade Zahl ist.

Übung 77 *Sei $N = p^k$ eine Primzahlpotenz. Dann können p und k effizient berechnet werden.*

Die effizientesten heutzutage bekannten Faktorisierungsalgorithmen für allgemeine N (die Elliptische Kurvenmethode ist besonders effizient für N mit einem kleinen Primfaktor) beruhen auf dem Finden eines Tupel $(a, b) \in \mathbb{Z}_N \times \mathbb{Z}_N$ mit

$$a^2 = b^2 \bmod N \quad \text{und} \quad a \neq \pm b \bmod N.$$

Satz 78 *Seien N, $(a, b) \in \mathbb{Z}_N^2$ gegeben mit*

$$a^2 = b^2 \bmod N \quad \textit{und} \quad a \neq \pm b \bmod N.$$

Dann kann ein nicht-trivialer Faktor $d \neq 1, N$ von N in Zeit $\mathcal{O}(\log^2 N)$ berechnet werden.

Beweis: Es gilt

$$a^2 - b^2 = (a + b)(a - b) = 0 \bmod N.$$

Damit teilt N das Produkt $(a+b)(a-b)$. Wegen $a+b \neq 0 \bmod N$ und $a-b \neq 0 \bmod N$ teilt N aber weder den Faktor $a+b$ noch den Faktor $a-b$. D.h. aber ein nicht-trivialer Faktor p von N muss $a+b$ teilen und der restliche nicht-triviale Faktor $\frac{N}{p}$ von N muss ein Teiler von $a-b$ sein. Die Berechnung von $d = \gcd(N, a+b)$ liefert ein Vielfaches p, aber nicht N. Damit ist d ein nicht-trivialer Teiler von N. □

Übung 79 *Sei ein Algorithmus A gegeben, der bei Eingabe N einen nicht-trivialen Faktor von N in Zeit polynomiell in $\log N$ berechnet. Zeigen Sie, dass dann die komplette Primfaktorzerlegung von N in Zeit polynomiell in $\log N$ berechnet werden kann.*

11.1 Faktorisieren mit Faktorbasen

Unser Ziel wird es im folgenden sein, einen Algorithmus für die Konstruktion von Tupeln (a, b) der in Satz 78 genannten Form zu konstruieren.

Wir wollen zunächst zeigen, dass Tupel dieser Form überhaupt existieren.

Satz 80 *Für jedes $a \in \mathbb{Z}_N^*$ gibt es mindestens zwei $b \in \mathbb{Z}_N^*$ mit*

$$a^2 = b^2 \bmod N \quad \textit{und} \quad a \neq \pm b \bmod N.$$

Daher gibt es mindestens $\phi(N)$ Tupel $(a, b) \in \mathbb{Z}_N^2$ mit dieser Eigenschaft.

Beweis: Sei $a \in \mathbb{Z}_N^*$ beliebig. Nach Voraussetzung können wir N als Produkt pq für teilerfremde, ungerade $p, q > 2$ schreiben. Nach dem Chinesischen Restsatz hat a eine eindeutige Darstellung

$$\left| \begin{array}{c} a = a_p \bmod p \\ a = a_q \bmod q \end{array} \right|.$$

mit $a_p \in \mathbb{Z}_p^*$ und $a_q \in \mathbb{Z}_q^*$. Wir definieren nun $b_1, b_2 \in \mathbb{Z}_N^*$ als

$$\left| \begin{array}{c} b_1 = -a_p \bmod p \\ b_1 = a_q \bmod q \end{array} \right| \quad \text{und} \quad \left| \begin{array}{c} b_2 = a_p \bmod p \\ b_2 = -a_q \bmod q \end{array} \right|.$$

Man beachte, dass $b_1 = -b_2 \bmod N$. Ferner gilt offenbar

$$\left| \begin{array}{c} a^2 = a_p^2 = b_1^2 = b_2^2 \bmod p \\ a^2 = a_q^2 = b_1^2 = b_2^2 \bmod q \end{array} \right|.$$

Damit gilt nach dem Chinesischen Restsatz $a^2 = b_1^2 = b_2^2 \bmod N$. Wir zeigen nun, dass $a \neq b_1 \bmod N$. Analog folgt $a \neq b_2 \bmod N$ und damit $a \neq \pm b_1, \pm b_2 \bmod N$.

Angenommen $a = b_1 \bmod N$. Dann gilt $a_p = -a_p \bmod p$ und damit teilt p den Term $2a_p$. Wegen p ungerade und $a_p \in Z_p^*$, ist $2a_p \in Z_p^*$ und daraus folgt

$$2a_p \neq 0 \bmod p.$$

Dies ist ein Widerspruch zu p teilt $2a_p$. Damit muss $a \neq b_1 \bmod N$ gelten.

Bleibt zu zeigen, dass es mindestens $\phi(N)$ Tupel (a, b) mit der geforderten Eigenschaft gibt. Wir haben für jedes $a \in Z_N^*$ zwei verschiedene b_1, b_2 konstruiert. Damit haben wir $2\phi(N)$ Tupel und da jedes Tupel höchstens zweimal gezählt wird, erhalten wir $\phi(N)$ verschiedene Tupel (a, b). □

Beispiel 81 *Wir wollen die Zahl 143 faktorisieren. Nun gilt $12^2 = 144 = 1^2 \bmod 143$, und wir erhalten ein Tupel $(a, b) = (12, 1)$. Damit teilt 143 den Term $12^2 - 1^2 = (12 - 1) \cdot (12 + 1)$, aber keinen der beiden Faktoren $(12 - 1)$ und $(12 + 1)$. Wir berechnen $ggT(143, 12 - 1) = 11$ und $ggT(143, 12 + 1) = 13$ und finden die Faktorisierung $143 = 11 \cdot 13$.*

Die Frage ist nun, wie man Tupel (a, b) algorithmisch finden kann. Satz 80 garantiert uns lediglich, dass es für jedes $a \in Z_N^*$ zwei geeignete b gibt, die Konstruktion der b verlangt aber im Beweis zu Satz 80 Kenntnis einer nicht-trivialen Faktorisierung von N, die wir nicht kennen (Das ist ja gerade unser Ziel!).

Die Idee unseres Faktorisierungsalgorithmus ist die folgende:

(1) Wähle hinreichend viele a_i mit der Eigenschaft, dass $a_i^2 \bmod N$ in kleine Primfaktoren zerlegt werden kann.

(2) Finde mit Hilfe der in Schritt(1) bestimmten Primfaktorzerlegungen eine Teilmenge der a_i^2, deren Produkt modulo N wiederum eine Quadratzahl ist.

Um den ersten Punkt näher zu spezifizieren, definieren wir eine sogenannte *Faktorbasis*, d.h. eine Menge bestehend aus allen Primzahlen bis zu einer Schranke B. Wir wollen später die $a_i^2 \bmod N$ ausschließlich in Primfaktoren aus dieser Faktorbasis zerlegen.

Definition 82 (Faktorbasis) *Sei $B \in \mathbb{N}$. Wir bezeichnen die Menge*

$$F_B = \{x \in \mathbb{N} \mid x \text{ prim}, x \leq B\} \cup \{-1\}$$

als Faktorbasis. *Eine ganze Zahl, die ausschließlich Primfaktoren aus F_B enthält, bezeichnen wir als* B-glatt.

Wir werden nun zeigen, dass man mit Hilfe einer Faktorbasis F_B der Kardinalität h und mindestens $h+1$ B-glatten Zahlen $a_1^2 \bmod N, \ldots, a_{h+1}^2 \bmod N$ ein Tupel (a, b) mit $a^2 = b^2 \bmod N$ konstruieren kann.

Satz 83 *Sei $B = \{p_1, \ldots, p_h\}$ und seien $a_1^2 \bmod N, \ldots, a_{h+1}^2 \bmod N$ B-glatte Zahlen. Dann kann man in Laufzeit polynomiell in $\log N$ und h ein Tupel (a, b) berechnen mit $a^2 = b^2 \bmod N$.*

Beweis: Da die a_i^2 nach Voraussetzung B-glatt sind, kann man sie als Produkt der Primzahlen aus B schreiben:

$$a_i^2 = \prod_{j=1}^{h} p_j^{e_{i,j}} \bmod N. \tag{11.1}$$

Aus den Exponenten der Faktorisierung der a_i erhalten wir den Vektor $e_i = (e_{i,1}, \ldots, e_{i,h})$. Der Term auf der rechten Seite der Gleichung ist genau dann ein Quadrat, falls die Exponenten $e_{i,j}$ alle gerade sind. Wir betrachten daher den Vektor

$$f_i = (f_{i,1}, \ldots, f_{i,h}) = (e_{i,1} \bmod 2, \ldots, e_{i,h} \bmod 2).$$

Falls einer der Vektoren f_i mit $1 \leq i \leq h+1$ der Nullvektor ist, so haben wir ein Quadrat gefunden. Nehmen wir an, dass keiner der Vektoren f_i der Nullvektor ist. Nun

ist jedes Produkt von B-glatten Zahlen $a_i^2 \bmod N$ wieder eine B-glatte Zahl. Wenn wir verschiedene a_i^2 multiplizieren, so addieren sich deren Exponentenvektoren e_i und damit auch die korrespondierenden Vektoren f_i.

Unser Ziel ist es nun, eine Summe von Vektoren f_i zu finden, die sich zum Nullvektor aufaddieren. Dazu schreiben wir $f_1, \ldots, f_{h+1}$ als Zeilenvektoren einer $(h+1) \times h$-Matrix mit Einträge aus dem Körper $\mathbb{F}_2$. Da die Anzahl der Zeilen größer als die Anzahl der Spalten ist, gibt es eine nicht-triviale Linearkombination von Zeilen mit Koeffizienten aus $\mathbb{F}_2$, die den Nullvektor liefert. Diese Linearkombination kann mit Hilfe von Gaußelimination in Zeit polynomiell in h berechnet werden.

Seien die Zeilen $i_1, \ldots, i_k$ linear abhängig, d.h. $f_{i_1} + f_{i_2} + \cdots + f_{i_k} = 0 \bmod 2$. Wir berechnen den h-dimensionalen Vektor g als Linearkombination $g = e_{i_1} + e_{i_2} + \cdots + e_{i_k}$ der korrespondierenden Exponentenvektoren. Nach Konstruktion enthält der Vektor g nur gerade Komponenten. Daher setzen wir

$$a = \prod_{\ell=1}^{k} a_{i_\ell} \bmod N \quad \text{und} \quad b = \prod_{j=1}^{h} p_j^{\frac{g_j}{2}} \bmod N.$$

Gemäß Gleichung (11.1) ist $a^2 \bmod N$ ein Produkt der $p_j^{g_j}$ und wir erhalten wie gewünscht

$$a^2 = b^2 \bmod N.$$

Die Berechnung von a und b benötigt Zeit polynomiell in $\log N$ und h. □

Beispiel 84 *Wir wollen die Zahl* 33 *mit Hilfe der Faktorbasis* $\{-1, 2\}$ *faktorisieren. Dazu wählen wir* $a_i = 4 + i$, $i = 1, 2, \ldots$ *und versuchen* $a_i^2 \bmod 33$ *mit der Faktorbasis zu zerlegen. Wir erhalten:*

i	a_i	$a_i^2 \bmod 33$	e_i	f_i
1	5	$25 = (-8) = -1 \cdot 2^3$	$(1, 3)$	$(1, 1)$
2	6	$36 = 3$		
3	7	$49 = 16 = 2^4$	$(0, 4)$	$(0, 0)$
4	8	$64 = (-2) = -1 \cdot 2$	$(1, 1)$	$(1, 1)$

Der Term $a_2 = 6$ *läßt sich nicht mit Hilfe der Faktorbasis zerlegen. Für* $a_3 = 7$ *erhalten wir einen Nullvektor* f_3*, den wir verwenden könnten. Gleichfalls liefert aber auch* $f_1 + f_4$ *eine nicht-triviale Linearkombination des Nullvektors. Wir setzen daher* $g = e_1 + e_4 = (2, 4)$ *und*

$$a = a_1 \cdot a_4 = 5 \cdot 8 = 7 \bmod 33 \quad \textit{und} \quad b = (-1)^{\frac{g_1}{2}} \cdot 2^{\frac{g_2}{2}} = (-1) \cdot 2^2 = (-4) \bmod 33$$

Es gilt nach Konstruktion

$$a^2 = 7^2 = 49 = 16 = (-4)^2 = b^2 \bmod 33.$$

Ferner gilt aber $a \neq b \bmod 33$. *In diesem Fall liefert uns* $ggT(33, 7-4) = 3$ *und* $ggT(33, 7+4) = 11$ *die komplette Primfaktorzerlegung* $33 = 3 \cdot 11$.

Sei $B = \{p_1, \ldots, p_h\}$. In Satz 83 haben wir gezeigt, dass wir mit Hilfe von $h+1$ B-glatten $a_i^2 \bmod N$ ein Tupel (a, b) konstruieren können für das gilt: $a^2 = b^2 \bmod N$. Nach Satz 78 benötigen wir aber zusätzlich, dass $a \neq \pm b \bmod N$ gilt. Wir überlegen uns nun, dass unter der Annahme, dass wir ein zufälliges Tupel $(a, b) \in Z_N^2$ mit der Eigenschaft $a^2 = b^2 \bmod N$ generieren können, $a \neq \pm b \bmod N$ mit Wahrscheinlichkeit mindestens $\frac{1}{2}$ gilt:

Die ungünstige Wahl von $b = \pm a$ erfüllt zwar offensichtlich die Gleichung $a^2 = b^2 \bmod N$, aber nach Satz 80 gibt es ebenfalls mindestens zwei $b_1, b_2 \neq \pm a \bmod N$ mit $a^2 = b^2 \bmod N$. Daher ist die Wahrscheinlichkeit einer ungünstigen Wahl höchstens $\frac{2}{4} = \frac{1}{2}$.

Wie aber konstruiert man ein zufälliges Tupel (a, b) mit der gewünschten Eigenschaft? Sicherlich muss gelten, dass $a_i^2 > N$. Ansonsten wird beim Quadrieren nicht modulo N reduziert und es gilt $a_i^2 \bmod N = a_i^2$. Damit liefert unsere Konstruktion letztendlich $b = a$. Aus diesem Grund wählt man im allgemeinen

$$a_i = \lfloor \sqrt{N} \rfloor + i, i = 1, 2, \ldots,$$

um eine Reduktion modulo N zu erhalten. Man nimmt nun heuristisch an, dass diese Reduktion genügt, um zu gewährleisten, dass sich das konstruierte Tupel (a, b) bezüglich obiger Überlegungen wie ein zufälliges Tupel $(a, b) \in Z_N^2$ mit der Eigenschaft $a^2 = b^2 \bmod N$ verhält.

Die vorige Wahl der a_i hat noch einen anderen Vorteil. Wir benötigen bei unserer Konstruktion $h+1$ viele B-glatte $a_i^2 \bmod N$. Je kleiner eine Zahl ist, desto größer ist die Wahrscheinlichkeit, dass sie in kleine Primfaktoren zerfällt, d.h. dass sie B-glatt ist für kleines B. Bei unserer Wahl der a_i ist $a_i^2 \bmod N$ etwa von der Größenordnung $2i\sqrt{N}$, d.h. für kleines i ist die Bitlänge von $a_i^2 \bmod N$ nur etwa halb so groß wie die durchschnittliche Bitlänge einer Zahl in $\mathbb{Z}_N$. Damit können wir erwarten, dass vergleichsweise viele $a_i^2 \bmod N$ in kleine Primfaktoren zerfallen.

Wir sind nun in der Lage einen Faktorisierungsalgorithmus mit Hilfe von Faktorbasen zu beschreiben.

Faktorisieren mit einer Faktorbasis

EINGABE: N ungerade, keine Primzahlpotenz

(1) Wähle B geeignet. Setze $p_1 = (-1)$. Berechne mit Hilfe des Sieb des Erathostenes alle Primzahlen $p_2, \ldots, p_h \leq B$ und setze $F_B = \{p_1, p_2, \ldots, p_h\}$.

(2) Setze $m = \lfloor N \rfloor$ und $i = 0$.

(3) Wiederhole solange, bis $h+1$ B-glatte $a_i^2 \bmod N$ gefunden sind:

(3a) Setze $i = i + 1$ und $a_i = m + i \bmod N$. Falls $a_i^2 \bmod N = \prod_{j=1}^{h} p_j^{e_{i,j}}$, speichere a_i und den Vektor $e_i = (e_{i,1}, \ldots, e_{i,h})$. Schreibe den Vektor $t_i := e_i \bmod 2$ als Zeilenvektor einer Matrix M.

(4) Finde in M eine Teilmenge von Zeilenvektoren $t_{i_1}, \ldots, t_{i_k}$, die in $\mathbb{F}_2$ linear abhängig sind. Setze $g = e_{i_1} + e_{i_2} + \cdots + e_{i_k}$,

$$a = \prod_{\ell=1}^{k} a_{i_\ell} \bmod N \quad \text{und} \quad b = \prod_{j=1}^{h} p_j^{\frac{g_j}{2}} \bmod N.$$

Falls $a = \pm b \bmod N$, versuche eine neue Teilmenge linear abhängiger Vektoren in M zu bestimmen. Bei Bedarf gehe zu Schritt(3a) zurück, um weitere Zeilenvektoren für M zu generieren.

(5) AUSGABE $d = \gcd(N, a + b)$.

Wir müssen noch den Parameter B in unserem Algorithmus spezifizieren. Bei der Wahl von B gibt es einen Tradeoff: Je größer man B wählt, desto größer ist die Wahrscheinlichkeit, dass man B-glatte $a_i^2 \bmod N$ findet, d.h. desto größer ist die Wahrscheinlichkeit, dass man in Schritt(3a) erfolgreich ein $a_i^2 \bmod N$ in der Faktorbasis komplett zerlegen kann. Auf der anderen Seite bewirkt eine größere Faktorbasis, dass h größer ist und man in Schritt(3) mehr B-glatte Zahlen finden muss. Zudem wird unsere Matrix M größer. Eine Laufzeitanalyse zeigt, dass die Schleife in Schritt(3) der laufzeitbestimmende Schritt ist. D.h. wir müssen B derart balancieren, dass nicht zu viele B-glatte Zahlen gefunden werden müssen, aber auf der anderen Seite noch genügend Zahlen in Schritt(3a) B-glatt sind.

Mit Hilfe von analytischer Zahlentheorie und heuristischen Annahmen über die B-Glattheit von Zahlen erhält man, dass die Anzahl der Schleifendurchläufe in Schritt(3) minimiert wird für die Wahl B der Größenordnung $e^{\frac{1}{2}\sqrt{\ln N \ln\ln N}}$. Unter Verwendung zusätzlicher Tricks, wie der im folgenden kurz skizzierten Siebmethode erhält man eine Gesamtlaufzeit von

$$e^{(1+o(1))\sqrt{\ln N \ln\ln N}}.$$

Man beachte, dass dies eine *subexponentielle* Laufzeit ist, d.h. die Laufzeitfunktion liegt zwischen polynomieller Laufzeit und exponentieller Laufzeit. Polynomiell wäre für das Faktorisierungsproblem eine Laufzeit von $(\ln N)^k = e^{k \ln \ln N}$ für ein festes k. Hingegen wäre eine exponentielle Laufzeit von der Form $N^k = e^{k \ln N}$. Betrachten wir die parametrisierte Laufzeitfunktion

$$L_N(\alpha, k) = e^{k(\ln N)^\alpha (\ln \ln N)^{1-\alpha}}$$

für konstante α, k. Diese Funktion $L_N(\alpha, k)$ interpoliert für $\alpha \in [0, 1]$ zwischen polynomieller Laufzeit $L_N(0, k) = (\ln N)^k$ und exponentieller Laufzeit $L_N(1, k) = N^k$. Der von uns betrachtete Algorithmus ist eine vereinfachte Variante des sogenannten *Quadratischen Siebs* mit einer Laufzeit von

$$L_N\left(\frac{1}{2}, 1 + o(1)\right),$$

d.h. die Laufzeit ist kleiner als jede exponentielle Funktion aber größer als jede polynomielle Funktion.

Der asymptotisch beste derzeit bekannte Faktorisierungsalgorithmus, das sogenannte *Zahlkörpersieb*, hat eine asymptotische Laufzeit von

$$L_N\left(\frac{1}{3}, \left(\frac{64}{9}\right)^{\frac{1}{3}} + o(1)\right),$$

wobei $\left(\frac{64}{9}\right)^{\frac{1}{3}} \approx 1.9$. Beim Zahlkörpersieb verwendet man an Stelle einer Faktorisierung von glatten Zahlen in den ganzen Zahlen wie beim Quadratischen Sieb eine Faktorisierung in den algebraischen Zahlen.

Im folgenden Abschnitt geben wir eine Übersicht von Faktorisierungsrekorden für RSA-Moduln, die zeigt bis zu welcher Bitlänge man derzeit faktorisieren kann.

11.2 Die Idee des Siebens und Faktorisierungsrekorde

In Schritt(3a) unseres Faktorisierungsalgorithmus versuchen wir ein $a_i^2 \bmod N$ mit Hilfe unserer Faktorbasis zu zerlegen. Dazu kann man $a_i^2 \bmod N$ sukzessive durch die größtmögliche Potenz $p_j^{e_j}$ der Primzahlen $p_j \in F_B$ (außer $p_1 = (-1)$) teilen. Bei diesem Prozeß speichert man die Exponenten e_j. Falls man schließlich durch das Teilen zum Wert ± 1 gelangt, ist $a_i^2 \bmod N$ eine B-glatte Zahl.

In der Praxis zeigt sich, dass das Testen von B-Glattheit durch sukzessives Teilen nicht praktikabel ist. Anstatt jeden Kandidat einer Liste $a_1^2 \bmod N, \ldots, a_n^2 \bmod N$ einzeln durch alle Primzahlen $p \in F_B$ zu teilen, prüft man für jedes einzelne $p \in F_B$, welche der Kandidaten $a_1^2 \bmod N, \ldots, a_k^2 \bmod N$ durch p bzw. durch Potenzen von p teilbar sind. Diese Potenzen werden abdividiert und man verfährt mit der nächsten Primzahl

aus der Faktorbasis analog. B-glatt sind wiederum solche Kandidaten, die durch das Dividieren schließlich ± 1 sind.

Diesem Verfahren verdanken sowohl das Quadratische Sieb als auch das Zahlkörpersieb ihren Namen. Teilbarkeit durch p anhand eines Siebs ist leicht testbar: Man berechnet zunächst einen Kandidaten, der durch p teilbar ist. Bildlich gesprochen hat das Sieb an dieser Stelle ein Loch, durch den der Kandidat durchfällt. Die anderen Löcher des Siebs sind äquidistant mit Distanz jeweils p zueinander verteilt. Durch diese Löcher fallen die anderen Kandidaten, die ebenfalls durch p teilbar sind.

Tabelle 11.1: Heutiger Stand der RSA Faktorisierungsrekorde

Zahl	Bits	Preis	faktorisiert	Gruppe um	GIPS-J.	Algorithmus
RSA-100	332		April 1991	A. K. Lenstra	0,007	Quadr. Sieb
RSA-110	365		April 1992	A. K. Lenstra	0,075	Quadr. Sieb
RSA-120	399		Juni 1993	T. Denny	0,830	Quadr. Sieb
RSA-129	429	\$100	April 1994	A. K. Lenstra	5	Quadr. Sieb
RSA-130	432		April 1996	A. K. Lenstra	1	Zahlkörpersieb
RSA-140	465		Feb. 1999	H. te Riele	2	Zahlkörpersieb
RSA-155	512		Aug. 1999	H. te Riele	8,4	Zahlkörpersieb
RSA-160	532		April 2003	J. Franke		Zahlkörpersieb
RSA-576	576	\$10.000	Dez. 2003	J. Franke	13,2	Zahlkörpersieb
RSA-640	640	\$20.000	Nov. 2005	J. Franke	170	Zahlkörpersieb
RSA-703	703	\$30.000		offen		
⋮		⋮				
RSA-1024	1024	\$100.000		offen		

12 Index-Kalkulus: DL mit Faktorbasen

In Kapitel 11 haben wir gesehen, wie man das Faktorisierungsproblem mit Hilfe eines auf Faktorbasen basierenden Algorithmus in subexponentieller Zeit lösen kann. Interessanterweise kann auch das Problem des diskreten Logarithmus - obwohl es zunächst grundverschieden zum Faktorisierungsproblem zu sein scheint - mit Hilfe eines auf Faktorbasen basierenden Algorithmus in subexponentieller Zeit gelöst werden.

Wir wiederholen hier zur Erinnerung die Definition des Diskreten Logarithmus Problems aus Kapitel 4:

Definition 85 (DL-Problem) *Sei G eine multiplikative endliche Gruppe und $\alpha \in G$. Sei β in der von α erzeugten Untergruppe, d.h. $\beta \in \langle\alpha\rangle$. Gesucht ist das eindeutige $a \bmod ord(\alpha)$ mit $\alpha^a = \beta$. Wir verwenden die Notation $a = dlog_\alpha(\beta)$. Das diskrete Logarithmusproblem bezeichnen wir auch abkürzend als* DL-Problem.

Wir wollen hier stets davon ausgehen, dass unsere Gruppe G gleich der Gruppe $\mathbb{Z}_p^*$, p prim, ist, da $\mathbb{Z}_p^*$ die in der Kryptographie meistverwendete Gruppe ist. Abkürzend setzen wir $n := \mathrm{ord}_G(\alpha)$. Falls das Element α ein Generator von $\mathbb{Z}_p^*$ ist, gilt $n = p - 1$, ansonsten ist n ein echter Teiler der Gruppenordnung $p-1$. Wir berechnen mit unserem Algorithmus in jedem Fall $\mathrm{dlog}_\alpha(\beta) \bmod p - 1$, dieses Ergebnis kann dann modulo n reduziert werden. Daher hängt die Laufzeit des Index-Kalkulus Algorithmus stets von p ab, auch wenn n signifikant kleiner als $p - 1$ sein sollte.

Es ist ein offenes Problem, einen Algorithmus zu konstruieren, dessen Laufzeit subexponentiell im Parameter n ist. Im folgenden wollen wir stets davon ausgehen, dass α ein Generator von $\mathbb{Z}_p^*$ ist, ansonsten würden wir zusätzliche Annahmen über die B-Glattheit von Zahlen aus der Untergruppe $\langle\alpha\rangle$ benötigen.

Unsere Idee zum Lösen des DL-Problems mit Hilfe einer Faktorbasis $F_B = \{p_1, \ldots, p_h\}$ ist wie folgt:

Schritt (1): Bestimme für alle $p_j \in F_B$ den Wert $\mathrm{dlog}_\alpha(p_j)$, d.h. wir berechnen zunächst für alle Primzahlen aus der Faktorbasis den Diskreten Logarithmus zur Basis α.

Schritt (2): Wähle $r_0 \in Z_p$, so dass $\alpha^{r_0}\beta \bmod p$ eine B-glatte Zahl ist, d.h.

$$\begin{aligned}\alpha^{r_0}\beta = \alpha^{a+r_0} &= \prod_{j=1}^{h} p_j^{e_j} \bmod p \\ &= \prod_{j=1}^{h} \left(\alpha^{\mathrm{dlog}_\alpha(p_j)}\right)^{e_j} \bmod p \\ &= \prod_{j=1}^{h} (\alpha)^{e_j \mathrm{dlog}_\alpha(p_j)} \bmod p\end{aligned}$$

Damit gilt

$$a = \mathrm{dlog}_\alpha(\beta) = -r_0 + \sum_{j=1}^{h} e_j \cdot \mathrm{dlog}_\alpha(p_j) \bmod p-1.$$

Da wir alle Parameter auf der rechten Seite bereits berechnet haben, können wir das gesuchte $a \in \mathbb{Z}_{p-1}$ effizient bestimmen.

Auf den ersten Blick erscheint obiger Ansatz unsinnig: Um den diskreten Logarithmus eines einzelnen Elements β zu bestimmen, muss man zunächst den diskreten Logarithmus aller Elemente aus der Faktorbasis berechnen. In der Tat wird der laufzeitbestimmende Teil unseres Index-Kalkulus Algorithmus die Berechnung der diskreten Logarithmen in Schritt(1) sein. Für beide Schritte werden wir aber subexponentielle Laufzeit erzielen, wobei wir für den zweiten Schritt einen geringfügig besseren Laufzeitexponenten erhalten. Man beachte, dass man für die Berechnung mehrerer diskreter Logarithmen in $\mathbb{Z}_p^*$ bezüglich derselben Basis α nur einmal Schritt(1) durchführen muss.

Die Vorgehensweise zum Finden der $\mathrm{dlog}_\alpha(p_j)$ in Schritt(1) lässt sich analog zum Vorgehen in Kapitel 11 wieder auf das Finden genügend vieler B-glatter Elemente zurückführen. In diesem Kapitel sind unsere gesuchten B-glatten Elemente von der Form $\alpha^{r_i} \bmod p$ für $r_i \in Z_{p-1}$.

Sei $F_B = \{p_1, \ldots, p_h\}$ unsere Faktorbasis. Angenommen wir wählen $r_1, \ldots, r_k$ derart, dass die Elemente $\alpha^{r_1}, \ldots, \alpha^{r_k}$ alle B-glatt sind. Dann gilt

$$\alpha^{r_i} = \prod_{j=1}^{h} p_j^{e_{i,j}} = \prod_{j=1}^{h} \alpha^{e_{i,j}\mathrm{dlog}_\alpha(p_j)} \bmod p \quad \text{für } i = 1, \ldots, k.$$

Daraus folgen die k Gleichungen

$$\sum_{j=1}^{h} e_{i,j}\mathrm{dlog}_\alpha p_j = r_i \bmod p-1$$

in den h Unbekannten $\mathrm{dlog}_\alpha(p_1), \ldots, \mathrm{dlog}_\alpha(p_h)$. Wir definieren nun die $(k \times h)$-Matrix E mit Hilfe der Einträge $e_{i,j}$ für $i = 1, \ldots, k$ und $j = 1, \ldots, h$. Analog definieren wir den Vektor $r = (r_1, \ldots, r_k)$. Damit sind unsere gesuchten diskreten Logarithmen der Faktorbasiselemente in der Lösungsmenge des Gleichungssystems

$$E \cdot x = r \bmod p - 1$$

enthalten. Besitzt E Rang h, dann ist die Lösung des Gleichungssystems eindeutig und besteht aus dem Vektor $(\mathrm{dlog}_\alpha(p_1), \ldots, \mathrm{dlog}_\alpha(p_h))$, der unsere gesuchten Logarithmen liefert.

Man kann zeigen, dass für zufällig gewählte $r_i \in \mathbb{Z}_{p-1}$ und für Gruppenordnungen $p-1$ mit nicht zu vielen kleinen Primteilern[1] eine Wahl von $k = \mathcal{O}(h)$ genügt, damit die Matrix E vollen Rang hat. D.h. man muss im Erwartungswert $\mathcal{O}(h)$ viele B-glatte α^{r_i} finden, um h linear unabhängige Relationen zu erhalten. Wir werden uns mit diesem Thema in den Übungen näher beschäftigen.

Wir müssen noch erläutern, wie wir die Lösung x eines Gleichungssystems $E \cdot x = r \bmod p-1$ bestimmen können. In der Praxis geht man dabei wie folgt vor: Man führt eine Gaußelimination in $\mathbb{Z}_{p-1}$ durch. Wann immer beim Dividieren ein Inverses modulo $p-1$ nicht existiert, erhält man einen Teiler d von $p-1$. Nun bestimmt man eine teilerfremde Faktorisierung von $p-1$ in zwei Faktoren d^ℓ und $\frac{p}{d^\ell}$ und löst das Gleichungssystem rekursiv modulo der beiden Faktoren. Am Ende werden die Einzellösungen mit Hilfe des Chinesischen Restsatzes zu einer Komplettlösung modulo $p-1$ zusammengesetzt.

[1] Dies sind die für die Kryptographie interessanten Gruppen.

Index-Kalkulus Algorithmus

EINGABE: p prim, α Generator in $\mathbb{Z}_p^*$, $\beta = \alpha^a \bmod p$

(1) Wähle B geeignet. Setze $p_1 = (-1)$. Berechne mit Hilfe des Sieb des Erathostenes alle Primzahlen $p_2, \ldots, p_h \leq B$ und setze $F_B = \{p_1, p_2, \ldots, p_h\}$.

(2) Definiere E als leere Matrix und r als leeren Spaltenvektor.

(3) Wiederhole bis E Rang h hat:

(3a) Wähle $r_i \in \mathbb{Z}_{p-1}$ zufällig. Falls $a^{r_i} \bmod p$ ein B-glattes Element ist, finde die Faktorisierung $a^{r_i} = \prod_{j=1}^{h} p_j^{e_j} \bmod p$. Falls der Exponentenvektor $e = (e_1, \ldots, e_h)$ linear unabhängig in $\mathbb{Z}_{p-1}$ zu den Vektoren in der Matrix E ist, füge e als Zeilenvektor zu E und den Eintrag r_i zum Vektor r hinzu.

(4) Berechne die Lösung $x = (\mathrm{dlog}_\alpha(p_1), \ldots, \mathrm{dlog}_\alpha(p_h)) \in Z_{p-1}^h$ des Gleichungssystems $E \cdot x = r \bmod p - 1$.

(5) Wähle solange $r_0 \in \mathbb{Z}_{p-1}$ zufällig, bis $a^{r_0}\beta \bmod p$ ein B-glattes Element ist, d.h. $a^{r_0}\beta = \prod_{j=1}^{h} p_j^{e_j} \bmod p$. Berechne

$$\mathrm{dlog}_\alpha(\beta) = -r_0 + \sum_{j=1}^{h} e_j \cdot \mathrm{dlog}_\alpha(p_j) \bmod p - 1.$$

AUSGABE: $\mathrm{dlog}_\alpha(\beta)$

Beispiel 86 *Wir wollen den diskreten Logarithmus von* 6 *zur Basis* 2 *in* Z_{11}^* *berechnen. Dazu verwenden wir die Faktorbasis* $F_2 = \{-1, 2\}$*. Die* r_i *wählen wir zufällig aus* $\mathbb{Z}_{10}$*.*

i	r_i	α^{r_i}	e_i	*zu* E
1	3	$8 = 2^3$	$(0, 3)$	*ja*
2	4	$16 = 5$		
3	1	$2 = 2^1$	$(0, 1)$	*nein*
4	7	$128 = -4 = (-1) \cdot 2^2$	$(1, 2)$	*ja*

Man beachte, dass die Vektoren $(0, 3)$ *und* $(0, 1)$ *in* Z_{10} *linear abhängig sind, denn* $7 \cdot (0, 3) = (0, 21) = (0, 1) \bmod 10$*. Daher wird der Vektor* $(0, 1)$ *nicht in die Basis aufgenommen. Unsere Tabelle liefert das Gleichungssystem*

$$\begin{pmatrix} 0 & 3 \\ 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} dlog_2(-1) \\ dlog_2(2) \end{pmatrix} = \begin{pmatrix} 3 \\ 7 \end{pmatrix} \bmod 10$$

Durch elementare Zeilenumformungen in Z_{10} erhalten wir

$$\left(\begin{array}{cc|c} 0 & 3 & 3 \\ 1 & 2 & 7 \end{array}\right) \rightsquigarrow \left(\begin{array}{cc|c} 0 & 1 & 1 \\ 1 & 2 & 7 \end{array}\right) \rightsquigarrow \left(\begin{array}{cc|c} 0 & 1 & 1 \\ 1 & 0 & 5 \end{array}\right)$$

Daher gilt $dlog_2(2) = 1$ und $dlog_2(-1) = 5$.

Ferner ist für die Wahl $r_0 = 2$ das Gruppenelement $\alpha^{r_0}\beta \bmod 11$ ein B-glattes Element, denn:

$$\alpha^{r_0}\beta = 2^2 \cdot 6 = 24 = 2 = (-1)^0 2^1 \bmod 11.$$

Wir berechnen daher

$$dlog_2(\beta) = 0 \cdot dlog_2(-1) + 1 \cdot dlog_2(2) - 2 = (-1) = 9 \bmod 10.$$

Wir verifizieren, dass $2^9 = 2^4 \cdot 2^4 \cdot 2 = 5 \cdot 5 \cdot 2 = 3 \cdot 2 = 6 \bmod 11$ ist.

In unserem Index-Kalkulus Algorithmus müssen wir noch die Wahl des Parameters B spezifizieren. Bei dieser Wahl erhalten wir den uns schon aus Kapitel 11 bekannten Tradeoff: Wählen wir B sehr klein, dann müssen wir nur wenige B-glatte a^{r_i} finden, bis E vollen Rang hat. Andererseits ist dann die Wahrscheinlichkeit, dass ein in Schritt(3a) gewähltes Element a^{r_i} in Primfaktoren der Größe höchstens B zerfällt entsprechend klein, so dass wir Schritt(3a) sehr oft durchlaufen müssen.

Analog zu Kapitel 11 kann man zeigen, dass eine Wahl von B in der Größenordnung

$$L_p\left(\frac{1}{2}, \frac{1}{2}\right) = e^{\frac{1}{2}\sqrt{\ln p \ln\ln p}}$$

die Anzahl der Schleifendurchläufe in Schritt(3) und damit die Gesamtlaufzeit minimiert. Die Gesamtlaufzeit des Algorithmus ist

$$L_p\left(\frac{1}{2}, 2 + o(1)\right) = e^{2\sqrt{\ln p \ln\ln p}}.$$

Für Schritt(5), der eigentlichen Berechnung des diskreten Logarithmus von β, kann man eine Laufzeit von $L_p(\frac{1}{2}, \frac{3}{2} + o(1))$ zeigen.

Der asymptotisch schnellste heutzutage bekannte Algorithmus ist eine DL-Variante des in Kapitel 11 vorgestellten *Zahlkörpersiebs* mit einer Laufzeit von $L_p(\frac{1}{3}, (\frac{64}{9})^{\frac{1}{3}} + o(1))$. Dieses Zahlkörpersieb für das DL-Problem ist wiederum eine Variante unseres Index-Kalkulus Verfahrens, bei dem die Faktorbasis aus algebraischen Zahlen besteht.

Tabelle 12.1: Heutiger Stand der Rekorde bei diskreten Logarithmen

Bits	wann	Gruppe um	GIPS-J.	Algorithmus
193	1991	Lamacchia-Odlyzko	< 0.01	Gaußsche Methode
216	1995	Weber	< 0.01	Zahlkörpersieb
282	1996	Weber	0.03	Gaußsche Methode
299	1998	Joux-Lercier	0.07	Gaußsche Methode
332	1999	Joux-Lercier	0.45	Zahlkörpersieb
365	2000	Joux-Lercier	0.17	Zahlkörpersieb
398	2001	Joux-Lercier	0.35	Zahlkörpersieb
432	2005	Joux-Lercier	1.2	Zahlkörpersieb

13 Angriff auf iterierte Hashfunktion mit Multikollisionen

Wir wollen uns in diesem Kapitel mit Angriffen auf kryptographische Hashfunktion beschäftigen. Hashfunktionen finden in der Kryptographie vielfältig Einsatz:

Fingerabdruck: Um die Integrität eines großen Dokuments x zu schützen, berechnet man davon einen Hashwert $h(x) = y$, den sogenannten Fingerabdruck (Fingerprint, Message digest). Jede Änderung des Dokuments sollte ebenfalls zu einer Änderung des Fingerabrucks führen, so dass unerlaubte Änderungen detektiert werden können.

MAC: Ein MAC (Message Authentication Code) soll Authenzität und Integrität einer Nachricht x sicherstellen. Hier verwendet man eine Hashfunktion $h_k(x)$ mit geheimem Schlüssel k. Angenommen, Alice und Bob haben sich auf einen geheimen Schlüssel k geeinigt. Dann kann Alice an Bob eine authentifizierte Nachricht (x, y) mit $y = h_k(x)$ schicken. Bob berechnet $y' = h_k(x)$ und überprüft dass $y' = y$.

Unter der Annahme, dass kein Angreifer Eve ein gültiges Tupel $(x, h_k(x))$ berechnen kann, kann sich Bob sicher sein, dass die Nachricht x wirklich von Alice gesendet wurde (Authenzität). Unter der Annahme, dass Eve kein x' mit $h(x) = h(x')$ berechnen kann, kann sich Bob sicher sein, dass die Nachricht x nicht von Eve durch eine andere Nachricht x' ausgetauscht wurde (Integrität).

Hash&Sign: Bei Signaturverfahren wird eine Nachricht x beliebiger Länge zunächst auf einen String fester Länge gehasht. Dieser Hashwert wird anschließend unterschrieben. Man beachte, dass es keine geeignete Strategie ist, ein elektronisches Dokument in Blöcke zu unterteilen und diese einzeln zu unterschreiben. Zum einen wäre dies ineffizient, zum anderen könnte ein Vertragspartner einzelne Blöcke des signierten Dokuments entfernen und hätte immer noch eine gültige Unterschrift.

Das Hashen vor dem Signieren ist aber oft auch notwendig, um ungewünschte algebraische Eigenschaften des Signaturverfahrens zu verhindern. Wir wissen bereits, dass RSA aufgrund seiner multiplikativen Eigenschaft nicht sicher gegen Chosen-Message Angriffe ist, da für RSA-Signaturen $\text{sig}(m) \cdot \text{sig}(m') = \text{sig}(m \cdot m')$ gilt. Eine solche Relation soll durch Hashen der Nachricht vor dem Signieren verhindert werden.

13.1 Sicherheitsanforderungen an Hashfunktionen

Definition 87 (Hashfamilie) *Eine* Hashfamilie *ist ein* 4*-Tupel* $(\mathcal{X}, \mathcal{Y}, \mathcal{K}, \mathcal{H})$ *mit den folgenden Eigenschaften:*

1. *Der Nachrichtenraum* $\mathcal{X}$ *ist eine endliche Menge von möglichen Nachrichten.*
2. *Der Hashraum* $\mathcal{Y}$ *ist eine endliche Menge von möglichen Hashwerten.*
3. *Der Schlüsselraum* $\mathcal{K}$ *ist eine endliche Menge von Schlüsseln.*
4. *Für jedes* $k \in \mathcal{K}$ *gibt es eine effizient berechenbare Hashfunktion* $h_k \in \mathcal{H}$ *mit*

$$h_k : \mathcal{X} \to \mathcal{Y}.$$

Wenn der Schlüsselraum $\mathcal{K}$ *aus einem einzigen Element besteht, so schreiben wir auch einfach* $h \in \mathcal{H}$ *ohne Schlüssel. Ferner bezeichnen wir mit* $F^{\mathcal{X},\mathcal{Y}}$ *die Menge aller Funktionen* $\mathcal{X} \to \mathcal{Y}$*.*

Wie wir schon bei unseren Anwendungsbeispielen Fingerabdruck, MAC und Hash&Sign gesehen haben, müssen kryptographische Hashfunktionen einige Sicherheitsanforderungen erfüllen. Genauer gesagt darf keines der folgenden drei Probleme effizient lösbar sein.

Urbild Problem :
Gegeben: $h \in \mathcal{H}, y \in \mathcal{Y}$
Gesucht: $x \in X$ mit $h(x) = y$.
Wenn das Urbild Problem nicht effizient gelöst werden kann, dann bezeichnet man h auch als *Urbild resistente* Hashfunktion bzw. als *Einweg-Hashfunktion.*

Zweites-Urbild Problem :
Gegeben: $h \in \mathcal{H}, x \in \mathcal{X}$
Gesucht: $x' \neq x \in X$ mit $h(x) = h(x')$.
Wenn das Zweite-Urbild Problem nicht effizient gelöst werden kann, dann bezeichnet man h auch als *Zweites-Urbild resistent.*

Kollision :
Gegeben: $h \in \mathcal{H}$
Gesucht: $x, x' \in X$ mit $x \neq x'$ und $h(x) = h(x')$.
Wenn für h Kollisionen nicht effizient berechnet werden können, dann bezeichnet man h als *kollisionsresistent.*

Wir betrachten in diesem Kapitel zusätzlich eine natürliche Verallgemeinerung des Kollisionsproblems.

r-Kollision :

Gegeben: $h \in \mathcal{H}, r \geq 2$
Gesucht: paarweise verschiedene $x_1, \ldots, x_r \in X$ mit $h(x_1) = \cdots = h(x_r)$.
Wenn für h r-Kollisionen nicht effizient berechnet werden können, dann bezeichnet man h als *r-kollisionsresistent.*

Eine r-Kollision mit $r > 2$ bezeichnen wir auch als *Multikollision.* Die r-Kollisionsresistenz einer Hashfunktion wird z.B. für die Sicherheit des E-Cash Bezahlsystems Micromint von Rivest und Shamir benötigt.

13.2 Das Random-Oracle Modell

Wir wollen uns nun die Sicherheit einer Hashfunktion $h \in \mathcal{H}$ gegenüber den zuvor vorgestellten Problemen im sogenannten *Random-Oracle Modell* betrachten.

Definition 88 (Random-Oracle Modell:) *Im* Random-Oracle Modell *nimmt man an, dass $h \in F^{\mathcal{X},\mathcal{Y}}$ zufällig uniform gewählt ist. Ferner kennt man nicht die Beschreibung von h. Man darf lediglich Anfragen $x \in \mathcal{X}$ an das Zufallsorakel stellen, die mit $h(x)$ beantwortet werden.*

Man muss sich ein Zufallsorakel also als einen schwarzen Kasten für die Hashfunktion h vorstellen, der die Hashfunktion auswertet. Als Konsequenz aus Definition 88 erhalten wir sofort den folgenden Satz.

Satz 89 *Sei $X_0 \subset \mathcal{X}$. Angenommen $h(x)$ wurde für alle $x \in X_0$ angefragt. Dann gilt für alle $x \in \mathcal{X} \setminus X_0$ und $y \in \mathcal{Y}$:*

$$\Pr(h(x) = y) = \frac{1}{|\mathcal{Y}|}.$$

Definition 90 ((ϵ, q)-Algorithm) *Ein* (ϵ, q)-Algorithm *ist ein* Las-Vegas Algorithm *mit Erfolgswahrscheinlichkeit mindestens ϵ, der höchstens q Anfragen stellt. Ein Las-Vegas Algorithm ist ein probabilistischer Algorithm, der als Ausgabe* "Keine Lösung" *geben kann. Falls der Las-Vegas Algorithm allerdings eine Lösung ausgibt, so ist diese stets korrekt (im Gegensatz zu den probabilistischen Monte-Carlo Algorithmen, bei denen auch inkorrekte Ausgaben erlaubt sind).*

Ein (ϵ, q)-Algorithmus darf daher mit Wahrscheinlichkeit höchstens $1 - \epsilon$ die Ausgabe *"Keine Lösung"* geben. Betrachten wir den folgenden (ϵ, q)-Algorithmus PREIMAGE zum Lösen des Urbild Problems.

Algorithmus PREIMAGE

EINGABE: h, y, q

- Wähle $X_0 \subset X$ zufällig mit Größe $|X_0| = q$.
- Für alle $x \in X_0$
  - Falls $h(x) = y$ Ausgabe x, EXIT.
- Ausgabe „Keine Lösung“.

Satz 91 *Im Random-Oracle Modell ist der Algorithm PREIMAGE ein (ϵ, q)-Algorithm mit Erfolgswahrscheinlichkeit*

$$\epsilon = 1 - \left(1 - \frac{1}{|\mathcal{Y}|}\right)^q$$

Beweis: Sei $X_0 = \{x_1, \ldots, x_q\}$ und E_i das Ereignis $h(x_i) \neq y$, $i = 1, \ldots, q$. Nach Satz 89 gilt

$$\Pr(E_i) = 1 - \frac{1}{|\mathcal{Y}|} \quad \text{für } i = 1, \ldots, q.$$

Der Algorithmus ist genau dann nicht erfolgreich, wenn die Ereignisse $E_1, \ldots, E_q$ alle eintreten. Damit erhalten wir Erfolgswahrscheinlichkeit

$$\epsilon = 1 - \Pr(E_1 \cap \cdots \cap E_q) = 1 - \prod_{i=1}^{q} \Pr(E_i) = 1 - \left(1 - \frac{1}{|\mathcal{Y}|}\right)^q.$$

□

Anmerkung 92 *Man beachte, dass*

$$1 - \left(1 - \frac{1}{|\mathcal{Y}|}\right)^q = 1 - \left(1 - \binom{q}{1} \cdot \frac{1}{|\mathcal{Y}|} + \binom{q}{2} \cdot \frac{1}{|\mathcal{Y}|^2} - \ldots\right) \approx 1 - \left(1 - \frac{q}{|\mathcal{Y}|}\right) = \frac{q}{|\mathcal{Y}|}$$

für $q \ll |\mathcal{Y}|$. Für ein vorgegebenes ϵ muss man also etwa $q \approx \epsilon \cdot |\mathcal{Y}|$ Anfragen stellen. D.h. um konstante Erfolgswahrscheinlichkeit zu erhalten, muss man $\Theta(|\mathcal{Y}|)$ Anfragen an das Zufallsorakel stellen.

Übung 93 *Konstruieren Sie im Random-Oracle Modell einen (ϵ, q)-Algorithmus für das Zweite-Urbild Problem mit*

$$\epsilon = 1 - \left(1 - \frac{1}{|\mathcal{Y}|}\right)^{q-1}.$$

Wir betrachten nun den folgenden Algorithmus COLLISION für das Finden von r-Kollisionen.

Algorithmus COLLISION

EINGABE: h, q, r

- Wähle $X_0 \subset X$ zufällig mit Größe $|X_0| = q$.
- Für alle $x_i \in X_0$
  - Berechne $y_i = h(x)$.
- Falls eine Teilmenge $I = \{x_{i_1}, \ldots, x_{i_r}\} \subseteq X_0$ existiert mit $y_{i_1} = \cdots = y_{i_r}$, AUSGABE $x_{i_1}, \ldots, x_{i_r}$.
- Sonst Ausgabe "Keine Lösung".

Wir wollen uns nun überlegen, wieviele Anfragen man im Erwartungswert an das Zufallsorakel benötigt, bis man eine r-Kollision findet. Nach Satz 89 gilt für beliebige $x_{i_1}, \ldots, x_{i_r}$

$$\begin{aligned}\Pr(h(x_{i_1}) = \cdots = h(x_{i_r})) &= \Pr(y_{i_1} = \cdots = y_{i_r}) \\ &= \sum_{y \in \mathcal{Y}} \Pr(y_{i_1} = y) \cdot \ldots \cdot \Pr(y_{i_r} = y) \\ &= |\mathcal{Y}| \cdot |\mathcal{Y}|^{-r} = |\mathcal{Y}|^{1-r}.\end{aligned}$$

Wir definieren die Indikatorvariable

$$X = \begin{cases} 1 & \text{für } y_{i_1} = \cdots = y_{i_r} \\ 0 & \text{sonst} \end{cases}.$$

Damit liefert $E(X)$ die erwartete Anzahl von r-Kollisionen. Wir berechnen diesen Erwartungswert nun für q Anfragen an das Zufallsorakel:

$$E(X) = \sum_{\{x_{i_1}, \ldots, x_{i_r}\} \subseteq X_0} \Pr(y_{i_1} = \cdots = y_{i_r}) = \frac{\binom{q}{r}}{|Y|^{r-1}} \approx \frac{q^r}{|\mathcal{Y}|^{r-1}}$$

für $r \ll q$. D.h. um im Erwartungswert eine r-Kollision zu erhalten, müssen wir etwa $q = |Y|^{\frac{r-1}{r}}$ Anfragen an das Zufallsorakel stellen. Die Anzahl der zu stellenden Anfragen geht also im Random-Oracle Modell für wachsende r gegen die Größe des Hashraums $\mathcal{Y}$.

13.3 Iterierte Hash Funktionen & Multikollisionen

Nach Definition 87 bilden Hashfunktionen von einem endlichen Nachrichtenraum $\mathcal{X}$ in einen endlichen Bildraum $\mathcal{Y}$ ab. Dabei soll die Nachricht komprimiert werden, d.h. es gilt im allgemeinen $|\mathcal{X}| > |\mathcal{Y}|$. Dies nennt man auch eine *Kompressionsfunktion*. Z.B. liefert die Wahl $\mathcal{X} = \{0,1\}^{3n}$ und $\mathcal{Y} = \{0,1\}^n$ eine Kompressionsfunktion von $3n$ auf n Bit.

Nun will man in der Praxis aber Nachrichten beliebiger Länge hashen können. Dazu verwendet man heutzutage zumeist eine von Merkle und Damgård 1989 vorgeschlagene Methode, um aus einer Kompressionsfunktion f für Eingaben fester Länge eine Hashfunktion auf Eingaben x beliebiger Länge zu konstruieren:

Merkle-Damgård Konstruktion

EINGABE: x, f

1. Splitte x in Blöcke $x_1, \ldots, x_n$ und padde gegebenenfalls mit einem festgelegten String, der nur von n abhängt.
2. Setze h_0 auf einen initialen Wert IV.
3. For $i = 1$ TO n
   a) Setze $h_i = f(h_{i-1}, x_i)$.

AUSGABE: $h(x) = h_n$

Wir bezeichnen eine gemäß der Merkle-Damgård Konstruktion aufgebaute Hashfunktion auch als *iterierte Hashfunktion*. Fast alle heutzutage verwendeten Hashfunktionen wie MD4, MD5 und die SHA-Familie gehören zu der Klasse der iterierten Hashfunktionen.

Wir wollen uns nun einen Algorithmus zum Konstruieren von Multikollisionen bei iterierten Hashfunktionen betrachten. Dieser Algorithmus wurde 2004 von Joux vorgeschlagen.

Sei h eine iterierte Hashfunktion mit Kompressionsfunktion f. Ferner sei der Hashraum $\mathcal{Y}$ unserer Hashfunktion n Bit groß. Angenommen wir verfügen über einen Kollisionsalgorithmus K_A, der bei Eingabe f, h_i zwei Blöcke x, x' ausgibt, die eine Kollision für die Kompressionsfunktion f liefern, d.h. $f(h_i, x) = f(h_i, x')$. Dabei setzen wir voraus, dass die Blocklänge – d.h. die Bitlänge von x bzw. x' – signifikant größer als n Bit ist, um die Existenz von Kollisionen zu garantieren.

Der Algorithmus K_A kann z.B. den im letzten Abschnitt vorgestellten Algorithmus COLLISION zum Finden von Kollisionen verwenden. In diesem Fall würde K_A etwa

$q \approx 2^{\frac{n}{2}}$ Anfragen an f benötigen. Wir wollen nun zeigen, wie man eine 4-Kollision durch 2 Anfragen an den Algorithmus K_A konstruieren kann.

Wir starten mit dem initialen Wert IV. Ein erster Aufruf von K_A liefert uns zwei Blöcke x_1, x_1' mit

$$f(IV, x_1) = f(IV, x_1').$$

Wir rufen nun unseren Algorithmus K_A für den Wert $f(IV, x_1)$ auf. K_A liefert uns zwei Werte x_2, x_2' mit

$$f(f(IV, x_1), x_2) = f(f(IV, x_1), x_2') = f(f(IV, x_1'), x_2') = f(f(IV, x_1'), x_2).$$

Damit erhalten wir also eine 4-Kollision für die Nachrichten $x_1||x_2$, $x_1||x_2'$, $x_1'||x_2'$, $x_1'||x_2$ [1]. Man beachte, dass wir den Paddingprozess vernachlässigen können, da wir Kollisionen von Nachrichten derselben Länge erzeugen und daher das Padding identisch ist. Unsere Nachrichten bleiben aber Kollisionen, wenn man identische Bitstrings an die Nachrichten anhängt.

Wir wollen nun unsere Attacke für t Aufrufe des Algorithmus K_A verallgemeinern.

Satz 94 (Joux 2004) *Sei h eine iterierte Hashfunktion mit Kompressionsfunktion f. Sei K_A ein Algorithmus, der bei Eingabe f, h_i zwei Werte x, x' liefert mit $f(h_i, x) = f(h_i, x')$. Dann kann durch t Aufrufe von K_A eine 2^t-Kollision für h gefunden werden.*

Beweis: Wir betrachten den folgenden Algorithmus

2^t-KOLLISION

EINGABE: h, f, t

1. Setze $h_0 = \text{IV}$.
2. FOR $i = 1$ TO t
   a) Bestimme durch Aufruf den K_A zwei Werte x_i, x_i' mit $f(h_{i-1}, x_i) = f(h_{i-1}, x_i')$.
   b) Setze $h_i = f(h_{i-1}, x_i)$.

AUSGABE: $x_1, x_1', \ldots, x_t, x_t'$

Nach Konstruktion liefert jede Nachricht der Form $b_1||b_2||\ldots||b_t||\text{Padding}$ mit $b_i \in \{x_i, x_i'\}$ denselben Hashwert. Da es für jeden Block b_i zwei verschiedene Möglichkeiten gibt, erhalten wir insgesamt eine 2^t-Kollision. □

[1] Wir bezeichnen mit $x_1||x_2$ die Konkatenation von x_1 und x_2.

13.4 Angriff auf kaskadierte Hashfunktionen mittels Multikollisionen

Um die Kollisionsresistenz von Hashfunktionen zu vergrößern, verwendet man sogenannte kaskadierte Hashfunktionen. Seien g und h zwei Hashfunktionen, dann besteht die kaskadierte Hashfunktion $g||h$ aus der Konkatenation der Ausgaben von g und h. Sei der Hashraum von g und h von der Größe n_g bzw. n_h Bit. Dann kann man Kollisionen der einzelnen Hashfunktion im Random-Oracle Modell mit $2^{\frac{n_g}{2}}$ bzw. $2^{\frac{n_h}{2}}$ Hashanfragen erzeugen.

Die Konkatenation $g||h$ beider Hashfunktionen erfordert für unseren Algorithmus 2-COLLISION aus dem vorangegangenen Abschnitt $2^{\frac{n_g+n_h}{2}}$ Aufrufe der konkatenierten Hashfunktion. Wir werden nun sehen, dass es einen Algorithmus zum Erzeugen einer Kollision mit deutlich weniger Aufrufen gibt, sofern eine der beiden Hashfunktionen g oder h eine iterierte Hashfunktion ist.

Satz 95 (Joux 2004) *Sei g eine iterierte Hashfunktion mit n_g-Bit Hashraum und h eine Hashfunktion mit n_h-Bit Hashraum. Dann kann eine Kollision der konkatenierten Hashfunktion $g||h$ mit erwartet*

$$\Theta(n_h 2^{\frac{n_g}{2}} + 2^{\frac{n_h}{2}})$$

Hashfunktionsaufrufen gefunden werden.

Beweis: Wir betrachten den folgenden Algorithmus.

Algorithmus Kaskadierte Hashfunktion

EINGABE: g, h, n_g, n_h

(1) Finde eine 2^t-Kollision mit $t = \lceil \frac{n_h}{2} \rceil$ von g mit Algorithmus 2^t-KOLLISION.

(2) Finde unter den 2^t Kollisionen für g eine Kollision x, x' für h.

AUSGABE: x, x' mit $g(x)||h(x) = g(x')||h(x')$

Unter den 2^t Kollisionen für g ist mit konstanter Wahrscheinlichkeit ϵ eine Kollision für h, da $t \geq \frac{n_h}{2}$. Wir können durch geringfügiges Erhöhen von t diese Erfolgswahrscheinlichkeit ϵ beliebig nahe an 1 bringen.

Nach Satz 94 benötigen wir für Schritt(1) lediglich t Aufrufe eines Kollisionsalgorithmus K_A für die Kompressionsfunktion von g. Einen trivialen Kollisionsalgorithmus erhalten wir durch unseren Algorithmus 2-COLLISION mit erwarteter Anzahl $\Theta(2^{\frac{n_g}{2}})$ Aufrufen. Damit benötigen wir in Schritt(1) erwartet $\Theta(n_h 2^{\frac{n_g}{2}})$ Aufrufe.

In Schritt(2) müssen wir h für alle 2^t Kollisionen auswerten. Dies kostet $\mathcal{O}(2^{\frac{n_h}{2}})$ Aufrufe von g. Sollte h selbst eine iterierte Hashfunktion sein, dann können wir die Aufrufe an die Kompressionsfunktion von h zählen. Wir müssen 2^t Nachrichten mit jeweils Länge $n_g t$ auswerten. Is $n_g = n_h$, dann erfordert dies naiv implementiert $t2^t$ Aufrufe der Kompressionsfunktion von h. Wir können aber die spezielle Struktur der in Schritt(1) konstruierten 2^t Kollisionen ausnutzen, um lediglich $\mathcal{O}(2^t)$ Aufrufe der Kompressionsfunktion zu verwenden. (Dies zu zeigen, ist eine Übungsaufgabe.)

Insgesamt erhalten wir aus Schritt(1) und (2) $\Theta(n_h 2^{\frac{n_g}{2}} + 2^{\frac{n_h}{2}})$ Hashaufrufe. □