Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Theoretische Informatik

Konstruktion von Hashfunktionen

Diplomarbeit

von

Sidi Mohamed El yousfi Alaoui

Betreuer:
Prof. Dr. J. Buchmann
Dip. Math. Erik Dahmen

15. September 2007

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne fremde Hilfe und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, September 2007

Sidi Mohamed El yousfi Alaoui

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Aufgrund der zunehmenden Verbreitung des Internets ist der Datenaustausch auf elektronischen Wegen selbstverständlich geworden. Es ist also notwendig, die Daten in elektronischer Form gegen Manipulationen und Betrugsmöglichkeiten zu schützen.
In der digitalen Welt wurde die Funktionalität der handlichen Unterschriften oder „Verhandlungen unter vier Augen“ durch die digitale Signatur ersetzt. Die verwendeten Signaturverfahren müssen geeignet sein, alle Manipulationen an signierten Daten zu entdecken und die erzeugten Signaturen nicht einfach zu verfälschen.
In der Praxis wird die Anwendung der digitalen Signatur auf umfangreiche Daten sehr aufwendig. Deshalb verwendet man Hashfunktionen, welche die Aufgabe haben, aus einem Dokument beliebiger Länge einen eindeutigen Wert („digitalen Fingerabdruck“) fester Länge zu erzeugen. Dieser Wert wird als Hashwert bezeichnet, welcher mit Hilfe eines bestimmten Signaturverfahrens signiert wird. So erhält man eine digitale Signatur unabhängig von der Länge der zu signierenden Daten.
Da der Definitionsbereich solcher Funktionen in der Regel erheblich größer als der Bildbereich ist, können mehrere Dokumente den gleichen Hashwert besitzen. Solche Fälle werden Kollisionen genannt, welche zur Signaturverfälschung ausgenutzt werden können.
Es sollten deshalb technische Maßnahmen ergriffen werden, um die Wahrscheinlichkeit des Auftretens von Kollisionen zu verringern. Aus diesem Grund sollten die Hashfunktionen bestimmte Sicherheitsanforderungen erfüllen. Daneben sollte auch auf die Performance geachtet werden, damit die Rechenzeit bei der Anwendung der Hashfunktionen auf große Dokumente so gering wie möglich wird.
Hauptsächlich gibt es drei Hauptklassen von Hashfunktionen. Die Erste heißt Blockchiffren-basierte Hashfunktion. Dabei werden symmetrische Verfahren wie DES eingesetzt. In der Praxis sind allerdings solche Hashfunktionen sehr aufwendig. Die zweite Klasse enthält die dedizierte Hashfunktionen, welche nach einem Ad-Hoc-Design konstruiert sind. Mit anderen Worten verwenden solche Funktionen übliche Rechenoperationen wie AND, OR und XOR und werden in der Praxis am meisten eingesetzt. Einige Beispiele davon sind MD4, MD5, SHA1. Die dritte Klasse, welche als Schwerpunkt dieser Arbeit gilt, enthält Hashfunktionen, die eine algebraische Struktur besitzen und deren Sicherheit auf zahlentheoretischen Problemen reduziert wird.

Das Ziel dieser vorliegenden Arbeit ist, eine allgemeine Übersicht zu geben, wie die Hashfunktio-

nen auf der Basis der algebraischen Strukturen aufgebaut sind und wie ihre Sicherheit mit den zugrundeliegenden zahlentheorischen Problemen zusammenhängt.

1.2 Aufbau der Arbeit

Die vorliegende Diplomarbeit ist wie folgt aufgebaut:

Im Kapitel 2, *Grundlagen und Anwendung von Hashfunktionen,* werden die wichtigen Grundbegriffe sowie die Sicherheitseigenschaften von Hashfunktionen und darauf duchgeführten Angriffe vorgestellt und erläutert. Am Ende des Kapitels werden einige Beispiele von Anwendungen der Hashfunktionen gegeben.

In dem Kapitel 3, *Hashfunktionen auf der Basis modularer Arithmetik,* werden zunächst Grundlagen der Zahlentheorie eingeführt. Danach werden die allgemeinen Überlegungen zur Konstruktion von Hashfunktionen auf Basis modularer Arithmetik vorgestellt und diskutiert. Schließlich befasst sich der letzte Teil mit konkreten Beispielen von solchen Hashfunktionen.

Das Kapitel 4, *Hashfunktionen auf Basis der Rucksackprobleme*, gibt zunächst eine Einführung in die Gitter- und Knapsacktheorie und danach deren Anwendung um sichere Hashfunktionen zu realisieren. Am Ende werden zwei konkrete Beispiele gegeben und deren Sicherheit besprochen.

In Kapitel 5, *Ausblick*, wird das Wesentliche dieser Arbeit zusammengefasst.

Kapitel 2

Grundlagen und Anwendung von Hashfunktionen

Zu den Hauptaufgaben der theoretischen Informatik gehört die Suche nach sicheren Hashfunktionen, welche in vielen Teilgebieten in der Kryptographie eingesetzt werden. In diesem Kapitel werden zunächst die Grundbegriffe sowie die Sicherheitseigenschaften solcher Funktionen gegeben und danach deren Einsatz und Anwendungen im Bereich der IT-Sicherheit vorgestellt und erklärt.

2.1 Grundlagen und Definitionen

Definition 2.1
Sei A ein Algorithmus.

- A bezitzt eine *polynomiale* Laufzeit genau dann, wenn die Zeitaufwand von A nach oben *polynomial* beschränkt ist. In diesem Fall ist die Komplexität von A aus der Klasse $O(n^c)$.
- A bezitzt eine *exponentielle* Laufzeit genau dann, wenn der Zeitaufwand nach unten *exponentiell* beschränkt ist. In diesem Fall ist die Komplexität von A aus der Klasse $\Omega(2^{n^c})$.

Wobei n die Länge der Input und $c \geq 0$ sind.

Definition 2.2
Ein Problem heißt *effizient* oder *polynomial* lösbar, wenn ein Algorithmus existiert, der das Problem mit polynomialer Laufzeit löst. Eine solche Klasse von Problemen wird als „Klasse P“ bezeichnet. Ein Problem heißt praktisch *unlösbar*, wenn jeder Algorithmus, welcher das Problem löst, mindestens eine exponentielle Laufzeit benötigt. Eine solche Klasse von Problemen wird als „Klasse NP“ bezeichnet.

Beispiel 2.1
Das Produkt zweier Primzahlen p und q ist ein P-Problem, aber die Faktorisierung einer großen zusammengesetzten Zahl ist im Allgemeinen ein NP-Problem.

Definition 2.3 (Kompressionsfunktion)
Sei Σ ein Alphabet (d.h. eine Menge von Zeichen).

Σ^n bezeichnet die Menge aller Zeichenfolgen der Länge n mit $n \in \mathbb{N}$.
Eine *Kompressionsfunktion* ist eine Abbildung, welche Strings fester Länge auf Strings fester und kürzer Länge abbildet. Mathematisch beschrieben sieht diese Funktion folgendermaßen aus:

$$f : \Sigma^m \to \Sigma^n, \quad n, m \in \mathbb{N} \text{ mit } m \geq n.$$

Eine solche Funktion soll folgende Eigenschaften aufweisen:

- Der Wert $y = f(x)$ soll in polynomialer Laufzeit berechnet werden.
- Sie soll eine Einwegfunktion sein, d.h. soll die Berechnung eines Urbilds x zu einem gegebenen Funktionswert y schwer sein oder in einer exponentiellen Laufzeit erfolgen.

Beispiele 2.2
Um diese Definition zu verdeutlichen, wird ein Beispiel für eine Einwegfunktion erläutert:

1. Eine Vase aus zehn Metern Höhe auf den Boden fallen zu lassen, ist sehr einfach, jedoch ist es sehr schwer, diese Vase aus den zerbrochenen Einzelteilen wieder zusammenzusetzen.
2. Die Abbildung, die jedem Wort $b_1 b_2 \cdots b_m$ aus $\{0,1\}^m$ die Zahl $b_1 \oplus b_2 \cdots \oplus b_m \bmod 2$ zuordnet, ist eine Einwegfunktion.

Bemerkung
Eine entscheidende Frage in der Kryptographie lautet: *Existiert* tatsächlich eine Einwegfunktion. Eine *notwendige* Bedingung für die Existenz ist $P \neq^1 NP$, aber ob sie hinreichend ist, diese Frage ist immer noch offen.

Definition 2.4
Eine *Falltürfunktion* oder *Trapdoor-Funktion* ist eine Einwegfunktion, so dass ihre Invertierung bei Kenntnis einer Geheiminformation wieder leicht möglich ist.

Beispiel 2.3
Die RSA-Verschlüsselung $E(M) = M^e \bmod n$ ist eine Einwegfunktion, solange die privaten Schlüssel unbekannt ist. Bei Kenntnis der privaten Schlüssel (Falltür) ist E aber leicht umzukehren.

Definition 2.5 (Hashfunktion)
Eine *Hashfunktion* ist ein Verfahren zur Komprimierung von Daten beliebiger Länge anhand einer Kompressionsfunktion, so dass die ursprünglichen Daten nicht wiederherstellbar sind.
Die Hashfunktion h ist also eine Abbildung, die Strings *beliebiger* Länge auf Strings *fester* Länge abbildet. Formal heißt das:

$$h : \Sigma^* \to \Sigma^n, \quad n \in \mathbb{N}.$$

Der Ausgabewert wird als *Hashwert* oder *Prüfsumme* bezeichnet.
Σ^* bezeichnet die Menge aller Zeichenfolgen beliebiger Länge.
Nach der Einführung der Grundbegriffe werden im Folgenden Abschnitte die Grundkriterien und die Sicherheitseigenschaften der Hashfunktionen vorgestellt und erläutert.

[1] Das ist ein berühmtes Problem in der theoretischen Informatik, für welches es bis heute keinen Beweis gibt.

2.2 Grundkriterien für Hashfunktionen

2.2.1 Schnelle Ausführbarkeit

Da die Hashfunktion auf große Datenmengen angewendet wird, soll die Hashberechnung relativ schnell und einfach durchgeführt werden, d.h. der Rechenaufwand darf nur in einer polynomialen Laufzeit in Bezug auf die Bitlänge von der zu hashenden Daten steigen.

2.2.2 Zufälligkeit und Gleichverteilung

Eine Hashfunktion soll pseudozufällig sein, d.h. die ausgegebenen Hashwerte sollen statistisch gleichmäßig über den möglichen Wertebereich verteilt werden. Jeder Hashwert soll mit einer Wahrscheinlichkeit von $\frac{1}{2^{L_H}}$ auftreten (L_H bezeichnet die Länge des Hashwertes), ansonsten hätte ein Angreifer bei bestimmten Hashwerten, die statistisch gesehen öfter vorkommen, die Chance, durch Raten und Probieren ein Urbild zum Hashwert zu finden. Gute Hashfunktionen sollen einen Hashwert erzeugen, wobei eine Modifikation eines Bits von einer Nachricht zur Änderung der Hälfte mindestens von den Bits des Hashwertes führt.

2.3 Sicherheitsanforderungen an kryptographische Hashfunktionen

In der Kryptographie soll eine Hashfunktion zusätzliche Sicherheitseigenschaften haben, damit sie sich für verschiedene Anwendungen in der Informationssicherheit eignet. Für eine Hashfunktion stehen verschiedene Sicherheitseigenschaften zur Verfügung und für jede Eigenschaft gibt es bestimmte Arten von Angriffe, die man berücksichtigen muss und für jeden Angriff gibt es geeignete Gegenmaßnahmen, die man treffen muss. Im Folgenden werden die möglichen Sicherheitsstufen vorgestellt, welche eine sichere Hashfunktion erfüllen soll.

2.3.1 Einwegeigenschaft

Eine Hashfunktion h ist eine *Einwegfunktion* (eng.: preimage resistant), wenn es keine effizient berechenbare inverse Funktion von h gibt, welche zu jedem vorgegebenen Hashwert H eine sinnvolle Nachricht M bestimmen kann, so dass $h(M) = H$ gilt.

2.3.2 Schwach Kollisionsresistenz

Eine Hashfunktion h heißt *schwach Kollisionsresistenz* (eng.: second preimage resistant), wenn es nicht effizient (d.h. in polynomiale Zeit) möglich ist, zu einer vorgegebenen Nachricht M eine zweite Nachricht M' mit $h(M) = h(M')$ zu finden.

2.3.3 Stark Kollisionsresistenz

Eine Hashfunktion h heißt *stark Kollisionsresistenz* (eng.: collision resistant), wenn es unmöglich ist, in polynomialer Zeit eine Kollision zu finden, d.h. zwei beliebige unterschiedliche Nachrichten M und M' mit identischem Hashwert zu finden.

Definition 2.6 (kryptographische Hashfunktion)
Eine kryptographische Hashfunktion ist eine Einwegfunktion, die stark (und damit auch schwach) Kollisionsresistenz ist.

2.3.4 Unempfindlichkeit des Signaturverfahrens

Im Zusammenhang mit dem Signaturverfahren muss beachtet werden, dass die verwendete Hashfunktion höhere Sicherheitsanforderungen erfüllen muss. Diese soll anfällig gegen die Unempfindlichkeit des Signaturverfahrens sein, d.h. es muss schwierig sein, zwei Nachrichten zu bestimmen, deren Hashwerte einer mathematischen Beziehung genügen, mit dem Ziel, die Signatur einer Nachricht aus der anderen zu berechnen. Diese Anforderung hängt sowohl von der Hashfunktion als auch von dem verwendeten Signaturverfahren ab.

Bemerkungen

1. h stark Kollisionsresistenz $\Rightarrow$ h schwach Kollisionsresistenz.
2. Falls schwach bzw. stark Kollisionsresistenz h existiert, so existiert eine h' schwach bzw. stark Kollisionsresistenz, die keine Einwegfunktion ist.
3. Ist h schwach Kollisionsresistenz und hat jedes y aus Σ^n mindestens zwei Urbilder unter h, so ist h eine Einwegfuntion.
4. Existiert eine Einwegfuntion $h : \Sigma^* \to \Sigma^n$, so existiert eine Einwegfuntion $h' : \Sigma^* \to \Sigma^n$, die nicht schwach Kollisionsresistenz ist.

Beweis

1. trivial.
2. Sei h schwach bzw. stark Kollisionsresistenz Hashfuntion. Definiere $h' : \Sigma^* \to \Sigma^{n+1}$ mit:

   $$h'(x) = \begin{cases} 1x & \text{falls x die Bitlänge } n \text{ hat .} \\ 0h(x) & \text{sonst.} \end{cases}$$

   h' ist Kollisionsresistenz, aber für mindestens die Hälfte aller Bilder von h' (nämlich alle denen, die mit 1 beginnen) kann man leicht das Urbild finden.
3. Angenommen es existiert ein probabilistischer polynomialer Algorithmus A, welcher mit einer höheren Wahrscheinlichkeit zu vorgegebenen x ein $x' = A(h(x))$ mit $h(x) = h(x')$ liefert.
   Da jedes $h(x)$ mindestens zwei Urbilder besitzt, folgt daraus, dass die Wahrscheinlichkeit für $x \neq x'$ größer gleich 2 ist. Dies führt zu einem probabilistischen polynomialen Algorithmus zur Erzeugung von Kollisionen.
4. Sei $h' : \Sigma^* \to \Sigma^n$ mit:

   $$h'(x) = \begin{cases} h(x) & \text{falls x mit 1 beginnt.} \\ h(1x) & \text{falls x mit 0 beginnt.} \end{cases}$$

   Dann ist h' eine Einwegfunktion, aber für alle x, die mit 10 beginnen (1/4 der Fälle), $x = 1$ x' mit $x' = 0 \cdots$, ist also $h'(x) = h'(x')$.

2.4 Angriffe auf Hashfunktionen

Eine Hashfunktion kann in der Kryptographie als sicher eingestuft werden, wenn sie gegen alle Angriffsformen immun ist. Im Folgenden werden die möglichen Angriffe, welche im Laufe dieser Arbeit gebraucht werden, beschrieben.

2.4.1 Random Attack

Der Random Attack (Brute-Force-Angriff) ist der einfachste Angriff (einige halten ihn für "weder intelligent noch effektiv"), dessen Erfolgswahrscheinlichkeit unabhängig von einem speziellen Hash-Algorithmus und der Struktur der Input-Daten ist.
Beim Random Attack besteht die Idee darin, Hashwerte für verschiedene Nachrichten auszurechnen in der Hoffnung, dass sich einmal der gesuchte Hashwert ergibt.
Sind die Hashwerte von 0 bis 2^n statistisch gleichverteilt, so hat der Random Attack die maximale Komplexität $O(2^n)$, mit n die Länge des Hashwertes. Um diesen Angriff abzuwehren, muss man einfach die Bitlänge des Hashwertes bzw. den Wertebereich der Hashfunktion so groß wählen, dass die Berechnung von 2^n verschiedenen Hashwerten nicht innerhalb einer polynomialen Laufzeit möglich ist.
Eine ideale Hashfunktion sollte die Eigenschaft haben, dass der Random Attack die maximale Komplexität 2^n hat.

2.4.2 Birthday Attack

Die Idee Beim Birthday Attack besteht darin, aus einer Menge von Hashwerten zwei Hashwerte zu finden, welche eine gleiche Nachricht als Urbild haben. Mittels des Birthday Attack sind idealerweise $2^{n/2}$ Berechnungsschritte für eine Hashfunktion mit n-Bits als Hashwertlänge benötigt, um eine Kollision zu finden. Gegenwärtig betrachtet man die Zahl 2^{80} als minimale Zahl für das Sicherheitsniveau[2] einer Hashfunktion. Dieser Angriff beruht auf das Geburtstagsparadox.

Satz 2.1 (Geburtstagsparadox)
Sei eine Urne, bestehend aus m unterscheidbaren Kugeln, aus welcher k Kugeln zufällig, unabhängig und mit Zurücklegen aus dieser Urne gezogen werden. Also Ist $k \geq \left(\frac{1}{2} + \sqrt{\frac{1}{4} + 2 \cdot m \cdot \ln 2}\right)$, so ist $P(m,k) \geq \frac{1}{2}$, wobei $P(m,k)$ die Wahrscheinlichkeit bezeichnet, dass mindestens eine Kugel gezogen wird.

Beweis
Bezeichene mit $Q(m,k)$ die Wahrscheinlichkeit des Gegen Ereignisses, dass alle gezogenen Kugeln verschieden sind. Es gilt:

$$Q(m,k) = \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdots \frac{m-(k-1)}{m}$$

Wobei $\frac{m-(i-1)}{m}$ die Wahrscheinlichkeit für i-ite Ziehung ist. Daraus folgt

$$\mathrm{Q(m,k)} = \prod_{i=1}^{k-1} (1 - \tfrac{i}{m}).$$

[2]Die Anzahl der Berechnungsschritte, um eine Kollision zu finden.

Unter Verwendung $(1-x) \leq e^{-x}$ für alle x $\in \mathbb{R}$ und
$\sum_{i=1}^{k-1} i = \frac{(k-1)k}{2}$ und $(k-\frac{1}{2})^2 - \frac{1}{4} = k^2 - k = (k-1)k$

erhalten wir $Q(m,k) \leq \prod_{i=1}^{k-1} e^{\frac{-i}{m}} = e^{\frac{-i}{m}\sum_{i=1}^{k-1} i} = e^{-\frac{(k-1)k}{2m}} = e^{-\frac{(k-\frac{1}{2})^2-\frac{1}{4}}{2m}}$

Da $k \geq \left(\frac{1}{2} + \sqrt{\frac{1}{4} + 2 \cdot m \cdot \ln 2}\right)$ ist, so gilt $Q(m,k) \leq e^{-\frac{\frac{1}{4}+2m\ln 2-\frac{1}{4}}{2m}} = e^{-\ln 2} = \frac{1}{2}$.

■

Satz 2.2 (Birthday Attack)
Seien $x_1, \cdots, x_k$ verschiedene Elemente aus $\{0,1\}^*$, deren Funktionswerte als $h(x_1), \cdots, h(x_k)$ aus $\{0,1\}^n$ bezeichnet sind. Suchen wir dann nach einer Kollision.
Nehmen wir an, dass die $h(x_1), \cdots, h(x_k)$ unabhängig und gleichmäßig auf der Menge $\{0,1\}^n$ verteilt sind.
Ist dann $n \geq 18$ und $k \geq 1.18 \cdot 2^{n/2}$, so ist die Wahrscheinlichkeit, dass eine Kollision gefunden wird, mindestens 1/2 beträgt.

Beweis
Setze $m = 2^n$. Diese m-Werte entsprechen den m verschiedenen Kugeln im vorhergehenden Satz (Birthday Attack). Die Hashberechnungen können als die Ziehung von k-Kugeln aufgefasst werden. Damit eine Kollision mit einer Wahrscheinlichkeit größer gleich 1/2 auftritt, muss für $n \geq 18$ gelten:

$$\begin{aligned} \frac{1}{2} + \sqrt{\frac{1}{4} + 2 \cdot 2^n \cdot \ln 2} &\leq \frac{1}{2} + \sqrt{\frac{1}{4}} + \sqrt{2 \cdot 2^n \cdot \ln 2} \\ &\leq 1 + 1.1775 \cdot 2^{n/2} \\ &\leq 1.18 \cdot 2^{n/2} \\ &\leq k \end{aligned}$$

Nach Geburtstagsparadox-Satz folgt dann die Behauptung.

■

Bemerkung
Um die Erfolgswahrscheinlichkeit der Birthday Attacks zu verringern, muss n so groß gewählt werden, dass die Berechnung sowie die Speicherung von $2^{n/2}$ Hashwerte unmöglich ist.

2.4.3 Direkt Attack

Beim Direkt-Attack **(DA)** geht es darum, zu gegebenen Werten M_i und H_{i-1} jeweils ein M_i' $(M_i \neq M_i')$ mit $f(M_i, H_{i-1}) = f(M_i', H_{i-1})$ zu finden.
Durch die Einwegeigenschaft von f kann man diesen Angriff abwehren. Es gibt auch eine andere Möglichkeit zur Abwehr dieses Angriffs, indem man Redundanz-Bits anwendet, d.h. Sicherung der Nachricht bzw. Blöcke durch gezieltes Einfügen zusätzlicher Bits.

2.4.4 Forward Attack

Die Idee beim Forward Attack **(FA)** besteht darin, zu gegebenen Werten M_i, H_{i-1} und H'_{i-1} ein M'_i ungleich M_i zu finden, so dass $f(M_i, H_{i-1}) = f(M'_i, H'_{i-1})$.
Zur Abwehr dieses Angriffs kann man auch hier einfach Redudanzbits in die M_i-Blöcke anwenden.

2.4.5 Backward Attack

Beim Backward Attack **(BA)** versucht man rückwärts die Iteration der Funktion f zu durchlaufen, d.h. zu gegebenen H_i versucht man ein Paar (M_i, H_{i-1}) mit $f(M_i, H_{i-1}) = H_i$ zu konstruieren.
Durch Redundanz-Bits kann dieser Angriff abgewehrt werden.

2.4.6 Correcting Block Attack

Beim diesem Angriff geht es darum, einen Block zu einem Nachricht einzufügen, um einen bestimmten Hashwert zu erhalten.
In der Praxis versucht ein Angreifer mit Hilfe des Correcting-Block Angriffs die inhaltliche Bedeutung eines Dokuments durch eine kleine Manipulation nicht merkwürdig ändern.

2.4.7 Fixed Point Attack

Die Idee beim Fixed Point Attack **(FPA)** besteht darin, zwei Werte M_i und H_{i-1} zu finden, so dass $f(M_i, H_{i-1}) = H_{i-1}$. In diesem Fall wird der Block M_i in die Nachricht beliebig oft hintereinander eingebaut, ohne ihren Hashwert zu verändern. Zur Konstruktion eines zweiten Urbildes sieht der Fixpunkt so aus, dass man den Wert H_{i-1} vorgegeben hat und ein passendes M_i sucht. Auch hier helfen die Redundanz-Bits, um diesen Angriff abzuwehren.

2.4.8 Permutation Attack

Beim Permutation Attack **(PA)** geht es darum, dass man aus der Beziehung $H_i = H_{i-1} \oplus g(M_i)$ ein M_i bestimmen kann, wobei g eine Einwegfunktion ist.

2.4.9 Fixed Point Attack

Es geht hier darum, zwei werte M_i und H_{i-1} zu finden, so dass $f(M_i, H_{i-1}) = H_{i-1}$.
Um dieses Attacks abzuwehren, kann man Redundanz-Bits einfügen.

2.5 Anwendung von Hashfunktionen

Wie schon erwähnt bilden die sicheren Hashfunktionen gegenwärtig sehr wichtige Bestandteile der Kryptographie und werden in vielen Teilgebieten eingesetzt [8]. In diesem Abschnitt werden die wichtigsten Anwendungen von Hashfunktionen vorgestellt.

2.5.1 Message Authentication Code

Ein Message Authenticikation Code (MAC) ist eine Hashfunktion, die zusätzlich einen geheimen Schlüssel K enthält. Dieser Schlüssel ist nur den beiden Kommunikationspartnern A und B bekannt. Mit Hilfe von MACs können A und B Authentizität und auch die Integrität von den ausgetauschten

Daten gewährleisten.
Stimmt der vom Datenempfänger berechnete MAC mit dem Übertragenen überein, so wurden die Daten nicht verändert.

2.5.2 Digitale Signatur

Es handelt sich hier um eine Erzeugung einer digitalen Signatur durch eine Hashfunktion. Um ein Dokument zu signieren, wird dessen Hashwert berechnet. Dieser Wert wird signiert und dieses Ergebnis wird danach an das Originaldokument angehängt und anschliessend an den Empfänger elektronisch übermittelt. Genauso wird vom Empfänger mit der Hashfunktion auch der Hashwert des mitgeschickten Dokuments berechnet, auf den sich die digitale Signatur beziehen soll, danach werden sie verglichen. Stimmen die zwei Werte überein, so ist die digitale Signatur authentisch.
Mit Hilfe dieses Verfahren spart man Speicherplatz und Rechenzeit.

2.5.3 Erzeugung von Passwörtern

Ein starkes Passwort kann aus einer langen Zeichenkette statt eines einzelnen Wortes gebildet werden. Solche lange Ketten sind im Allgemeinen als Passwörter nicht erforderlich und unerwünscht. Mit Hilfe einer Hashfunktion können sie in ein kürzeres Passwort konvertiert werden. Dieses Verfahren nennt man key crunching.

2.5.4 Vertraulichkeit der Passwörter

Durch Passwörter kann man die Authentifizierung verwirklichen, so dass keine unautorisierten Zugriffe auf ein System möglich wären. Um die Vertraulichkeit solcher Passwörter zu gewährleisten, setzt man die Hashfunktionen ein. Die Idee besteht darin, dass man nicht die Passwörter der berechtigten Benutzer selbst auf eine Schreib- und Lesezugriff geschützte Datei speichern muss, sondern nur deren Hashwerte, da nach diesem Prinzip eine direkte Wiederherstellung eines Passwortes aus der Kenntnis seines Hashwertes nicht ohne erheblichen Aufwand möglich ist.
Das ganze Szenario funktioniert wie folgt:
Bei der Zugangskontrolle gibt ein Benutzer ein Passwort ein, daraus wird ein Hashwert berechnet. Das System vergleicht diesen Hashwert mit dem in der Datei abgespeicherten Wert dieses Benutzers. Stimmen die zwei Werte überein, erlangt man den Zugang. Im Fehlerfall wird der Benutzer informiert, z.B. mit der Meldung „login failed“.
Ein Beispiel hierfür sind Passwort-geschützte Karten mit Magnetstreifen (z.B. Bank- oder Kreditkarten), die nach dem gleichen Prinzip funktionieren, d.h. der Magnetstreifen enthält nicht das Passwort selbst, sondern nur dessen Hashwert.

2.5.5 Softwareschutz

Eine Software kann gegen unerwünschte Veränderungen geschützt werden (z.B gegen Viren), indem ein Hashwert aus dem Software erzeugt wird und dann an einem sicheren gespeichert wird. Ein Administrator kann danach verifizieren, ob es eine Modifikation gab oder nicht.
So ähnlich können auch sowohl die Bank-Transaktionen vor Verfälschung als auch mobile Code vor Veränderungen der Arbeitsweise geschützt werden.

Kapitel 3

Hashfunktionen auf der Basis modularer Arithmetik

Die meisten bekannten kryptographischen Hashfunktionen sind im Allgemeinen auf elementaren Bitoperationen wie AND,OR und XOR basiert, da solche Operationen einfach in Hardware und Software zu implementieren sind. In vielen Fällen wurde auch mit modularen Rechenoperationen kombiniert. Dieser Typ von Hashfunktionen wurde als *Hashfunktionen auf der Basis der modulare Arithmetik* oder einfach *modulare Hashfunktionen* bezeichnet.
Die Grundidee zur Konstruktion von Hashfunktionen durch Funktionen Modulo einer natürlichen Zahl besteht darin, dass bestimmte Softwarekomponenten, beispielsweise aus Public-Key-Verfahren, wiederverwendet werden können und auch über die Wahl der Größe der verwendeten Moduln einfach gesteuert werden können. Diese letzte Eigenschaft ist sehr günstig im Vergleich mit anderen Konstruktionen wie z.B. MD4, bei denen man Änderungen vornehmen muss, um einen längeren Hashwert zu erzeugen.
Die Sicherheit modularer Hashfunktionen hängt von der Schwierigkeit der zahlentheoretischen Probleme wie dem Faktorisierungsproblem und dem diskreten Logarithmus ab. Zur Lösung solcher mathematischen Probleme sind noch keine in Polynomialzeit effizienten durchführbaren Algorithmen bekannt. Es ist aber nicht klar, ob dieses Problem in der Zukunft schwierig bleibt.
Im Bezug auf die Performance sind solche Hashfunktionen im Allgemeinen langsam in der Ausführbarkeit, da die modulare Rechenoperationen mehr Zeit im Vergleich mit elementaren Bitoperationen wie AND, OR, XOR benötigen. Neben der längeren Rechenzeit braucht man bei modularen Rechnungen viel Speicherplatz. Deswegen ist die Verwendung solcher Funktionen in einer Chipkarte ungünstig. Aus diesen Gründen sind solche Konstruktion in der Vergangenheit und heutzutage nicht weit verbreitet.

3.1 Mathematische Grundlagen der Zahlentheorie

In diesem Abschnitt werden die wichtigen Begriffe und Grundlagen über das Thema modulare Arithmetik vorgestellt.
Bei modularer Rechnung beschäftigt man sich nicht mit allen natürlichen Zahlen, sondern nur mit denen unterhalb einer gewissen Grenze N. Oft treten bei Berechnungen Zahlen auf, die größer als N sind; dann muss man diese *modular reduzieren*. Hier soll auf die uns bekannte Arithmetik der

ganzen Zahlen verzichtet und statt dessen eine *modulare Arithmetik* verwendet werden, welche viel stärkere Eigenschaften als die Ganzzahlarithmetik besitzt. Wenn z.B. der Modul eine Primzahl ist, so kann in dieser Arithmetik durch jede beliebige von Null verschiedene Zahl dividiert werden.

Definition 3.1
Sind a, b und q aus $\mathbb{Z}$ mit $a = q \cdot b$, so sagt man, dass b die Zahl a *teilt* und schreibt hierfür $b|a$. Die Zahl b wird dann auch *ein Teiler* von a genannt.

Definition 3.2
Sind a und $b \in \mathbb{Z}$ mit $a \neq 0$ und $b \neq 0$, so gibt es offensichtlich eine größte ganze Zahl, die sowohl a als auch b teilt. Diese wird der *größte gemeinsame Teiler* von a und b genannt und mit $ggT(a,b)$ bezeichnet.
Wenn $ggT(a,b)= 1$ ist, so heißen a und b *teilfremd*.

Definition 3.3
Man definiert den Operator mod so folgendes:

$$a \bmod b := r \Leftrightarrow r \text{ die kleinste natürliche Zahl mit } b|a - r.$$

Wir betrachten die Struktur der natürlichen Zahlen, die kleiner als N sind:

$$\mathbb{Z}_N = \{0, \ldots, N-1\}.$$

In dieser Menge können wir Addieren, Subtrahieren und Multiplizieren:
Seien $a, b \in \mathbb{Z}_N$. Dann ist die Summe $a \oplus b$, die Differenz $a \ominus b$ und das Produkt $a \otimes b$ wie folgt definiert:

$$\begin{aligned} a \oplus b &:= (a+b) \bmod N \\ a \ominus b &:= (a-b) \bmod N \\ a \otimes b &:= (a \cdot b) \bmod N \end{aligned}$$

Man kann zeigen dass $(\mathbb{Z}_N, \oplus)$ eine Gruppe ist, jedoch ist die Struktur $\mathbb{Z}_N$ mit der Multiplikation $\otimes$ nicht in der Regel eine Gruppe. Zum Beispiel ist die Menge $\mathbb{Z}_{12}$ keine Gruppe bezüglich „$\otimes$“, da 2 keine multiplikativ Inverse hat.
Aber die Teilmenge derjenigen Elemente aus $\mathbb{Z}_N$, die teilerfremd zu N sind, bildet eine multiplikative Gruppe. Diese Menge wird mit $\mathbb{Z}_N^*$ bezeichnet und definiert folgendermaßen:

$$\mathbb{Z}_N^* = \{a \in \mathbb{Z}_N \quad ggT(\text{a,b}) = 1 \ \}$$

Die Anzahl der Elemente von $\mathbb{Z}_N^*$ wird mit $\phi(N)$ (*Eulersche Phi-Funktion*) bezeichnet.
Man kann $\phi(N)$ mittels einer Formel berechnen, wenn man die Faktorisierung von N kennt. Es gilt zum Beispiel:

$$\begin{aligned} \phi(p) &= p - 1. \\ \phi(p.q) &= (p-1)(q-1). \end{aligned}$$

Die Zahl $\phi(N)$ ist für $N = p.q$ genauso schwer zu berechnen wie die Faktorisierung von N.
Für die Anwendung der Gruppe $(\mathbb{Z}_N^*, \otimes)$ in der Kryptographie, insbesondere im RSA-Verfahren, ist der folgende *Satz von Euler-Fermat* wichtig:

$$\forall a \in \mathbb{Z}_N^* \text{ gilt: } a^{\phi(N)} = 1 \mod N$$

3.1.1 Faktorisierungsproblem

Es gibt Algorithmen, mit denen man schnell überprüfen kann, ob eine natürliche Zahl eine Primzahl ist oder nicht, z.B. kann man mit Hilfe des Satzes von Euler-Fermat, relativ schnell nachweisen, dass eine Zahl N keine Primzahl ist, ohne N faktorisieren zu müssen.
Dazu muss man den Satz für den Speziallfall formulieren, dass N eine Primazahl ist.
Dieser Speziellfall ist als der *kleine Fermatsche Satz* bekannt und lautet:
Für jede Primzahl p und jede natürliche a mit $1 \leq a \leq p-1$ gilt:

$$a^{p-1} = 1 \mod p$$

Eine natürliche Zahl N (von der man weiß, dass sie keine Primzahl ist) in ihre Primfaktoren zu zerlegen, ist ein sehr schwieriges Problem. Der einfachste Faktorisierungsalgorithmus besteht darin, alle möglichen Teiler von n durchzuprobieren. Im Extremfall muss man dies für alle Zahlen von 1 bis $\sqrt{N}$ tun, was für große Zahlen (über 150 Dezimalstellen oder ca.500 Bits) unmöglich ist.
Die heute in der Praxis zur Faktorisierung großen Zahlen verwendeten Algorithmen sind der *Quadratic-Sieve*-Algorithmus, der *Elliptic-Curve*-Algorithmus und der *Number-Field-Sieve-Algorithmus.*
Zur Faktorisierung einer 512 Bit lange Zahl benötigt der Quadratic Sieve Algorithmus auf einer 200 MIPS Workstations[1]etwa 11700 Jahre.

Die Quadratische Reste
Sei N eine natürliche Zahl. Das Element $a \in \mathbb{Z}_{\mathbb{N}}{}^*$ heißt *quadratischer Rest modulo N*, falls es ein $b \in \mathbb{Z}_{\mathbb{N}}{}^*$ gibt mit:

$$b^2 = a \bmod N.$$

Man nennt dann b eine *Quadratwurzel von a modulo N*. Die Menge aller Quadratwurzel $\mod N$ wird als QR_N bezeichnet. Wenn a kein quadratischer Rest ist, heißt *quadratischer Nichtrest modulo N*.
Die Schwierigkeit der Quadratwurzelberechnung hängt enscheidend von dem Modul N ab : Wenn N eine Primzahl ist, so gibt es schnelle Verfahren, um Quadratwurzeln modulo N zu berechnen. Wenn N eine zusammengesetzte Zahl ist, so ist es im Allgemeinen sehr schwer, Quadratwurzeln zu finden, da dieses Problem mit der Faktorisierung zusammenhängt.
Im Allgemeinen ist es nicht nur schwierig, Quadratwurzeln modulo N zu berechnen, sondern auch zu entscheiden, ob eine Zahl überhaupt eine quadratischer Rest ist oder nicht. Dazu benötigt man zwei wichtige Begriffe.

- Das *Legendresche Restsymbol* für eine ganze Zahl $a \in \mathbb{Z}_{\mathbb{N}}{}^*$ und eine Primzahl $p > 2$ ist wie folgt definiert:

[1] 200.000.000 Befehle pro Sekunde

$$\left(\frac{a}{p}\right) = \begin{cases} 0, & \text{wenn } a \text{ durch } p \text{ teilbar ist.} \\ 1, & \text{wenn } a \text{ ein quadratischer Rest modulo } p \text{ ist.} \\ -1, & \text{wenn } a \text{ ein quadratischer Nichtrest modulo } p \text{ ist.} \end{cases}$$

- *Das Jacobi-Symbol* ist eine Verallgemeinerung des Legendreschen Restsymbols. Man betrachte eine ganze Zahl a und eine ungerade Zahl $N > 2$. Weiterhin sei $N = p_1^{i_1} \cdots p_k^{i_k}$ die Primfaktorzerlegung von N. Dann ist das Jacobi-Symbol definiert als das Produkt der zugehörigen Legendreschen Restsymbole:

$$\left(\frac{a}{N}\right) = \left(\frac{a}{p_1}\right)^{i_1} \cdot \left(\frac{a}{p_2}\right)^{i_2} \cdots \left(\frac{a}{p_k}\right)^{i_k}$$

Wesentliche Vereinfachungen bei der Berechnung des Jacobi-Symbols liefert *das Quadratisches Reziprozitätsgesetz* für das Jacobi-Symbol:
Es seien M und N zwei positive ungerade Zahlen größe als 2. Dann gilt:

$$\left(\frac{M}{N}\right) = (-1)^{\frac{(M-1)(N-1)}{4}} \left(\frac{N}{M}\right)$$

3.1.2 Das diskrete Logarithmus-Problem

Wichtige Grundbausteine der Kryptographie sind Funktionen, die leicht zu berechnen, aber schwer umzukehren sind. Eine wichtige Klasse solcher Funktionen sind die diskreten Exponentialfunktionen.
Sei p eine Primzahl und g eine natürliche Zahl mit $g \leq p - 1$. Dann ist die diskrete Exponentialfunktion zur Basis g definiert durch :

$$k \longmapsto g^k \mod p\,, \quad 1 \leq k \leq p - 1.$$

Die Umkehrfunktion wird diskrete Logarithmusfunktion genannt und mit log_g bezeichnet. Es gilt:

$$log_g(g^k) = k.$$

Zur Berechung des diskreten Logarithmus kann man *Baby-Step-Giant-Step*-Algorithmus oder *Silver-Pohlig-Helmann*-Algorithmus anwenden [5].

3.2 Abstrakte Methoden zur Konstruktion modularer Hashfunktionen

Ein wichtiger Bestandteil von Hashfunktionen ist die blockweise Verarbeitung einer Nachricht M beliebiger Länge. Um den Hashwert von M zu erzeugen, wird das sogenannte *Merkle-Damgård-Prinzip* verwendet. Die Nachricht wird in L-Blöcke M_i fest vorgegebener Länge aufgeteilt und mit Hilfe einer Kompressionsfunktion f wird iterativ eine Folge von Zwichen-Hashwerten berechnet. Das Schema sieht wie folgt aus:

$$H_i = f(X_i, H_{i-1}), \text{ für } i = 1, 2, \ldots, L.$$

Mit H_0 einem vorgegebenen Startwert ist.
Der letzte Wert ist dann der Hashwert der Nachricht M.
Für eine schematische Darstellung siehe die Abbildung (3.1).

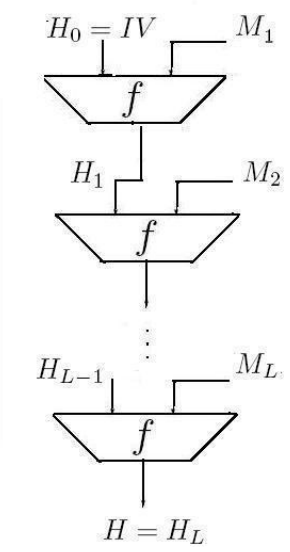

Abbildung 3.1: Merkle-Damgård-Prinzip

Wie schon erwähnt, beruht die Grundidee zur Konstruktion von Hashfunktionen modularer Arithmetik darauf, dass ihrer Sicherheit auf der Schwierigkeit zahlentheoritischer Probleme wie das Faktorisierungs- oder das diskrete Logarithmus Problem basiert und daher ergeben sich zwei Möglichkeiten, eine Hashfunktion zu realisieren.

1. Die erste Konstruktion basiert auf dem Faktorisierungsproblem, d.h. auf die Schwierigkeit der Invertierung. In diesem Fall geht man zur Erstellung einer Kompressionsfunktion wie folgt vor:
   Gegeben sind:

   - L_H die gewünschte Bitlänge des Hashwertes.
   - N eine natürliche Zahl (N kann entweder Primzahl oder zusammengesetzte Zahl[2] sein) mit Bitlänge $L_N = L_H$.
   - M eine Nachricht, welche in Blöcke M_i der Bitlänge L_H aufgeteilt wird.

   Nach der Nachrichteneinteilung wird den sogenannte *Paddingsvorgang* durchgeführt, indem der letzte Block mit bestimmten Bits aufgefüllt wird, bis dieser ein Vielfaches von L_H wird, damit die gesamte Bitlänge der zu hashenden Nachricht auch ein Vielfaches von L_H wird.
   Man definiert die Funktion :

$$\begin{aligned} g\colon \quad \{0,1\}^{L_H} \times \{0,1\}^{L_H} &\longrightarrow \{0,1\}^{L_H} \\ (M_i, H_{i-1}) &\longmapsto g(M_i, H_{i-1}) \end{aligned}$$

   Wobei g eine Verknüpfungsfunktion ist (z.B. XOR), M_i der i-te Block der Nachricht und H_{i-1} der Zwischen-Hashwert.
   In diesem Fall wird die obigen Funktion g als Basis genommen, um die Kompressionsfunktion f zu konstruieren.

[2] Produkt von mindestens zwei Primzahlen

f sieht so folgendes aus:

$$\begin{aligned} f: \quad \{0,1\}^{L_H} \times \{0,1\}^{L_H} &\longrightarrow \{0,1\}^{L_H} \\ (M_i, H_{i-1}) &\longmapsto g(M_i, H_{i-1})^e \mod N \end{aligned}$$

Wobei e eine natürliche Zahl ist.

2. Im zweiten Fall ist die Konstruktion von Hashalgorithmen auf dem diskreten Logarithmus basiert. Dabei werden ein festes Element a als Basis und eine Verknüpfungsfunktion u als Exponent ausgewählt.
   Die Kompressionsfunktion f in diesem Fall sieht wie folgt aus:

$$\begin{aligned} f: \quad \{0,1\}^{L_H} \times \{0,1\}^{L_H} &\longrightarrow \{0,1\}^{L_H} \\ f(M_i, H_{i-1}) &\longmapsto a^{u(M_i, H_{i-1})} \mod N, \end{aligned}$$

   wobei a eine Konstante der Bitlänge L_H und u ist eine Verknüpfungsfunktion ist, deren Bildbereich Bitstrings der gewünschten Bitlänge des Exponentes ist.

Da die Einteilung der Nachricht sich nach der gewünschten Bitlänge des Hashwertes orientiert, bevorzugt man die Konstruktion der Kompressionsfunktion nach dem ersten Schema. Die Verknüpfungsfunktion als Basis kann man einfach mit dem gewünschten Hashwertlänge anpassen. Im zweiten Fall ist dagegen die Einteilung der Nachricht total unabhängig von dem Bildbereich von u, was der allgemeinen Struktur eines Hashalgorithmus nicht entspricht.

Wahl der Parameter N und e

Für eine gute Sicherheit und Performance eines Hashalgorithmus sollen die Parameter N und e bei der Kompressionsfunktion einige Anforderungen erfüllen, welche im Folgenden vorgestellt werden:

- **Modul N**
  - Die Einteilung der Blöcke hängt von der Modullänge ab und deswegen soll die Länge von N nicht so klein gewählt werden, ansonsten hat man mehr kürzere Blöcke zu bearbeiten und daher wird einen großer Rechenaufwand benötigt.
    Für kleine Moduln wird auch die Erfolgswahrscheinlichkeit der Kollisionsangriffe gegen die zwei obigen Konstruktionen steigen.
    Der Modul soll aber auch nicht zu groß sein, wäre dies nicht der Fall, so ist ein großer Speicherbedarf erforderlich und findet die Modulo-Reduktion mit einer hohen Wahrscheinlichkeit nicht statt.
  - Da die mathematische Beziehung $x = y \mod 2^m$ dazu führt, dass die unteren m-Bits von x und y übereinstimmen, ist es empfohlen, einen zweierpotenzer Modul nicht zu benutzen. Dies kann von einem Angreifer ausgenutzt werden.
  - Wird den Hashwert mit dem RSA-Vefahren signiert, so ist es vorteilhaft, der RSA-Modul auch als Hash-Modul zu wählen. In diesem Fall wird den Speicherplatz gespart, wie zum Beispiel bei Chipkartenanwendungen.
  - Man kann auch L_N größer als L_H wählen. Nach dem Potenzieren werden aber die oberen $L_N - L_H$ Bits einfach weggeschnitten.

- **Exponent** e
  - Der Exponent soll nicht gerade sein, um die triviale kollision $x^e = (-x)^e \mod N$ zu vermeiden. Exponenten der Form $2^k + 1$ sind geeignet, da die Binärdarstellung solcher Zahlen nur zwei Einsen enthält und dies wirkt sich positiv auf die Laufzeit der Berechnung bei Anwendungen der schnellen Exponentiation aus.
  - Für eine günstige Performance wählt man in der Regel nicht so große Exponenten, ansonsten braucht man für die Durchführung der Hashfunktion enorm hohe Rechenzeiten, was allerdings zu fast dem gleichen Aufwand wie bei direkter Signierung eines Dokuments führt und deshalb ist die Verwendung von Hashfunktionen in diesem Fall sinnlos.

3.3 Allgemeine Konstruktionen modularer Hashfunktionen

Die Familie von Hashfunktionen kann man in zwei Gruppen unterteilen:
Die erste Gruppe MDC (*Modification detection codes*) besteht aus *parameterlosen* Hashfunktionen (*ohne geheimen Schlüssel*), welche am meisten für die Überprüfung von Datenintegrität eingesetzt werden.
Die zweite Gruppe MACs (*Message Authentication codes*) besteht aus *parametrisierten* Hashfunktionen (*mit einen geheimen Schlüssel*), welche noch dazu die Authentizität sichern können.
Im Folgenden werden zunächst ein paar Schemata von Konstruktionen der ersten Gruppe (MDCs) vorgestellt und anschließend wird es einen kurzen Überblick gegeben, welcher zeigt, wie die MACs mittels MDCs aufgebaut werden können.

3.3.1 Konstruktion der Hashfunktionen ohne Schlüssel(MDC)

Wie schon erwähnt, hat die Größe des Moduls N eine bedeutsame Auswirkung sowohl auf die Sicherheit als auch auf die Performance der Konstruktion von Hashfunktionen. Je größer der Modul gewählt wird, desto schwieriger werden die Angriffe und länger ist die Berechnungszeit.
Man kann also diese Schemata hauptsächlich in zwei Hauptkategorien klassifizieren, nähmlich in Konstruktionen entweder mit kleinen Moduln oder mit großen Moduln.

3.3.1.1 Modulare Hashfunktionen mit kleinen Moduln

Der erste Versuch eine modulare Hashfunktion zu konstruieren geht auf R.Jueneman und C. Meyer zurück. Ihre Grundidee war, eine Hashfunktion unter Verwendung von einfacher Genauigkeit auf einen 32-Bit-Koprozessor zu realisieren, die schneller als das auf das DES basierende Hashfunktion sein sollte. Diese Hashfunktion wurde als *quadratic congruential manipulation detection code (QCMDC)* oder als *Juenemans Verfahren* bezeichnet [32],[33], dessen Beschreibung im Folgenden erklärt wird.
Zunächst wird eine Mersenne-Primzahl N betrachtet, d.h. $N = 2^{31} - 1$. Die Nachricht M wird danach in L-Blöcke M_i mit je 32-Bit geteilt und blockweise bearbeitet. Gestartet mit einem Initialwert $H_0 = IV = 0$ wird jeder Block M_i mit dem vorherigen Zwischen-Hashwert H_{i-1} erst einmal mit einer XOR-Operation verknüpft und danach quadriert. Das daraus sich ergebende Resultat wird dann durch die modulare Operation reduziert, um den Zwischen-Hashwert H_i zu berechnen (Siehe Abbildung 3.4):

$$H_i = (M_i \oplus H_{i-1})^2 \bmod N.$$

Der endgültige Hashwert ist $H = H_L$.

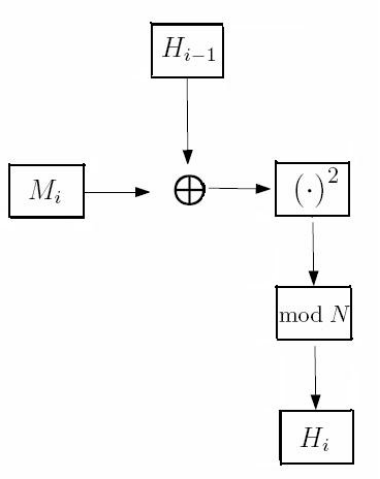

Abbildung 3.2: Quadratic congruential manipulation detection code (QCMDC)

Als Angriff gegen dieses Schema kann ein Correcting Block Attack (Kapitel 2, Abs. 2.4) durchgeführt werden, dessen Funktionsweise im Folgenden erklärt wird. Für eine beliebige Nachricht M mit L-Blöcke, deren Hashwert mit H_L bezeichnet ist. Der Angreifer kann eine andere Nachricht M' mit L'-Blöcke mit je 32-Bit bilden, deren Hashwert mit H_L übereinstimmt. Das geschieht wie folgt:
Für $1 \leq i \leq L' - 1$ wird zunächst

$$M'_i = S_i,$$

genommen, wobei S_i für ein beliebiges 32-Bitstring steht. Der letzte Block $M'_{L'}$ wird dann wie folgt gleichgesetzt:

$$M'_{L'} = M_L \oplus H_{L-1} \oplus H'_{L'-1}.$$

Hierbei $H'_{L'-1}$ der dazugehörige Zwischen-Hashwert des Nachrichtenblocks $M_{L'-1}$, welcher mit QCMD berechnet wurde. Daraus folgt

$$M'_{L'} \oplus H'_{L'-1} = M_L \oplus H_{L-1}$$

Schließlich gilt $H'_{L'} = H_L$ und daher erzeugen M und M' eine Kollision.

Da dieses Schema einen Hashwert der Länge 32 Bit produziert, braucht man durch Birthday Attack lediglich nur 2^{16} Operationen zum Finden einer Kollision. Damit ergab sich die Notwendigkeit, dieses Schema zu entwickeln.
In [34] wurde eine neue Version veröffentlicht. Dabei wird die Nachricht vier Mal durch das QCMD verarbeitet, um einen 128 Bit-Hashwert zu erzeugen. In jeder Verarbeitung wird das Ergebnis der

vorherigen Ausführung als Initialwert verwendet. Der endgültige Hashwert wird durch die Aneinanderkettung von den Hashwerten bestimmt, welche sich aus den vier Ausführungen ergeben.
Kurz danach zeigte D. Coppersmith, dass das Auffinden einer Kollision mithilfe vom Man-In-The-Middle-Angriff.[3] nur 2^{45} Operationen verlangt. Dieser hat sich Gedanken darüber gemacht, eine dritte Version zu entwickeln. In dieser neuen Version nimmt man die Blockeinteilung der Nachricht und die Zwischen-Hashwerte etwas anderes vor :
Der Nachrichtblock M_i bzw. der Zwischen-Hashwert H_i wird in vier 32-Bit-Blöcke eingeteilt, welcher mit M_{ij} bzw. H_{ij} $(0 \leq j \leq 3$) bezeichnet werden, danach wird der nächste Zwischen-Hashwert H_{i+1} folgendermaßen berechnet [2]:

In der i-ten Iteration :

$$f_{ij} = [(H_{ij \mod 4} \oplus M_{i0}) - (H_{ij+1 \mod 4} \oplus M_{i1}) + (H_{ij+2 \mod 4} \oplus M_{i2}) - (H_{ij+3 \mod 4} \oplus M_{i3})]^2 \mod N \qquad \text{für } 0 \leq j \leq 3$$

Somit ist : $H_{i+1} = f_{i0} \| f_{i1} \| f_{i2} \| f_{i3}$
Der erzeugende Hashwert wird: $H_L = f_{L0} \| f_{L1} \| f_{L2} \| f_{L3}$

Die verbesserte Version, als *QCMDCV4* bezeichnet, bei welcher in jeder i-ten Iteration vier unterschiedliche Moduln benutzt werden, welche wie folgt festgelegt sind:

$$\begin{aligned} N_0 &= 2^{31} - 19, \\ N_1 &= 2^{31} - 61, \\ N_2 &= 2^{31} - 69, \\ N_3 &= 2^{31} - 85. \end{aligned}$$

Außerdem wird dazu ein zusätzlicher Block M_{i4} konstruiert, welcher von vier vorhergehenden Blöcken abhängt. Dies funktioniert wie folgt:

$$M_{i4} = (00 \| M_{i0}[31-26] \| M_{i1}[31-24] \| M_{i2}[31-24] \| M_{i3}[31-24]),$$

wobei $X[l-k]$ die von l bis zum k liegende Bits einer Nachricht X darstellt.
Die Berechnung des Zwischen-Hashwerts H_{i+1} erfolgt dann durch die Konkatenation der Ergebnisse f_{ij} in $i-ten$ Iteration, welche durch die folgende Gleichung gegeben sind:

$$f_{ij} = [(H_{ij \mod 4} \oplus M_{i0}) - (H_{ij+1 \mod 4} \oplus M_{i1}) + (H_{ij+2 \mod 4} \oplus M_{i2}) - (H_{ij+3 \mod 4} \oplus M_{i3}) + (-1)^j M_{i4}]^2 \mod N_j,\ 0 \leq j \leq 3.$$

Ein Jahr später zeigte D.Coppersmith, dass diese Konstruktion mithilfe des Birthday Attack zusammen mit Correcting-Block-Angriff in etwa 2^{18} Operationen gebrochen werden kann.
Als Zusammenfassung kann man feststellen, dass die Wahl von kleinen Moduln (≤ 128 Bits) nicht geeignet sind, sichere modulare Hashfunktionen zu konstruieren.

3.3.1.2 Modulare Hashfunktionen mit großen Moduln

Der Modul bei solchen modularen Hashfunktionen ist am häufigsten ein RSA-Modul. Dieser ist im Allgemeinen ein Produkt zweier großer Primzahlen p und q, die bestimmten Anforderungen erfüllen

[3] Ein man-In-The-Middel-Angriff ist eine Angriffsform, wobei der Angreifer die Kontrollle über den Datenverkher zwischen zwei Kommunikationspartner hat.

müssten [22]. Andere Hashfunktionen verwenden eine Gruppe $\mathbb{Z}_p^*$, wobei p eine ausreichend große Primzahl ist, die aus Sicherheitsgründen wenigstens 1024 Bit lang sein muss.
Bevor diese Kategorie von Hashfunktionen beschrieben wird, soll betont werden, dass die Verwendung desselben Moduln für die Hashfunktion und das anschließende Signatur-RSA-Verfahren zwei verschiedene Sicherheitsprobleme verursacht:

• Das Problem liegt an dem Benutzer des Signaturverfahrens selbst, denn der Besitzer des geheimen RSA-Schlüssel kennt die Faktorisierung des RSA-Moduls und besitzt somit eine Information, wie die Hashfunktion invertiert werden kann.
Um dieses Problem zu beseitigen, besteht die Möglichkeit, gemeinsamen Modul zu verwenden, die von einem vertrauenswürdigen Dritte (z.B. Zertifizierungsstellen) erzeugt wird. Jedoch stellt das keinen optimalen Ausweg dar, wenn alle möglichen Angriffe auf diese Stelle konzentriert werden.
Die zweite Lösung besteht darin, zwei verschiedene Hashwerte mit dem Modul des Besitzer und des Signers zu berechnen. Diese Hashwerte werden konkateniert und signiert. So ergibt sich am Ende ein gültiger Hashwert. Dies wird zwar bestimmt das Vertrauensproblem beheben, aber gleichzeitig wird die Performance des Schemas schlechter.

• Das zweite Problem betrifft besonders die Unempfindlichkeit des Signaturverfahrens, wenn es eine multiplikative Beziehung von Hashwerten als Ganzzahlen überträgt. Mehr Details dazu werden am Beispiel der Konstruktion der MASH-Hashfunktionen ausführlich erklärt.

Grundsätzlich unterteilen sich die Konstruktionen von Hashfunktionen mit großem Modul in zwei Klassen: Zum einen gibt es die nicht-beweisbar Kollisionsresistenz-Hashfunktionen mit großem Modul und zum anderen gibt es die beweisbar Kollisionsresistenz Hashfunktion.

a - Nicht-beweisbar Kollisionsresistenz-Hashfunktionen mit großem Modul: In diesem Abschnitt werden einige Methoden für die Konstruktion von Hashfunktionen erläutert, die nicht Kollisionsresistenz sind. Solche Hashfunktionen sind langsamer als die anderen bekannten Hashfunktionen (wie z.B. SHA-1). Sie bedienen sich des iterativen Ansatzes auf der Basis der Quadrierungsmethoden, verknüpft mit anderen modularen Rechenoperationen wie XOR oder modularer Addition. Die Nachricht, für die der Hashwert zu berechnen ist, wird in mehrere Blöcke M_i geteilt.
Das allgemeine Schema, bei dem die Hashwertlänge und Blockgröße gleich sind, sieht wie folgt aus:

$$\begin{aligned} H_0 &= IV, \\ H_i &= (A)^2 \bmod N \oplus B, \end{aligned}$$

wobei A und $B \in \{IV, M_i, H_{i-1}, (M_i \oplus H_{i-1})\}$ sind.
Daraus ergeben sich insgesamt 16 Möglichkeiten, die in der Tabelle (3.1) dargestellt werden, welche verdeutlicht, welche Schemata gegen fünf verschiedene Angriffe anfällig sind, die im Kapitel 2 (Abschnitt 2.5) bereits erklärt wurden. Mit „−− „ gekennzeichnete Konstruktionen entsprechen den schwachen Konstruktionen, welche durch die fünf Angriffe geknackt werden können. Für mehr Details siehe [23], [26], [36], [13].

No.	Schema	Angriffe
1	$(M_i)^2 \bmod N \oplus M_i$	$--$
2	$(H_{i-1})^2 \bmod N \oplus M_i$	DA
3	$(M_i \oplus H_{i-1})^2 \bmod N \oplus M_i$	FPA
4	$(IV)^2 \bmod N \oplus M_i$	$--$
5	$(M_i)^2 \bmod N \oplus H_{i-1}$	PA
6	$(H_{i-1})^2 \bmod N \oplus H_{i-1}$	$--$
7	$(M_i \oplus H_{i-1})^2 \bmod N \oplus H_{i-1}$	FPA
8	$(IV)^2 \bmod N \oplus H_{i-1}$	$--$
9	$(M_i)^2 \bmod N \oplus (M_i \oplus H_{i-1})$	PA
10	$(H_{i-1})^2 \bmod N \oplus (X_i \oplus H_{i-1})$	DA
11	$(M_i \oplus H_{i-1})^2 \bmod N \oplus (M_i \oplus H_{i-1})$	FA
12	$(IV)^2 \bmod N \oplus (M_i \oplus H_{i-1})$	DA
13	$(M_i)^2 \bmod N \oplus IV$	$--$
14	$(H_{i-1})^2 \bmod N \oplus IV$	$--$
15	$(M_i \oplus H_{i-1})^2 \bmod N \oplus IV$	FA
16	$(IV)^2 \bmod N \oplus IV$	$--$

Tabelle 3.1: Die 16 möglichen Squaring-Methoden

Um die Sicherheit der obigen Schemata zu erhöhen, wurde von Davies und Price in [16] vorgeschlagen, die Quadrierung durch eine Potenzierung mit Exponent $e > 2$ zu ersetezen. Daher sieht die Kompressionfunktion folgendermaßen aus:

$$f(M_i, H_{i-1}) = (M_i \oplus H_i)^e \bmod N$$

Hierbei stellt e den öffentlichen Schlüssel von RSA-Kryptosystemen dar.
Ist die Faktorisierung des Moduls bekannt, so gilt dieses Schema als gebrochen, da die Kompressionsfunktion in diesem Fall einfach zu invertieren ist.
Zur Verbesserung der Sicherheit dieses schemas kann man nach dem Potenzieren auf der obigen Formel weitere elementare Bitoperationen einfügen, z.B XOR operation u.s.w, d.h auch bei bekannter Faktorisierung des Moduls N ist nicht einfach die folgende Gleichung $(X^e \bmod N) \oplus X$ nach X aufzulösen.

b - Beweisbar Kollisionsresistenz Hashfunktionen mit großem Modul: Die Beweisbar Kollisionsresistenz Hashfunktionen mit großem Modul stellen eine Klasse von Funktionen, deren Sicherheitsbeweis auf der Schwierigkeit der Lösung von bestimmten harten mathematischen Problemen basiert.
Der erste Vorschlag geht auf I.Damgård zurück [21]. Seine Idee war, kryptographische Hashfunktionen auf der Basis von *kollisionsresistenzen Permutationen* aufzubauen.

Defintion 3.4

Sei $\Sigma = \{0, 1, \cdots, s-1\}$ ein Alphabet der Kardinalität s, mit $s \in \mathbb{N}$.
Eine Menge von Permutationen $\mathcal{M} := \{f_i : 0 \leq i < s, s \in \mathbb{N}\}$ gilt als Kollisionsresistenz, wenn :

1. alle f_i aus $\mathcal{M}$ den gleichen Definitionsbereich D besitzen.
2. $f_i(x)$ einfach zu berechnen ist, für alle $x \in D$ und $f_i \in \mathcal{M}$
3. praktisch unmöglich ist, zwei unterschiedliche Elemente x, y zu finden, so dass $f_i(x) = f_j(y)$ mit $0 \leq i, j < s$.

Defintion 3.5
Sei A ein Nachrichtenraum. Eine Abbildung präf : $A \longrightarrow A$ heißt präfixfrei genau dann, wenn für zwei beliebige unterschiedliche Elemente x, y aus A, kein Wort $z \in \Sigma^*$ existiert, so dass die folgende Gleichung präf$(x)z =$ präf (y) gilt.

Auf Basis der Menge von Permutationen wird der folgende *Damgård-Hashfunktion*-Algorithmus dargestellt, wodurch der Hashwert H einer Nachricht M erzeugt wird.

Eingabe:

- M eine Nachricht aus einem Bildraum A.
- präf : $A \longrightarrow A$ präfixfrei, präf $(M) := M_1 M_2 \cdots M_r$ mit $M_i \in \Sigma$ und $f_i \in \mathcal{M}$ für alle i.
- R ein zufällig gewähltes Element aus D.

Ausgabe:

- Hashwert H.

Verlauf:

- $H_0 = 0$.
- $h(M) := F_{(M)}(R) := f_{M_1}(f_{M_2}(\cdots f_{M_r}(R) \cdots))$.
- Gebe $H = h(M)$.

Satz 3.1
Ist die $\mathcal{M}$ Kollisionsresistenz, so ist Damgård-Hashfunktion Kollisionsresistenz.

Aufbauend auf dieses Prinzip wird ein konkretes Beispiel von Permutationmengen dargestellt.
Sei:

- Ein Modul $N = p \cdot q$ mit $p, q = 3 \mod 4$ und $\log_b(p) \approx log_b(q) \approx k/2$ (p und q haben etwas 350 Bits als Bitlänge).
- Ein Element $a = a_1, a_2, \cdots a_{s-1}$, wobei $a_i \in QR_N$ mit $0 \leq i < s$ (Die Definition von QR_N findet man im Kapitel 3, Abschnitt (3.1)).

Definiere:
$\mathcal{M} = \{f_i : f_i(x) = a_i.x^2 \bmod N, \text{def}(f_i) = QR_N, 0 \leq i < s\}$
In [3] ist gezeigt, dass die soeben definierte Menge $\mathcal{M}$ Kollisionsresistenz ist, solange N schwer zu faktorisieren ist.
Daraus folgt nach dem obigen Satz, dass die Damgård Hashfunktion Kollisionsresistenz ist, wenn N schwer zu faktorisieren ist.

3.3.2 Hashfunktionen mit Schlüsseln (MACs)

Die MACs (Message Authentication Code) sind Schlüssel-abhängige kryptographische Hashfuntionen. Diese Menge von Hashfunktionen wird folgendermaßen bezeichnet: $\{H_k$, k aus K $\}$, wobei K die Schlüsselmenge und k einen Schlüssel aus k ist.
Der Schlüssel k ist nur den beiden Kommunikationpartnern bekannt, die authentifizierte Dokumente austauschen möchten. Man nennt Authentication Codes deshalb auch Hashfunktionen mit Schlüssel (engl.keyed-one-way-function) und der gemeisame Schlüssel wird als MAC-Geheimnis bezeichnet.
Als zusätzliche Eigenschaft im Vergleich mit normalen Hashfunktionen (MDCs) dienen die MACs auch dazu, Signaturen herzustellen und das funktioniert folgendermaßen:
Seien A und B zwei Partnern, die sich gegenseitig Nachrichten schicken wollen.

1. A und B tauschen einen Schlüssel k aus K aus.
2. A schickt an B $(m, H_k(m))$.
3. B überprüft einfach, ob mit der Hashfunktion zum vereinbarten Schlüssel das gleiche $H_k(m)$ gebildet worden ist.
4. B überprüft damit, dass die Nachricht unverändert ist (das ist das, was eine normale Hashfunktion leisten sollte), aber auch, ob die Nachricht von A kommt.

Die MACs haben den Vorteil, dass sie viel schneller als die anderen Signatur-Verfahren aus public-Key-Systemen sind, aber sie haben den Nachteil dass, die zwei Kommunizierenden müssen sich trauen (A kann abstreiten $(m, H_k(m))$ geschickt zu haben, weil B kann auch $(m, H_k(m))$ erzeugen). Der MAC-Code ist zum Beispiel beim Online-Banking in HBCI eingesetzt und eignet sich besonders für Chipkarten.

3.3.2.1 Konstruktion der Hashfunktionen mit Schlüssel (MACs)

Das Design eines MACs mit Hilfe von modularer Arithmetik zu konstruieren, wurde von F.Cohen und Huang vorgeschlagen [3]. Die Idee besteht darin, dass die modulo Reduktion mit dem RSA-Verfahren kombiniert wird. Diese Kombination kann durch den folgenden Algorithmus dargestellt werden.

MAC-Algorithmus

Eingabe:

- Eine Nachricht M, welche in t-Blöcke $M_1, M_2, \cdots M_t$ aufgeteilt wird.
- (N, e) der RSA-öffentliche Schlüssel, wobei RSA$(X)= X^e \mod N$
- k ein zufällig ausgewählter Geheimschlüssel.

Ausgabe:

- Hashwert H.

Verlauf:

- $H_1 = RSA(M_1 \oplus k)$
  $H_i = RSA(1 + [(M_i \oplus k) \mod (H_{i-1} - 1)) \mod N$ für $2 \leq i \leq t$
- Gebe $H = H_t$.

Solche Konstruktion weist viele Schwachstellen auf. Einige Beispiele davon sind Manipulation der Nachricht von einem Angreifer, indem er die ersten Bits des Geheimschlüssels erkennt. Eine weiterer wichtiger Schwachpunkt besteht in der Möglichkeit des Ausfalls von Modulo Reduktion. Mehr Details über diese Schwachstelle findet man in [3].

3.4 Konkrete Beispiele

Nachdem wir uns mit der allgemeinen Konstruktion von modularen Hashfunktionen beschäftigt haben, wollen wir im diesem Abschnitt auf wichtige Bespiele eingehen, auf welche die Aufmerksamkeit in der Literatur gerichtet wurde. Diese Beispiele können in zwei Gruppen unterteilt werden:

- Die nicht beweisbar sicheren Hashfunktionen, welche aus der Square-Mod-N und deren Erweiterung MASH-Familie besteht.
- Die beweisbar sicheren Hashfunktionen und nämlich die Chaum-van Heijst-Pfitzmann sowie VSH- und ihre Variante VSH-DL-Hashfunktion.

3.5 Square-Mod-N

1987 wurde von Marc Girault die folgende Kompressionfunktion veröffentlicht [27]:
Gegeben eine Nachricht M, welche in Blöcke $M_1, M_2, \cdots, M_n$ der Bitlänge L_H geteilt wird. Sei N eine natürliche Zahl der Bitlänge L_H. Definieren wir die folgende Funktion:

$$f(M_i, H_{i-1}) = (M_i \oplus H_{i-1})^2 \mod N,\ i = 1, \cdots n.,$$

wobei H_0 eine vorgegebene Initialwert ist.

Wie kann man bemerken, wurde dieses Schema bereits im vorherigen Kapitel vorgestellt. Es wurde gezeigt, dass ein Correcting Block Attack auf diese Funktion durchgeführt werden kann, welcher zur Erzeugung von trivialen Kollisionen für diese Hashfunktion führen kann.
Gegen diesen Angriff haben die ISO (International Organization for Standarisation) in 1989 eine neue Version entwickelt, welche Square-Mod-N-Hashfunktion genannt wurde [6].

3.5.1 Square-Mod-N-Algorithmus

Eingabe :

- L_H die Bitlänge des gewünschtes Hashwertes, welche ein Vielfaches von 16 Bits sein soll.
- Eine Nachricht M, welche in n Blöcke M_i der Bitlänge $L_H/2$ aufgeteilt wird.
- Ist die Nachricht keine Vielfache von $L_H/2$, so wird dann der letzte Block mit Einsen aufgefüllt (Padding).

- Jeder Block M_i wird danach zu einem Block B_i expandiert, indem jeweils vier Bits jedes Blocks M_i vier binäre Einsen vorangestellt werden. (Siehe Table (4.1) und (4.2) unten)

Ausgabe :

- L_H-Bit-Hashwert H.

Verlauf :

- $H_0 = 0$.
- Führe die Berechnung des Hashwert durch:
  für $i = 1, \cdots, n$ $\qquad f(M_i, H_{i-1}) = (B_i \oplus H_{i-1})^2 \mod N$.
- Gebe H= $f(M_n, H_{n-1})$ aus.

b_0	b_1	b_2	....	$b_{\frac{L_H}{2}-3}$	$b_{\frac{L_H}{2}-2}$	$b_{\frac{L_H}{2}-1}$

Tabelle 3.2: Binärdarstellung des Blocks M_i.

1	1	1	1	b_0	b_1	....	1	1	1	1	$b_{\frac{L_H}{2}-4}$	$b_{\frac{L_H}{2}-3}$	$b_{\frac{L_H}{2}-2}$	$b_{\frac{L_H}{2}-1}$

Tabelle 3.3: Expandierung des Blocks M_i zu einem Block B_i.

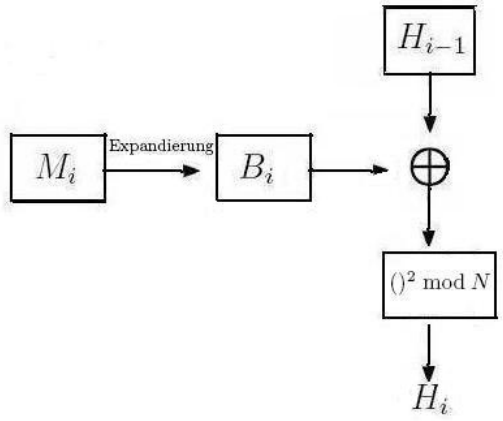

Abbildung 3.3: Funktionsweise der SQUARE-MOD-N Hashfunktion

3.5.2 Abwehr gegen Correcting-Block-Attack

Die Idee einer Expandierung der M_i -Blöcke der Nachricht zu den B_i-Blöcken mit den Redundanz-Bits besteht darin, dass der Angreifer in diesem Umstand den Einfluss nur auf die Hälfte der Bits hat, in diesem Fall hat er nicht die Möglichkeit, aus einen Wert H_i ein B_{i+1} zu erzeugen, so dass $B_i \oplus H_{i-1}$ einen bestimmten Wert ergibt.
Durch diese Expandierung können auch andere Angriffe wie zum Beispiel Direkt Attack und Forward Attack erfolgreich bekämpft werden.

Um die Erfolgswahrscheinlichkeit der generischen Angriffe (Birthday Attack oder Random Attack) zu minimieren, wird empfohlen, in Square-Mod-N-Algorithmus eine Hashwert-Länge größer als 160 Bits zu nehmen.
In der Regel sollte eine gute Hashfunktion möglichst mit jedem denkbaren Signatur-Algorithmus kompatibel sein. Eine Schwachstelle bei Square-Mod-N besteht darin, dass diese Hashfunktion im Zusammenhang mit dem RSA-Verfahren keine Unempfindlichkeit des Signaturverfahrens erfüllt, d.h. aus einer Signatur eines Dokuments kann man die Signatur anderer Dokumente konstruieren. Dieser Angriff wurde *Coppersmith-Attack* genannt, welcher im Folgenden vorgestellt wird.

3.5.3 Grundprinzip des Coppersmith-Attack

Angenommen es wird das RSA-Signaturverfahren mit dem Modul N' verwendet.
Die Idee beim Coppersmith-Attack besteht darin, für zwei beliebige Nachrichten M und M' zwei Bitstrings S und S' der Bitlänge $L_H/2$ zu finden, so dass die folgende Gleichung gilt:

$$h(M'||S') = 256.h(M||S) \tag{4.1}$$

In der Praxis soll das Anhängen von S keine große Änderung am Inhalt der Nachricht M zur Folge haben.
Unter Verwendung von RSA-Signatur auf der Gleichung (4.1) erhält man:

$$RSA(h(M'||S')) = RSA(256).RSA(h(M||S)) \bmod N' \tag{4.2}$$

Mittels der Gleichung (4.2) kann der Wert $RSA(256)$ berechnet werden. Erhält ein Angreifer in diesem Fall eine signierte Nachricht, so kann er danach die Signaturen anderer Nachrichten problemlos erzeugen.

3.5.4 Konstruktion des Coppersmith-Attack

Im Folgenden wird das Szenario vom Coppersmith-Angriff vorgestellt.
Seien M und M' zwei Nachrichten, welche in Blöcke der Bitlänge $L_H/2$ eingeteilt werden. Weiterhin bezeichnen wir mit m bzw. m' die Anzahl der Blöcke der Nachricht M bzw. M'.
Ist die Länge der zwei Nachrichten kein Vielfaches von $L_H/2$, so muss M bzw. M' einfach mit Einsen gepaddet werden.
Man fügt dann den Block S (bzw. S') mit Bitlänge $L_H/2$ zu der Nachricht M (bzw M') ein, d.h die Anzahl der Blöcke von den neuen Nachrichten $(M||S)$ (bzw. $M'||S'$) besteht aus $m+1$ (bzw $m'+1$) Blöcke.
Nach dieser Vorbereitungsphase wird der Square-Mod-N-Algorithmus auf $(M||S)$ und $(M'||S')$ ausgeführt.
Damit die Gleichung (4.1) erfüllt wird, muss die folgende Beziehung gelten:

$$(H'_{m'} \oplus B'_{m'+1}) = 16 \cdot (H_m \oplus B_{m+1}) \tag{4.3}$$

Die Gleichung (4.3) bedeutet, dass $H'_{m'} \oplus B'_{m'+1}$ um 4 Bits nach recht bezüglich $H_m \oplus B_{m+1}$ verschoben sind.
Wegen der Redundanzstruktur der Blöcke besteht die Angriffsidee darin, dass die manipulierbaren

Bits von $H_m \oplus B_{m+1}$ zur Deckung mit den nicht-manipulierbaren-Bits von $H'_{m'} \oplus B'_{m'+1}$ gebracht werden.
Man bezeichnet mit $B'_{m'+1}[i], B_{m+1}[i], H'_{m'}[i]$ und $H_m[i]$ für $i = 0, \cdots, L_H/4$ die Halbytes der vier Blöcke B_{m+1}, $B'_{m'+1}$,H_m und $H'_{m'}$.
In diesem Fall gilt die Gleichung (4.1) genau dann, wenn die folgenden Gleichungen erfüllt sind:

$$H'_{m'}[0] \oplus B'_{m'+1}[0] = 0 \tag{4.4}$$

$$H'_{m'}[i] \oplus B'_{m'+1}[i] = H_m[i-1] \oplus B_{m+1}[i-1] \quad \text{für } i = 1, \cdots, L_H/4 - 1 \tag{4.5}$$

$$H_m[L_H/4 - 1] \oplus B_{m+1}[L_H/4 - 1] = 0 \tag{4.6}$$

Der Coppersmith-Attack wird nur funktionieren, wenn die drei Voraussetzungen (4.4),(4.5) und (4.6) erfüllt sind.

Damit die Gleichung (4.4) gültig ist, reicht es einfach $H'_{m'}[0] = B'_{m'+1}[0]$ auszuwählen.
Nach der Definition von B_{m+1} ist $B_{m+1}[L_H/4 - 1] = (hex)F$ [4], d.h die Gleichung (4.6) gilt nur, wenn die Bedingung $H_m[L_H/4 - 1] = (hex)F$ erfüllt ist. (B_1)
Wegen der Redundanzstruktur der B-Blöcke bzw. B'-Blöcke hat man die Möglichkeit, jedes zweite Halbbyte zu manipulieren, also für ungeraden bzw. geraden Indizes i kann man aus $B_{m+1}[i]$ bzw. $B'_{m'+1}[i-1]$ ein $B'_{m'+1}[i-1]$ (bzw $B_{m+1}[i]$) bestimmen, so dass (4.5) erfüllt ist.

Sind die drei Gleichungen erfüllt, so erhält man aus (4.3) die folgende Beziehung:

$$h(M'||S') = 256 \cdot h(M||S) \mod N \tag{4.7}$$

Wir gehen davon aus, dass $h(M||S) < N/256$ gilt (B_2).
In diesem Fall gilt die Gleichung (4.1) und durch die Anwendung von RSA-Signatur wird das Ziel erreicht, welches durch die Gleichung (4.2) charakterisiert wird.

Zusammenfassung: Zur Realisierung des Coppersmith-Attacks müssen die zwei Bedingungen (B_1) und (B_2) erfüllt sein.

3.5.4.1 Erfolgswahrscheinlichkeit des Coppersmith-Attack

Nach dem letzten Abschnitt hängt die Erfolgswahrscheinlichkeit des Coppersmith-Attack von der Erfüllung der zwei Bedingungen (B_1), (B_2) ab.
(B_1) kommt mit der Wahrscheinlichkeit $\frac{1}{16}$ vor, während (B_2) mit der Wahrscheinlichkeit $\frac{1}{256}$ eintritt.
Da die zwei Ereignisse (B_1) und (B_2) unabhängig voneinander sind, ist die Gesamterfolgswahrscheinlichkeit des Coppersmith-Attack gleich $\frac{1}{16} \cdot \frac{1}{256} = \frac{1}{4096}$.

Die Wahrscheinlichkeit des Angriffs wird noch größer, wenn man $N = N'$ nimmt, d.h wenn RSA-Modul und Square-Mod-N gleich sind. In diesem Fall wird die Bedingung (B_2) überflüßig und

[4] $(hex)F$ hat 1111 als Binärdarstellung

daraus folgt eine Steigerung der Erfolgswahrscheinlichkeit von diesem Angriff um 256.
Durch dieses Beispiel kann man feststellen, dass die Verwendung von Hash-Modul als RSA-Modul zwar ungeeignet ist, aber es ist gleichzeitig nicht praktisch, zwei unterschiedliche Moduln anzuwenden, da ein großer Speicherplatz dazu benötigt wird.

3.5.5 Entwicklung des Square-Mod-N

Zur Bekämpfung des Coppersmith-Attacks ergibt sich die Notwendigkeit, die vorherige Version von Square-Mod-N zu entwickeln [1]. Dabei besteht die Idee darin, dass noch mehrere Bedingungen erzeugt werden, um die Erfolgswahrscheinlichkeit vom Coppersmith-Attack zu reduzieren.

3.5.5.1 Entwicklung des Square-Mod-N-Algorithmus

Die Idee bei dieser neuen Konstruktion basiert auf einer neuen Einteilung der Nachricht M, welche in Blöcke der Bitlänge $(L_H - 1)/16$ Byte statt L_H-Bitlänge Blöcke aufgeteilt wird. Hier wird L_H als ein Vielfaches von 8 gewählt, damit die Hashwert-Länge ein Vielfaches eines Bytes ist.
Die resultierende Blöcke werden dann mit (hex)F genauso wie beim vorherigen Algorithmus expandiert. In diesem Fall haben die Blöcke B_i eine Bitlänge kleiner als L_H. Damit jeder Block B_i mit H_{i-1} der Bitlänge L_H verknüpft wird, muss er mit führenden Nullen aufgefüllt werden. Sei l die Anzahl der Halbytes aller solchen Nullen. Beispielsweise für den letzten Block B_{m+1} muss gelten:

$$B_{m+1}[L_H/4 - i] = 0,\ i = 1, \cdots, l - 1.$$

Nach dieser neuen Änderung gilt die Voraussetzung (4.1), wenn die folgenden Gleichungen erfüllt sind:

$$H_m[L_H/4 - 1] = 0. \tag{4.8}$$
$$H_m[L_H/4 - i - 1] = H'_m[L_H/4 - i],\ i = 1, \cdots, l - 1. \tag{4.9}$$
$$H_m[L_H/4 - l - 1] = H'_m[L_H/4 - l] \tag{4.10}$$

Statt der Bedingung (B_1) bei der ersten Version hat man jetzt $l + 1$ Bedingungen. Somit ergibt sich den Wert $\frac{1}{256} \cdot \frac{1}{16^{l+1}}$ als die neue Erfolgswahrscheinlichkeit des Coppersmith -Attack.

Durch diese Neuformulierung von Square-Mod-N wurde die Erfolgswahrscheinlichkeit des Coppersmith-Attack zwar reduziert, aber nicht total bekämpft. Es war deswegen notwendig, einen neuen sicheren Hashalgorithmus zu entwerfen.
Im März 1994 wurde eine neue Hashfunktion vorgeschlagen, welche MASH-1 (Modular Arithmetik Secure Hash Number 1) genannt wurde [19].

3.6 MASH-1 (Erste Version)

Die Konstruktionsziel von MASH-1 besteht darin, dass man den Coppersmith-Attack abwehren muss, d.h es soll bei dieser neuen Konstruktion beachtet werden, nicht so leicht zwei Nachrichten zu erzeugen, deren Hashwerte ein ganzzahliges Vielfaches voneinander sind.

3.6.1 MASH-1-Algorithmus

Eingabe :

- L_H die Bitlänge des gewünschten Hashwertes.
- Eine Natürliche Zahl N als Modul mit $L_N > L_H$.
- Eine Nachricht M, welche in m Blöcke M_i der Bitlänge $L_H/2$ aufgeteilt wird.
- Hat der letzte Block M_m nicht die Bitlänge $L_H/2$, so wird dann mit Nullen aufgefüllt (Padding).
- Definiere einen zusätzlicher Block M_{m+1} der Länge $L_H/2$, welcher die Bitlänge der Nachricht M darstellt.
- Wie bei Square-Mod-N werden die Blöcke M_i zu den Blöcke B_i expandiert.
- Bezeichne mit "$\sim L_H$ „ das Wegschneiden der oberesten $L_N - L_H$ Bits.
- Bezeichne mit E den L_H-Bitstring, welche vier Einsen als obersten Bits hat und Nullen als Rest.

Ausgabe :

- L_H-Bit-Hashwert H.

Verlauf :

- $H_0 = 0$.
- Durchführung der Hashwertberechnung für $i = 1, \cdots, m+1$

$$f(M_i, H_{i-1}) = (((B_i \oplus H_{i-1}) \vee E)^2 \mod N) \sim L_H$$

- Gebe $H = f(M_{m+1}, H_m)$ aus.

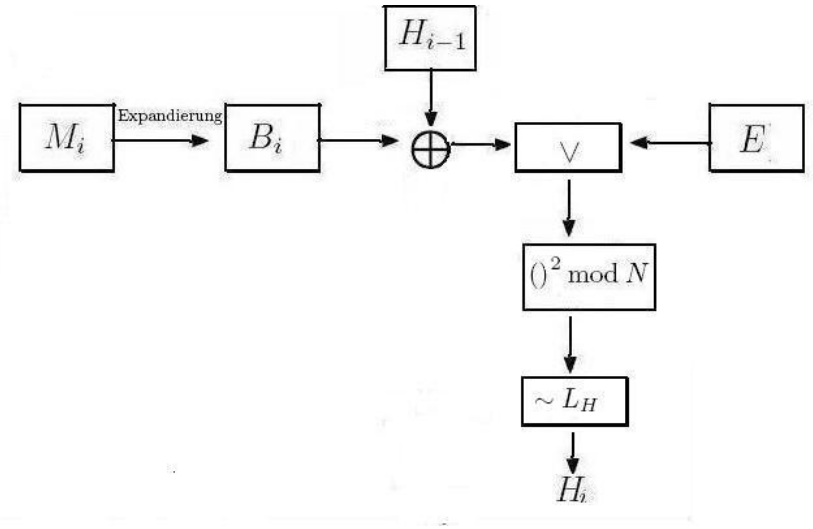

Abbildung 3.4: Funktionsweise der MASH-1-Hashfunktion

Bemerkungen:

1. Bei der Konstruktion vom vorherigen Algorithmus sollte die Bitlänge der Nachricht M nicht größer als $L_H/2$ sein, da die Bitlänge von letztem Block M_{m+1} gleich $L_H/2$ ist.

2. Eine Einfügung des Blocks M_{m+1} erhöht die Sicherheit des Algorithmus. Eine Manipulation der Nachricht führt automatisch zu einer Änderung des Hashwertes, welcher von der Bearbeitung des letzten Blocks abhängt.

3.6.2 Abwehr des Coppersmith-Attacks

Bei diesem neuen Algorithmus enthält $((B_i \oplus H_{i-1}) \vee E)$ vier Einsen in den obersten Bits. Die Idee zur Abwehr des Coppersmith-Attack ist auf solchen Redundanz-Bits basiert. In diesem Fall wird eine multiplikative Beziehung zwischen den Hashwerten zweier unterschiedlicher Nachrichten ausgeschlossen, da im Allgemeinen zwei unterschiedliche Zahlen gleicher Bitlänge niemals ein ganzahliges Vielfaches voneinander sein können, wenn diese eine Eins als führendes Bit enthalten.

3.6.3 Schwäche von MASH-1

Es handelt sich um zwei wichtige Eigenschaften, welche bei der Konstruktion einer sicheren Hashfunktion beachtet werden:
Einerseits wurde bewiesen, dass der MASH-1 Algorithmus anfällig gegen einen Backward-Attack bei bekannter Faktorisierung des Moduls N ist. Es ist also möglich ein H_{i-1} zu finden, wenn ein M_i existiert, so dass $f(M_i, H_{i-1}) = H_i$.
Anderseits sind gelieferte Hashwerte aus MASH-1-Algorithmus nicht gleichverteilt. Der Entropiewert [5] bei MASH-1-Hashfunktion ist $L_H - 4$ (kleiner als der Idealwert L_H) und das liegt daran, dass die Blöcke $((B_i \oplus H_{i-1}) \vee E)$ vier Einsen als redundanten Bits enthalten. MASH-1-Algorithmus ergibt in diesem Fall maximal 2^{L_H-4} unterschiedliche Hashwerte.
Wegen der zwei Schwachstellen von MASH-1-Algorithmus wurde im November 1994 eine Modifikation dieses Algorithmus vorgeschlagen [20].

3.6.3.1 Entwicklung des MASH-1-Algorithmus

Die Idee bei dieser Modifikation besteht darin, dass nach der Potenzierung noch eine Rückkoppelung (XOR) mit dem letzten Zwischen-Hashwert H_{i-1} eingefügt wurde, um die Modulo-Wurzelberechnung formelmäßig zu erschweren. In diesem Fall ist die Kompressionsfunktion eine Einwegfunktion auch bei bekannter Faktorisierung des Moduls, ein Angreifer muss H_{i-1} kennen, bevor er das Wurzelziehen berechnet. Die Modifikationen bei diesem neuen Algorithmus im Vergleich mit der ersten Version des MASH-1-Algorithmus wird wie folgt vorgestellt:

- Der Block M_{m+1}, der aus der Länge der Nachricht M besteht, wird jetzt mit (hex)A (d.h. 1010) statt (hex)F zu einem Block B_{m+1} expandiert.

- Das Ergebnis nach der Potenzierung Modulo N wird durch XOR mit H_{i-1} verknüpft.

[5] Die Entropie einer Zufallsvariablen ist ein Maß für die Abweichung von der Gleichverteilung.

Die Kompressionsfunktion dieser neuen Version sieht damit folgendermaßen aus:

Für $i = 1, \cdots, m+1$

$$f(M_i, H_{i-1}) = (((B_i \oplus H_{i-1}) \vee E)^2 \mod N) \sim L_H \oplus H_{i-1}$$

Bei dieser neuen MASH-1-Version wurde nicht nur die Erschwerung von Umkehrungen der Kompressionsfunktion erreicht, sondern auch die Gleichverteilung der Hashwerte. Es wurde gezeigt, dass diese Hashfunktion den gewünschten Entropiewert L_H hat.

Wegen der Quadrierung hat die Kompressionsfunktion dieser letzten Version eine Schwachstelle und zwar das Auftreten trivialer Kollisionen $x^e = (-x)^e \mod N$. Deswegen wurde es notwendig, eine Erweiterung des MASH-1 zu entwickeln, deren Bezeichnung MASH-2 ist [20].

3.7 MASH-2

Die Idee des MASH-2-Algorithmus besteht darin, ungerade Exponenten anzuwenden, damit die Beziehung $x^e = (-x)^e \mod N$ nicht erfüllt wird. Die Kompressionsfunktion des MASH-2 sieht dann folgendermaßen aus:

$$f(M_i, H_{i-1}) = (((B_i \oplus H_{i-1}) \vee E)^e \mod N) \sim L_H \oplus H_{i-1} \text{ , für } i = 1, \cdots, m+1,$$

wobei der Exponent e den Wert 257 gleichgesetzt wird.

Eigenschaften von MASH-2:
Die Auswahl eines höheren Exponentes ($e = 257$) wirkt sich vorteilhaft auf die Sicherheit des MASH-2-Algorithmus aus. Bei höheren Exponenten findet in der Regel mehr modulo-Reduktionen statt und daraus folgt, dass ein Hashwert nicht leicht von einer Zufallszahl zu unterscheiden ist. Der Nachteil ist aber der größere Zeitaufwand (bei solchen Konstruktionen ($x^{257} \mod N$ werden mindestens 8 Quadrierungen benötigt, wobei x eine beliebige Zahl ist).

3.8 MASH-3

Ein guter Kompromiss zwischen Sicherheit und Performance bietet eine erweiterte Version von MASH-2 namens MASH-3, welche von S Herda im Mai 1996 entwickelt wurde. Bei dieser neuen Hashfunktion wurde der Exponent $e = 3$ statt $e = 257$ verwendet, damit hat zum einen eine Potenzierung mit einem ungeraden Exponenten stattgefunden, zum anderen ergab sich dadurch eine Reduzierung des Rechenaufwandes im Vergleich mit MASH-2.

3.9 Die Chaum-van-Heijst-Pfitzmann-Hashfunktion

Bei Chaum-van-Heijst-Pfitzmann-Hashfunktion [14] handelt es sich um eine Funktion, die auf Basis der Schwierigkeit der Berechnung des diskreten Logarithmus beruht. Sie ist in der Praxis nicht anwendbar, weil sie einen großen Rechenaufwand erfordert.

3.9.1 Chaum-van-Heijst-Pfitzmann-Algorithmus

Gegeben sind:

- p und $q = (p-1)/2$ zwei große Primzahlen.
- a eine Primitivwurzel aus $\mathbb{Z}_p$ und b eine zufällige Zahl aus $\mathbb{Z}_p$.

Die Chaum-van Heijst-Pfitzmann-Hashfunktion sieht wie folgt aus:

$$\begin{aligned} h : \{0, \cdots, q-1\} \times \{0, \cdots, q-1\} &\longrightarrow \mathbb{Z}_p \\ (x_1, x_2) &\longmapsto h(x_1, x_2) = a^{x_1} b^{x_2} \mod p. \end{aligned}$$

3.9.2 Sicherheit des Chaum-van-Heijst-Pfitzmann-Algorithmus

Satz 3.2
Jeder, der leicht eine Kollision von h finden kann, kann genauso leicht den diskreten Logarithmus $\log_a b$ ausrechnen.

Beweis:
Sei $x \neq y$ mit $h(x) = h(y)$ eine Kollision wobei, $x = (x_1, x_2)$ und $y = (y_1, y_2)$ aus $\{0, \cdots, q-1\}^2$.

$$\begin{aligned} h(x_1, x_2) = h(y_1, y_2) \quad &\Longrightarrow a^{x_1} b^{x_2} = a^{y_1} b^{y_2} \mod p \\ &\Longrightarrow a^{x_1 - y_1} = b^{y_2 - x_2} \mod p \end{aligned}$$

Sei d $= ggt(y_2 - x_2, p-1)$.
$p - 1 = 2q$ und q eine Primzahl $\Longrightarrow$ d $\in \{1, 2, q, p-1\}$

1.Fall: $d = 1$
Definiere: $z = (y_2 - x_2)^{-1} \mod (p-1)$, dann gilt:

$$\begin{aligned} b &= b^{z(y_2 - x_2)} \mod p \\ &= a^{z(x_1 - y_1)} \mod p \\ &= a^{(x_1 - y_1)(y_2 - x_2)^{-1}} \mod p \end{aligned}$$

Daraus folgt : $log_a b = (x_1 - y_1)(y_2 - x_2)^{-1} \mod (p-1)$
Damit hat man den diskreten Logarithmus $log_a b$ bestimmt.

2.Fall: $d = 2$
$p - 1 = 2q$ und q ungerade (q Primzahl) $\Longrightarrow$ $ggt(y_2 - x_2, q) = 1$
Definiere: $z = (y_2 - x_2)^{-1} \mod q \Longrightarrow \exists k \in \mathbb{N}$ so daß : $(x_4 - x_2)z = kq + 1$

$$\begin{aligned} b^{z(y_2 - x_2)} &= b^{kq+1} \mod p \\ &= (-1)^k b \mod p \\ &= \pm b \mod p \quad (b^q = -1 \mod p\ ) \\ b^{z(y_2 - x_2)} &= a^{y(x_1 - y_1)} \mod p \\ &= \pm b \mod p \end{aligned}$$

Daraus folgt : $log_a b = (x_1 - y_1)z \mod (p-1)$
oder : $log_a b = (x_1 - y_1)z + q \mod (p-1)$
So erhält man zwei Lösungen, eine von diesen stellt der diskrete Logarithmus $log_a b$ dar.

3.Fall: $d = q$
x_2 und y_2 sind aus $\{0, \cdots, q-1\}$, d.h. $0 \leq x_2 \leq q-1$ und $0 \leq y_2 \leq q-1$

$$\Longrightarrow \quad -q-1 \leq y_2 - x_2 \leq q-1 \ .$$

In diesem Fall kann q kein Teiler von $y_2 - x_2$ sein.
Das ist ein Widerspruch mit $ggt(y_2 - x_2, p-1) = q$.

4.Fall: $d = p-1$
$ggt(y_2 - x_2, p-1) = p-1$ und $y_2 - x_2 \leq p-1$

$$\begin{aligned}
&\Longrightarrow x_2 = y_2 \\
&\Longrightarrow a^{x_1} b^{x_2} = a^{y_1} b^{y_2} \\
&\Longrightarrow a^{x_1} = a^{y_1} \mod p \\
&\Longrightarrow x_1 = y_1 \\
&\Longrightarrow (x_1, x_2) = (y_1, y_2). \text{ Widerspruch mit der Annahme.}
\end{aligned}$$

Diese Hashfunktion ist also Kollisionsresistenz, solange die Berechnung des diskreten Logarithmus schwierig ist.

Beispiel:
Sei $p = 15083 \Longrightarrow q = (p-1)/2 = 7541$.
Wähle $a = 154$ eine Primitivwurzel von $\mathbb{Z}_{15083}$ und $b = 2307$ aus $\mathbb{Z}_{6173}$.
Gegeben sei die Kollision: $a^{7431} b^{5564} = a^{1459} b^{954} \mod 15083 = 10584$.
Suche: $log_{154} 2307$?
In unserem Fall haben wir: $x_1 = 7431$, $x_2 = 5564$, $y_1 = 1459$ und $y_2 = 954$
ggt $(y_2 - x_2, p-1) = 2 \Longrightarrow log_{154} 2307 \in \{z, z+q \mod p-1\}$ mit: $z = (x_1 - y_1)(y_2 - x_2)^{-1} \mod q$.
$(y_2 - x_2)^{-1} \mod q = (954 - 5564)^{-1} \mod 7541 \quad = 4680$.
$\Longrightarrow \quad z = (7431 - 1459) \cdot 4680 \mod 15082 \quad = 2014$.
Man probiert mit: $a^z \mod p = 154^{2014} \mod 15083 = 12776 \neq 2307$.
d.h. $\log_{154} 2307 = (z+q) \mod (p-1)$

$$\begin{aligned}
&= 2014 + 7541 \mod 15082 \\
&= 9555.
\end{aligned}$$

Und damit: $154^{9555} = 2307 \mod 15083$.

3.10 VSH-Hashfunktion

Auf der Konferenz Crypto 2006 haben Contini, Lenstra und Steinfeld eine neue Hashfunktion vorgestellt [35], welche als VSH (Very Smooth Hash) bezeichnet wurde. Da ihre Sicherheit auf einem harten mathematischen Problem beruht, wurde es bewiesen, dass VSH eine Kollisionsresistenz Hashfunktion ist.
Zur formalen Beschreibung der VSH-Hashfunktion werden die folgenden Bezeichnungen als Eingaben verwendet:

- N eine natürliche Zahl, deren Bitlänge größer gleich 1024 ist und welche sich aus von zwei Primzahlen zusammengesetzt ist.
- k bezeichnet die größte natürliche Zahl mit $\prod_{i=1}^{k} p_i \leq N$.

3.10.1 VSH-Algorithmus

Eingabe:

- Eine Nachricht M, bestehende aus l Bits $M_1, M_2, \cdots, M_l$.
- $L = \lceil l/k \rceil$ die Anzahl der Blöcke vom M.
- Setzte $M_i = 0$ für $l \leq i \leq L \cdot k$ (Padding).
- $l = \sum_{i=1}^{k} l_i 2^{i-1}$ mit $l_i \in \{0, 1\}$ ist die Binärdarstellung von M.
- Setze $M_{L \cdot k+i} = l_i$ für $1 \leq i \leq k$.

Ausgabe :

- 1024-Bit-Hashwert H

Verlauf :

- Setzte ein $x_0 = 1$.
- Führe die Hashwertberechnung durch:
  für $j = 0, \ldots, L$ $\qquad x_{j+1} = x_j^2 \prod_{i=1}^{k} {p_i}^{M_{j \cdot k+i}} \mod N$ (4-1)
- Gebe $H = x_{L+1}$ als Hashwert aus.

Bemerkungen :

- Setzt man $e_i = \sum_{j=0}^{L} M_{jk+i} 2^{L-j}$ für alle $1 \leq i \leq k$, so ergibt sich die folgende Beziehung

$$H = x_{L+1} = \prod_{i=1}^{k} {p_i}^{e_i} \mod N. \qquad (4\text{-}2)$$

- Ein Modul mit 1024-Bits entspricht ein k =131 Bits.

3.10.2 Komplexität des VSH-Algorithmus

Der VSH-Algorithmus ist schneller im Vergleich zum Diskreten Logarithmus. Letzterer erfordert durchschnittlich eine modulare Multiplikation für zwei Bits, aber der VSH-Algorithmus nur eine modulare Multiplikation für fünfzig Bits ($N \approx 130$ Bits), wobei man zeigen kann, daß je größer n ist, desto weniger modulare Multiplikationen der Algorithmus erfordert.

Satz 3.3
Der Aufwand einer Iteration ($x_j^2 \prod_{i=1}^{k} {p_i}^{M_{j \cdot k+i}} \mod N$) des VSH-Algorithmus ist kleiner als drei modulare Multiplikationen. Daher ist der Gesamtaufwand von VSH asymptotisch gleich dem Aufwand einer modularen Multiplikation mal $C \cdot (\frac{\log N}{\log \log N})$, wobei C eine Konstante ist.

Beweis
Das Produkt zweier Zahlen mit M bzw. N Bits benötigt höchstens $M \cdot N$ Operationen, um ein Ergebnis mit maximal $M + N$ Bits zu liefern. Daraus folgt, dass die Berechnung von $\prod_{i=1}^{k} {p_i}^{m_{jk+i}}$

höchstens $\sum_{i=1}^{k-1}(\sum_{i=1}^{l} logp_i)logp_{l+1}$ benötigt.

Wegen $\prod_{i=1}^{k} p_i \leq n$ ergeben sich die zwei folgenden Ungleichungen

$$\sum_{i=1}^{l} \log p_i \leq \log N$$

und auch

$$\sum_{i=1}^{l} logp_{l+1} \leq \log N$$

Das liefert einen Rechenaufwand $O((\log N)^2)$. Da noch eine Multiplikation mit x_j^2 durchgeführt wird, ergibt sich daraus, dass der Aufwand einer VSH-Iteration kleiner als drei modulare Multiplikationen ist. Unter Annahme, dass jede Iteration aus der k zu bearbeitenden Bits besteht, mit $k = O\left(\frac{\log N}{\log \log N}\right)$, ergibt sich der Beweis .

■

Bemerkung
Der VSH-Algorithmus ist schneller im Vergleich mit dem schon vorgestellten MASH-1-Algorithmus und allerdings langsamer als der SHA-1-Algorithmus (25 Mal schneller als VSH).

3.10.3 Sicherheit von VSH

Es ist sehr vorteilhaft, wenn die Sicherheit eines Verfahrens auf der Schwierigkeit eines hartes mathematischen Problems beruht. Das ist genau der Fall von VSH. Nämlich wird im Folgenden bewiesen, dass die Sicherheit des VSH-Algorithmus von der Schwierigkeit eines mathematischen Problems abhängt, welches NMSRVS (Nontrivial Modular Square Root of Very Smooth numbers)-Problem genannt wird.

Definition 3.6 (NMSRVS-Problem)
Formal lässt sich das NMSRVS-Problem folgendermaßen definieren:
Gegeben sei eine zusammengesetzte Zahl N, bestehend aus zwei geheimen Primzahlen. Dazu sei $k \leq (\log N)^c$.
Finde ein x mit $x^2 = \prod_{i=0}^{k} {p_i}^{e_i} \mod N$ ($\exists$ mindestens ein i, so daß e_i ungerade ist.)

Das soeben vorgestellte Problem ist ein hartes mathematisches Problem, dies wird durch das folgende Lemma gegeben.

Lemma
Es existiert kein Algorithmus mit einer polynomialen Laufzeit zur Lösung des NMSRVS-Problems.

Wie das Problem mit der Sicherheit der VSH-Hashfunktion zusammenhängt, wird im folgenden Satz erklärt.

Satz 3.4
Lässt sich eine Kollision für VSH-Hashfunktion finden, so ist das NMSRVS-Problem lösbar.

Beweis
Bezeichne mit:

- h die VSH-Hashfunktion.
- M und M' zwei unterschiedlichen Nachrichten.
- l bzw. l' die länge von M bzw. M'.
- L bzw. L' die Anzahl der Blöcke von M bzw. M'.
- $M[j] = (M_{j\cdot k+i})_{1\leq i\leq k}$ bzw. $M'[j] = (M'_{j\cdot k+i})_{1\leq i\leq k}$ der j-te Block der Länge k-Bits von M bzw. M'.

Zeigen wir, dass aus $h(M) = h(M')$ eine Lösung vom NMSRVS-Problem folgt.

Aus $h(M) = h(M') \Longrightarrow x_{L+1} = x'_{L'+1}$ ergibt die folgende Fallunterscheidung:

1. **Falls** $l = l'$
   Sei t die größte natürliche Zahl mit $(x_t, M[t]) \neq (x'_t, M'[t])$. Daraus folgt

   $$x_{t+1} = x'_{t+1}.$$

   Aus der Gleichung (4.1) ergibt sich

   $$x_t^2 \prod_{i=1}^{k} {p_i}^{M_{t\cdot k+i}} = {x'_t}^2 \prod_{i=1}^{k} {p_i}^{M'_{t\cdot k+i}} \mod N \qquad (4.2)$$

   Weiterhin definiere dazu

   - $I = \{i : M_{t\cdot k+i} \neq M'_{tk+i},\ \ 1 \leq i \leq k\}$.
   - $I_{10} = \{i : (M_{t\cdot k+i}, M'_{t\cdot k+i}) = (1, 0),\ \ 1 \leq i \leq k\}$.
   - $I_{01} = \{i : (M_{t\cdot k+i}, M'_{t\cdot k+i}) = (0, 1),\ \ 1 \leq i \leq k\}$.

   Also folgt aus $M_{t\cdot k+i} \in \{0, 1\},\ \forall i$, dass $I = I_{10} \cup I_{01}$ gilt.
   Gemäß (4.3) gilt es

   $$\forall i \notin I,\ x_t^2 \prod_{i\in I_{10}} p_i = {x'_t}^2 \prod_{i\in I_{01}} p_i \mod N \qquad (4.4)$$

   Multipliziere die beiden Seiten der Gleichung (4.3) mit $\prod_{i\in I_{10}} p_i$ und wegen $I = I_{10} \cup I_{01}$ erhält man

   $$x_t^2 \prod_{i\in I_{10}} p_i = {x'_t}^2 \prod_{i\in I} p_i \mod N \qquad (4.5)$$

   Da $ggT(p_i, N) = 1,\ \forall i$ gilt, sind alle Terme in der Gleichung (4.2) invertierbar.
   Damit erfolgt

   $$[\tfrac{x_t}{x'_t} \prod_{i\in I_{10}} p_i]^2 = \prod_{i\in I} p_i \mod N \qquad (4.6)$$

   An dieser Stelle ergeben sich zwei Unterfälle:

(a) Falls $I \neq \emptyset$ liefert die Gleichung (4.5) eine Lösung des NMSRVS-Problems.

(b) Falls $I = \emptyset$ die Gleichung (4.5) liefert

$$(x_t)^2 = (x'_t)^2 \bmod N$$

- $x_t \neq \pm x'_t \mod N$:
  In diesem Fall ist der Modul faktorisierbar und somit ist das NMSRVS-Problem lösbar.
- $x_t = \pm - x'_t \mod N$:
  Wegen $I = \emptyset$, $(x_t, M[t]) \neq (x'_t, M'[t])$ und $M[t] = M'[t]$ ergibt sich

  $$x_t \neq x'_t$$

  Das liefert

  $$x_t = -x'_t \mod N$$

  Aus Definitionen von x_t und $-x'_t$ folgt

  $$x_{t-1}^2 \prod_{i=1}^{k} {p_i}^{M_{(t-1)k+i}} = -{x'_{t-1}}^2 \prod_{i=1}^{k} {p_i}^{M'_{(t-1)k+i}} \mod N$$

  Dann macht man das Gleiche wie oben für $(t-1)$ statt t, um

  $$(\tfrac{x_{t-1}}{x'_{t-1}})^2 = -1$$

  zu bekommen. Daraus erhält man eine Lösung des NMSRVS-Problems.

2. Falls $l \neq l'$
   $l \neq l'$ impliziert $x_{L+1} = x'_{L+1}$ und daher auch

   $$(\tfrac{x_L}{x'_L})^2 = \prod_{i=1}^{k} {p_i}^{l'_i - l_i} \mod N,$$

   wobei l_i und l'_i die i-Bit bzw. i'- Bit in der Binärdarstellung von l bzw. l' darstellen.
   Dann hat man für mindestens ein i mit $|l_i - l'_i| = 1$, da $l \neq l'$ gilt. In diesem Fall strukturiert man genauso wie oben, um (4.5) zu erhalten. Daraus folgt eine Lösung des NMSRVS-Problems.

■

Bemerkung
Für eine Sicherheit äquivalent zu 1024 Bits bei RSA-Modul, braucht man approximativ 1300 Bits für N bei VSH. Deswegen soll man große Zahlen p_i als Basis wählen, um die Faktorisierung von N zu erschweren.

3.10.4 Vorteile von VSH-Hashfunktion

Die wichtigsten Vorteile der VSH Hashfunktion sind:

1. VSH verwendet nur modulare Multiplikationen sowie Multiplikationen in $\mathbb{Z}$ genauso wie bei RSA-Verfahren. Sie kann auch in den selben Umgebungen wie RSA eingesetzt werden.
2. Die vom VSH-Algorithmus erzeugten Hashwerte sind gleichverteilt.
3. Nach dem oben dargestellten Beweis bietet die VSH-Funktion scheinbar eine große Sicherheit.

3.10.5 Nachteile von VSH-Verfahren

Die Nachteile des VSH-Algorithmus sind:

1. In der Praxis ist diese Hashfunktion ungeeignet, denn die Hashwerte haben die gleiche Länge wie N, d.h mehr als 1000 Bits, welche zu groß im Vergleich mit anderen Verfahren wie zum Beispiel SHA-1 oder MD5 (nicht mehr als 160 Bits) sind.

2. Der größte Nachteil bei VSH Verfahren besteht darin, dass VSH eine Falltürfunktion ist. Ist N faktorisierbar, so wird die Berechnung von $\phi(N)$ [6] möglich.
   Sei $e_i' = e_i + g \cdot \phi(n)$, mit g eine beliebige natürliche Zahl. Also es gilt

   $$\prod_{i=1}^{k} {p_i}^{e_i'} = \prod_{i=1}^{k} {p_i}^{e_i + g\phi(N)} = \prod_{i=1}^{k} {p_i}^{e_i}$$

   Somit können bei bekannter Fakorisierung des Moduls N Kollisionen für VSH-Hashfunktion erzeugt werden. Wegen dieser Eigenschaft lässt sich die VSH-Hashfunktion in der Praxis nicht einsetzen.

3. Die VSH-Hashfunktion besitzt die Eigenschaft, dass es möglich ist, durch Änderung von ein paar Bits einer Nachricht zwei unterschiedliche Nachrichten zu konstruieren, deren Hashwerte eine mathematische Beziehung erfüllen.

Bemerkung
Unter Verwendung von N als geheime Schlüssel besteht die Möglichkeit ein MAC mithilfe der VSH-Hashfunktion zu erzeugen,denn ohne Kenntnis von N kann der Angreifer keine Berechnung auf die Blöcke durchführen.

3.10.6 VSH-DL-Hashfunktion

Wie im letzten Abschnitt beschrieben wurde, ist die VSH-Hashfunktion eine Falltürfunktion. Um diese Schwachstelle zu beseitigen, wurde eine erweiterte Version von VSH vorgestellt, welche VSH-DL (VSH Discrete Logarithm) genannt wurde.
Die neue Hashfunktion läuft genauso wie VSH. Dabei wird statt N eine Germainsche Primzahl p mit $p = 2q+1$ als Modul verwendet. Solche Zahlen werden als sicher betrachtet, weil die Berechnung von diskreten Logarithmen auf $\mathbb{Z}_p$ noch schwieriger im Vergleich mit der Berechnung auf anderen Körpern ist.

3.10.6.1 VSH-DL-Algorithmus

Wie schon erwähnt, wird bei VSL-Algorithmus eine Primzahl p als Modul benutzt, dessen Bitlänge im Folgenden mit S bezeichnet wird. Die Anzahl L der Blöcke der Nachricht M wird kleiner als $S-2$ gewählt. Die restlichen Eingaben sind mit denen des VSH-Algorithmus identisch. Der S-Bit-Hashwert $H = x_L$ wird mit diesem neuen Algorithmus folgendermaßen berechnet:

$$x_j = x_{j-1}^2 \cdot \prod_{i=1}^{k} {p_i}^{M_{j\cdot k+i}} \mod p \quad , j = 1, \ldots, L$$

[6] Euler-Funktion

3.10.6.2 Sicherheit von VSH-DL

Die Sicherheit von VSL-Algorithmus hängt von der Schwierigkeit der Lösung von einem mathematischen Problem ab, welches NDLVS-Problem (*Nontrivial Discrete-Log of Very Smooth numbers*) genannt wurde und wie folgt definiert ist:

Definition 3.7 (NDLVS-Problem):
Sei p und q zwei Primzahlen mit $p = 2q + 1$ und $k \leq (\log n)^c$
Beim NDLVS-Problem geht es darum, die folgende Gleichung nach e_i zu lösen:

$$2^{e_1} = \prod_{i=2}^{k} {p_i}^{e_i} \mod p$$

mit

$$\{|e_i| \leq q - 1,\ \forall i\} \wedge \{\exists\, i\ :\ e_i \neq 0\}.$$

Satz 3.5
Existiert ein diskreter Logarithmus von einer der Zahlen p_i zur Basis 2, so kann man NDLVS-Problem lösen.

Beweis
Angenommen, existiert es ein a_i mit $2^{a_i} = p_i$.
Für $e_1 = a_2 e_2$ und $e_i =,\ \forall i \neq 1, 2$ gilt die Gleichung

$$2^{e_1} = \prod_{i=2}^{k} {p_i}^{e_i} \mod p$$

Daraus folgt eine Lösung von NDLVS.

Lemma
Es existiert kein Algorithmus mit polynomialer Laufzeit, der NDLVS-Problem lösen kann.

Satz 3.6
NDLVS-Problem besitzt eine Lösung genau dann, wenn VSH-DL nicht Kollisionsresistenz ist.

Beweis
Weil p eine Primzahl ist, sind die Zahlen p_i auf $\mathbb{Z}_p$ invertierbar.
Aus der Definition einer Kollision für VSH und Umformulierung von (4-2) erhält man:

$$\prod_{i=1}^{k} {p_i}^{a_i} = \prod_{i=1}^{k} {p_i}^{b_i}$$

und somit

$$2^{a_1 - b_1} = \prod_{i=2}^{k} {p_i}^{b_i - a_i},$$

mit mindestens ein i mit $a_i \neq b_i$. Daraus folgt eine Lösung von NDLVS.

Umgekehrt, wenn NDLVS eine Lösung besitzt, d.h. $\exists\ e_i$ mit $2^{e_1} = \prod_{i=2}^{k} {p_i}^{e_i}$, erhält man dann eine Kollision, indem alle Basis mit negativen Exponenten auf die linke Seite gebracht werden.

3.10.6.3 Vorteile der VSH-DL-Hashfunktion

Außer krytographischer Sicherheit hat VSH-DL andere wichtige Vorteile:

- VSH-DL hat keine Falltürfunktion (weil p bekannt ist).
- VSH und VSH-DL haben die gleiche Laufzeit.
- Die vom VSH-DL Hashalgorithmus erzeugten Hashwerte sind gleichverteilt.

Kapitel 4

Gitterbasierte Hashfunktionen

In dem vorhergehenden Kapitel wurde die Konstruktion der Hashfunktionen auf Basis der Schwierigkeit des Problems der Faktorisierung und des diskreten Logarithmus behandelt. Wenn man aber die Quantencomputer zur Verfügung hat, kann man die zwei Probleme mit polynomialer Laufzeit lösen. Es wäre also wünschenswert, durch Nutzung anderer schwieriger Probleme, beweisbar sichere krytographische Hashfunktionen zu realisieren.
1996 hat Ajtai [28] einen Ansatz erfunden, welcher die Forscher in den letzten Jahren veranlasst hat, die Gittertheorie ausgiebig zu studieren. Die Sicherheit zahlreicher Kryptosysteme wird mittels spezieller Gitterprobleme bewertet. Aus diesem Grund besteht die Idee, als Alternative, Hashfunktion-Verfahren auf der Basis von Gittern zu konstruieren.
Das Ziel dieses Kapitels ist es, eine Übersicht zu geben über den Ansatz von Ajtai und wie man die Sicherheit kryptographischer Hashfunktionen mit der Schwierigkeit bestimmter Gitterprobleme zusammensetzen kann.

4.1 Grundlagen

In diesem Abschnitt werden die grundlegenden Definitionen und Aussagen vorgestellt, welche für dieses Kapitel benötigt werden.

4.1.1 Gitter

Definition 4.1
Seien $b_1, b_2, \ldots, b_d$ beliebige unabhängige Vektoren aus $\mathbb{R}^n$.
Die Menge $L := L(b_1, b_2, \ldots, b_d) := \{\sum_{i=1}^{d} c_i b_i : c_i \in \mathbb{Z}\}$, d.h. die Menge der ganzzahligen Linearkombinationen von $b_1, b_2, \ldots, b_d$ nennt man ein Gitter der Dimension d ($\dim(L) = d$).
Wenn $\dim(L) = n$ ist, nennt man das Gitter L volldimensional.
Die Familie von Vektoren $b_1, b_2, \ldots, b_d$ wird eine Basis von L gennant und die Matrix
$B = [b_1, b_2, \ldots, b_d] \in \mathbb{R}^{n \times d}$ ist eine Basismatrix von L.
L wird auch als $L(B)$ bezeichnet.

Die wichtigsten Probleme in der Gittertheorie sind das Shortest Vector Problem (SVP) und das Closest Vector Problem (CVP). Bei der meisten Anwendungen von Gittern wird die Analyse der Sicherheit eines Kryptosystems auf eines von der beiden Problemen reduziert.

4.1.2 Gitterprobleme SVP und CVP

Definition 4.2
Das Shortest Vector Problem lautet :

- Gegeben eine Basismatrix B eines Gitters $L \subset \mathbb{R}^n$.
- Finde ein Vektor $x \in L \setminus \{0\}$, so dass $\|x\|$ minimal ist.
  Wobei $\|.\|$ die euklidische Norm in $\mathbb{R}^n$ ist.

Bezeichne $\lambda_1(L)$ als min $\{\|x\| : x \in L \setminus \{0\}\}$.

Vom SVP gibt es eine approximierte Variante (bezeichnet man mit apprSVP$_\gamma$ oder mit SVP$_\gamma$). In diesem Fall ist ein Vektor $x \in L \setminus \{0\}$, so dass $\|x \| \leq \gamma \cdot \lambda_1(L)$, zu finden.
γ wird als Approximationsfaktor bezeichnet.

Für $\gamma \geq 2$ wurde in [29] bewiesen, dass SVP$_\gamma$ nicht mit polynomialen Zeitaufwand lösbar ist, d.h es gehört zu NP-Problemen.

Definition 4.3
Das closest vector problem (CVP) lautet:

- Gegeben eine Basismatrix B eines Gitters $L \subset \mathbb{R}^n$ und einen Vektor $t \in \mathbb{R}^n$.
- Finde ein $x \in L$, so dass $\|x - t \|$ minimal ist.

Genauso wie bei SVP gibt es auch eine approximierte Variante vom CVP (bezeichnet mit apprCVP$_\gamma$). Hierbei geht es darum, einen Vektor x zu finden, so dass $\|x - t \| \leq \gamma \cdot \|x' - t \|$ ist, $\forall x' \in L$.
Nach [30] ist CVP im Allgemeinen ein NP-Problem.

4.1.3 Rucksackproblem

Das Rucksackproblem nennt man auch Subsetsum-Problem. In [4] wurde bewiesen, dass es ein NP-Problem ist. Aus diesem Grund sind viele Verfahren in der Kryptographie auf der Basis der Rucksackprobleme basiert, aber ausser Okamoto-Verfahren [38] sind die meisten von ihnen gebrochen.

Definition 4.4
Das Rucksackproblem lautet:

- Gegeben ein Vektor $(a_1, a_2, \ldots, a_m) \in \mathbb{N}^m$ und ein $s \in \mathbb{N}$.
- Finde ein $x = (x_1, x_2, \ldots, x_m) \in \{0, 1\}^m$, so dass $\sum_{i=1}^{m} x_i \cdot a_i = s$ ist.

Die a_i werden Gewichte genannt.
Man definiere die Dichte eines Rucksackproblems als $d := \frac{m}{\log_2(\max a_i)_i}$.
Für $d \approx 1$ sind die Rucksackprobleme schwer zu lösen, allerdings besitzen sie in der Regel viele Lösungen für $d \gg 1$.

4.2 Konstruktion gitterbasierter Hashfunktionen

In der Praxis gelten viele Kryptosysteme als sicher, wenn sie auf Problemen basieren, welche im average-case schwer lösbar sind. Es gibt Instanzen, bei denen die Lösung von vielen Problemen wie SVP zum Beispiel nur im worst-case schwer ist. Eine sehr wichtige Idee besteht darin, die Lösung eines schwierigen Problems A in average-case auf die Lösung eines schwierigen Problems B in worst-case zurückzuführen. Diese worst-case/average-case Reduktionsidee wurde zum ersten Mal von Ajtai entwickelt. Wie schon erwähnt, hat dies zur Folge, dass die Forscher ermutigt wurden, die Analyse der Sicherheit von einigen kryptographischen Algorithmen auf Gitterprobleme zu reduzieren. Ein typisches Beispiel dafür sind Konstruktionen von Hashfunktionen bzw Einwegsfunktionen auf der Basis von Gitterproblemen. Begriffe, welche nachfolgend nicht definiert sind, werden im Anhang erläutert.

Das verallgemeinerte Schema zur Konstruktion der Hashfunktionen sieht wie folgt aus:
Sei ein Ring R, eine Untermenge $S \subset R$ und eine natürliche Zahl $m \geq 1$, so definieren wir die folgende Knapsack-Funktionfamilie $\mathcal{H}(R, S, m)$ mit

$$\mathcal{H}(R, S, m) \quad = \quad \{f_a : S^m \to R\}_{a \in R^m},$$

wobei $f_a(x) = \sum_{i=1}^{m} x_i \cdot a_i$.

Der Vektor $a = (a_1, \ldots, a_m)$ wird als der Schlüssel der Knapsack-Funktion bezeichnet.
Durch bestimmte Auswahl der Komponenten R, S und m sowie anderer zusätzlicher Parameter werden im folgenden die wichtigsten der schon erläuterten Hashfunktionen-Verfahren vorgestellt, welche anhand dieser Funktionsfamilie $\mathcal{H}(R, S, m)$ aufgebaut sind.

4.2.1 Ajtai-Hashfunktion

Bei Ajtai Konstruktion werden die Komponenten R und S folgendermaßen gewählt.
$R = \mathbb{Z}_p^n$ und $S = \{0, 1\}$, wobei p und n zwei natürliche zahlen mit $p = p(n) = n^{O(1)}$.
Sei $a = (a_1, a_2, \ldots, a_m)$ einen Schlüssel (Matrix) aus $\mathbb{Z}_p^{n \times m}$.
In [28] hat Ajtai gezeigt, dass die folgende Funktion:

$$\begin{aligned} f_a : \{0,1\}^m &\longrightarrow \mathbb{Z}_p^n \\ x &\longmapsto \sum_{i=1}^{m} x_i \cdot a_i \mod p \quad (x = x_1 x_2 \ldots x_m \text{ bitstrings der Länge } m). \end{aligned}$$

eine Einwegfunktion ist, wenn es für das SVP keinen effizienten Lösungsalgorithmus gibt.
Hierbei erfüllen m und p die folgenden Voraussetzungen: $n \cdot \log p < m < \frac{p}{2 \cdot n^4}$ und $p = O(n^c)$ $(c > 0)$
Noch genauer hat Ajtai bewiesen, dass die Abbildung f_a in average-case schwer zu invertieren ist, wenn es in worst-case schwierig ist, das Minimum eines Gitters bis auf einen Faktor von $O(n^9)$ zu approximieren. Die Beweisidee seines Ergebnisses ist auf die Äquivalenz worst-case/average-case basiert.
In [31] haben Goldreich, Goldwasser und Halevi danach gezeigt, dass die obige definierte Abbildung f_a (unter der gleichen Voraussetzungen) für eine geeignete Wahl der Matrix a eine Kollisionsresistenz Hashfunktion ist.

4.2.2 Micciancio-Einwegfunktion

In [12] hat Micciancio eine Familie von auf Knapsackproblemen basierenden Eiwegfunktionen vorgeschlagen, deren Schwierigkeit der Invertierung im average-Case auf ein hartes in worst case Gitterproblem reduziert wird, nämlich auf das sogennante guaranted distance decoding Problem (GDD).

Definition 4.5
Ein Gitter L heißt zyklisch genau dann,
wenn rot $(x_0, x_1, \ldots, x_{n-2}, x_{n-1}) \in L$, $\forall x = (x_0, x_2, \ldots, x_{n-2}, x_{n-1}) \in L$.
Wobei rot eine Funktion wie folgt definiert:

$$\text{rot } (x_1, x_2, \ldots, x_{n-1}, x_n) = (x_n, x_1, x_2, \ldots, x_{n-1}).$$

Bei Micciancio Konstruktion werden die folgenden Menge bzw. Paramater verwendet:

- $R = (\mathbb{Z}^n_{p(n)}, +, \otimes)$ zyklisch, wobei $\otimes$ bzw. $+$ die Konvolution [1] bzw. die Addition darstellt und $p(n) = p^{O(1)}$, mit p primzahl ist.
- $S = \{0, 1, \ldots, \lfloor p^\delta \rfloor\}^n$, wobei δ eine kleine positive reel Zahl ist und $\lfloor x \rfloor$ bezeichnet die größte ganze Zahl, die kleiner oder gleich x ist.

Definition 4.6
Eine Funktion $f : A \to B$ heisst regulär, falls die folgende Gleichung gilt:

$$|f^{-1}(y_i)| \quad = \quad |f^{-1}(y_j)| \ , \forall \ i \neq j.$$

In diesem Fall besitzen alle Elemente aus B die gleiche Anzahl von Urbilder aus der Menge A.
Diese Eigenschaft besagt, dass die Wahrscheinlichkeit, ein Urbild für jedes Element zu finden, gleich ist. Daher ist das Auffinden solcher Urbilder aufwendig. Für die Micciancio-Funktionsfamilie $\mathcal{H}(R, S, m)$ wurde in [12] bewiesen, dass diese Funktionsfamilie regulär ist.

Definition 4.7
Das guaranted distance decoding Problem (GDD^ϕ_γ) lautet:

- Gegeben eine Basismatrix B eines Gitters $L = L(B)$ und einen Vektor $t \in L(B)$.
- Finde ein $x \in L(B)$, so dass $\text{dist } (t, x) \leq \gamma(n) \cdot \phi(L(B))$ ist, wobei n für die Dimension vom Gitter $L(B)$, $\gamma(n)$ für den Approximationsfaktor dieses Problem stehen und ϕ eine beliebige Funktion auf $L(B)$ definiert ist.
  Im Allgemeinen bezeichnet ϕ die Durschmesse von $L(B)$.

Bemerkung:
Die Funktion ϕ dient dazu, die Qualität der Lösung eines Gitterproblems zu messen. Wenn $\phi = \rho$ ist, bezeichnet man das (GDD^ϕ_γ) einfach mit (GDD_γ), wobei $\rho(L(B))$ bezeichnet die Durschmesse von $L(B)$.

[1] Die Konvolution von zwei Elemnten x und y ist wie folgt definiert: $x \otimes y = \text{Rot } (x) \cdot y$

Vom (GDD^{ϕ}_{γ}) gibt es eine Approximationsvariante, welche für eine reel Zahl $c > 2$ mit $IncGDD^{\phi}_{\gamma,c}$ (incremental guaranted distance decoding Problem) bezeichnet wird.

Dieses Problem lautet folgendermaßen:

- Gegeben eine Basismatrix B eines n-dimensionalen Gitters $L = L(B)$, einen Vektor $t \in L(B)$, eine Menge S von linear unabhängigen Vektoren aus $L(B)$ und eine reelle Zahl r mit $r > \gamma(n) \cdot \phi(L(B))$.
- Finde ein Vektor $s \in L(B)$, so dass $\|s - t\| \leq (\|S\| + r)/c$. Wobei $\|S\| = \max(\|s_i\|)_i$, mit $S = (s_1, s_2, \ldots, s_n)$.

Das Gitterproblem $IncGDD^{\phi}_{\gamma,c}$ spielt eine zentrale Rolle zum Beweis der Einwegeigenschaft der Funktionsfamilie $\mathcal{H}(R, S, m(n))$. Genauer beweist Micciancio, dass das Lösungsproblem der Invertierung der Funktionsfamilie $\mathcal{H}(R, S, m(n))$ in average-case auf die Lösung des $IncGDD^{\phi}_{\gamma,c}$ Problems in worst-case reduziert kann. Dieses Ergebnis wird im folgenden Satz erläutert.

Satz 4.1
Seien eine Konstante c' mit $c' > 2 \cdot c, \delta > 0$ und eine vernachlässigbare Funktion $\epsilon(n) = n^{-w(1)}$. Wenn $m(n) = w(1)$ und $p(n) \geq (c' \cdot m(n) \cdot n^{2 \cdot 5})^{1/(1-\delta)}$, so kann man die Schwierigkeit der Invertierung der Micciancio-Funktionsfamilie $\mathcal{H}(R, S, m(n))$ in average case auf die worst case Schwierigkeit des $IncGDD^{\phi_\epsilon}_{\gamma(n),c}$ Problem zurückführen, wobei $\gamma(n) = c' \cdot m(n) \cdot n \cdot p(n)^{\delta}$ ist.

Für den Beweis siehe ([12],Theorem 4.9)

Definition 4.8 ($SIVP^{\phi}_{\gamma}$)
Das ($SIVP^{\phi}_{\gamma}$) (shortest independent vector problem) lautet:

- Gegeben ein n-dimensionales Gitter $L(B)$.
- Finde eine Menge S von n unabhängigen Vektoren aus $L(B)$, sodass $\|S\| \leq \gamma(n) \cdot \phi(L(B))$ ist.

Korollar 4.1
Für jede $\epsilon > 0, p(n) = n^{2 \cdot 5 + \theta(1)}$ und $m(n) = w(1) \leq n^{o(1)}$, existiert es $\delta > 0$, so dass für $\gamma(n) = n^{1+\epsilon}$ $\mathcal{H}(R, S, m(n))$ ist eine Einwegfunkionsfamilie in average-case genau dann wenn, eins von den beiden folgenden Problemen in worst-case lösbar ist:

- Das guaranteed distance decoding Problem GDD^{δ}_{γ} in zyklischen Gittern.
- Das shortest independent vector Problem $SIVP^{\delta}_{\gamma}$ in zyklischen Gittern.

4.2.3 Peikert-Rosen-Hashfunktion

Aufbauend auf der vorhergehenden Konstruktion haben Peikert und Rosen eine Kollisionsresistenz Hashfunktion entwickelt. In ihrem Paper [7] haben sie gezeigt, dass es recht einfach ist, Kollisionen für die Micciancio-Einwegsfunktion zu konstruieren. Dies kann folgendermaßen geschehen:
Betrachten wir $x, x' \in S^m \subset \mathbb{Z}_p^{n \times m}$ und $a \in R^m = \mathbb{Z}_p^{m \times n}$. Es ist klar, dass die Funktion f_a linear

ist, weil es gilt: $f_a(x + x') = f_a(x) + f_a(x')$.
Ist x, x' eine Kollision für f_a, so muss gelten :

$$f_a(x) \quad = \quad f_a(x').$$

Dies impliziert

$$f_a(x) - f_a(x') \quad = \quad 0.$$

Aus der Linearität von f_a folgt

$$f_a(x - x') \quad = \quad 0.$$

Das bedeutet, dass die Suche nach Kollisionen sich reduziert auf die Suche nach Elementen x mit der folgenden Gleichung:

$$f_a(x) \quad = \quad 0.$$

In diesem Fall ist es aureichend, ein $x \neq 0$ mit $f_a(x) = 0$ zu finden.
Bei der Micciancio-Konstruktion kann man davon ausgehen, dass der Ring R zu einem Ideal aus $(\mathbb{Z}_p[\alpha]/\alpha^n - 1)$ isomorph ist, indem man ein Element $x = (x_1, \ldots, x_n)$ aus R als ein Polynom $x(\alpha) = x_0 + x_1 \cdot \alpha + \ldots + x_{n-1} \cdot \alpha^{n-1}$ in $\mathbb{Z}_p[\alpha]/\alpha^n - 1$ bezeichnet. Dadurch ergibt sich:

$$f_a(x) \quad = \quad \sum_{i=1}^{m} x_i(\alpha).a_i(\alpha) \mod (\alpha^n - 1)$$

Sei $x = (x_1, \ldots, x_m) \in R^m$, mit

$$x_j = \begin{cases} \frac{\alpha^n - 1}{\alpha^q - 1} = \alpha^{n-q} + \alpha^{n-2q} + \ldots + 1 & j = 1, \\ 0 & \forall j \neq 1, \end{cases}$$

wobei q die kleinste natürliche Zahl ist (einschliesslich $q = 1$), welche n teilt. Also gilt

$$f_a(x) \quad = \quad x_1(\alpha) \cdot a_1(\alpha) \mod (\alpha^n - 1).$$

Ist $a_1(\alpha)$ durch $\alpha^q - 1$ teilbar, so kann $x_1(\alpha) \cdot a_1(\alpha)$ durch $\alpha^n - 1$ mit einer Wahrscheinlichkeit von $\frac{1}{p^q}$ teilbar sein und somit ist $f_a(x) = 0$. Daraus folgt, dass die Micciancio-Hashfunktion nicht Kollisionsresistenz ist.

Wie man feststellt, nutzt der soeben vorgestellte Angriff die Tatsache aus, dass $\alpha^n - 1$ nicht irreduzibel in $\mathbb{Z}_p[\alpha]$ ist. Um dieses Problem zu umgehen, muss die algebraische Struktur der Menge S geändert werden, damit keine Kollision auftritt.
Zu diesem Zweck wird die folgende Definitionsbereich definiert:

$$S_{D,\Phi} \quad = \quad \{x \in \mathbb{Z}_p^n : \| x \|_\infty \leq D \; und \; \Phi(\alpha) \; teilt \; x_{\mathbb{Z}}(\alpha) \in \mathbb{Z}\,[\alpha]\},$$

wobei

- $\Phi(\alpha) = \frac{\alpha^n - 1}{\Phi_k(\alpha)}$ mit k ein Teiler von n ist und $\Phi_k(\alpha) = \prod_{\substack{1 \leq c \leq k \\ c \wedge k = 1}} (\alpha - e^{\frac{2 \cdot i \cdot \pi \cdot c}{k}})$.
- $v_{\mathbb{Z}}(\alpha)$ ist der einzige Vertreter von $v \in \mathbb{Z}_p$ in der Menge $(-p/2, p/2]$
- Für jedes $x \in \mathbb{Z}_p^n$ gilt: $\| x \|_\infty \in [0, p/2]$

In diesem Fall erhält man eine neue Knapsack-Funktionsfamilie $\mathcal{H}(\mathbb{Z}_p^n, S_{D,\Phi}, m)$.

4.2.3.1 Sicherheit der Peikert-Rosen-Hashfunktion

Wie schon erwähnt, ist die Peikert-Rosen-Hashfunktion Kollisionsresistenz und deren Sicherheit ist auf die Schwierigkeit der Lösung von ein bestimmtes Gitterproblem reduziert kann, welches mit $\text{SubIncSVP}_\gamma^{\eta_\epsilon}$ bezeichnet wird. Im Folgenden wird dieses Problem definiert sowie sein Beziehung zur Peikert-Rosen-Hashfunktion als Satz erläutert.

Definition 4.9
Das Problem $\text{SubIncSVP}_\gamma^{\eta_\epsilon}$ lautet:

- Gegeben ein Gitter $L(B)$ mit vollem Rang n, ein Polynom Φ mit $\Phi(\alpha) \neq 0 \mod (\alpha^n - 1)$, welche $(\alpha^n - 1)$ teilt und ein Vektor $(c \neq 0)$ mit $c \in L(B) \cap H_\Phi$, so dass $\|c\| > \gamma(n) \cdot \eta(L(B) \cap H_\Phi)$.
- Finde einen Vektor $c' \in L(B) \cap H_\Phi$, so dass $\|c'\| \leq \|c\|/2$ ist.

Satz 4.2
Seien $D(n), m(n), p(n)$ drei polynomial beschränkte Funktionen und eine vernachlässigbare Funktion $\epsilon(n)$, so dass $p(n) \geq 8 \cdot n^{2,5} \cdot m(n) \cdot D(n)$ und $\gamma(n) \geq 16 \cdot n \cdot m(n) \cdot D(n)$.
Das Auffinden von Kollisionen in $\mathcal{H}(\mathbb{Z}_{p(n)}^n, S_{D(n),\Phi}, m(n))$ kann auf die Lösung von $\text{SubIncSVP}_\gamma^{\eta_\epsilon}$ mit Polynomialzeit-Algorithmus reduziert werden.

Korollar 5.2
Für jede $\delta > 0$ existieren $D(n) = n^{\theta(1)}$, $p(n) = n^{2.5+\theta(1)}$ und $m(n) = \theta(1)$.
Das Auffinden von Kollisionen für die Hashfunktionfamilie $\mathcal{H}(\mathbb{Z}_{p(n)}^n, S_{D(n),\Phi_1(\alpha)}, m(n))$ kann mit einer grossen Wahrscheinlichkeit auf die Lösung in worst-case des SVP_γ (mit $\gamma(n) = n^{1+\delta}$) reduziert werden.

Die schon vorgestellten Ergebnisse können in einer Tabelle zusammengefasst werden.

	Sicherheit	**Laufzeit**	**Gitter**	**bz.Problem**	**Approx-faktor**
Ajtai	CRHF	$O(n^2)$	Allgemein	SVP	$poly(n)$
Micciancio	OWF	$\tilde{O}(n)$	Zyklich	GDD	$n^{1+\epsilon}$
Peikert-Rosen	CRHF	$\tilde{O}(n)$	Zyklich	SVP	$O(n)$

Tabelle 4.1: Vergleich der drei dargestellten Hashfunktionenfamilien

Wobei, OWF: Einwegfunktion, CRHF: Kollisionresistent Hashfunktion und bz.Problem: entsprechendes Problem in Gitter.

4.3 Beispiele von Gitterbasierten Hashfunktionen

In diesem Abschnitt werden wir zwei Hashfunktionen auf Basis von Gitterproblemen vorstellen und deren Funktionsweise erläutern.

4.3.1 FFT-Hashfunktion

Das erste Beispiel für eine Gitterbasierte Hashfunktion ist die Fast Fourier Transformation Hashfunktion (FFT). Diese Hashfunktion wurde zuerst von Schnorr 1991 entwickelt [9],[10],[11] und später kamen weitere Versionen hinzu, welche als nicht kryptographisch sicher gelten.
Aufbauend auf der Fast Fourier Transformation haben Lyubashevsky, Micciancio, Peikert und Rosen auf dem zweiten Hash Workshop im August 2006 eine neue Version von FFT Hahsfunktionen vorgestellt [25]. Es wurde gezeigt, dass diese Version Kollisionsresistenz ist.

4.3.1.1 Beschreibung von FFT-Hashfunktionen

Die Konstruktion von FFT-Hashfunktionen basiert auf zwei wichtigen Eigenschaften, zum einen einer Konfusion und zum anderen einer Diffusion. Die Konfusion ist durch eine Fourier Transformation gewährleistet, während die Diffusion mittels einer linearen zufälligen Funktion sichergestellt wird. Dadurch erfolgt eine einfache Impementierung dieser Funktion sowohl in Software als auch in Hardware. Nachfolgend sehen wir eine schematische Darstellung der Konstruktion der FFT-Kompressionsfunktion.

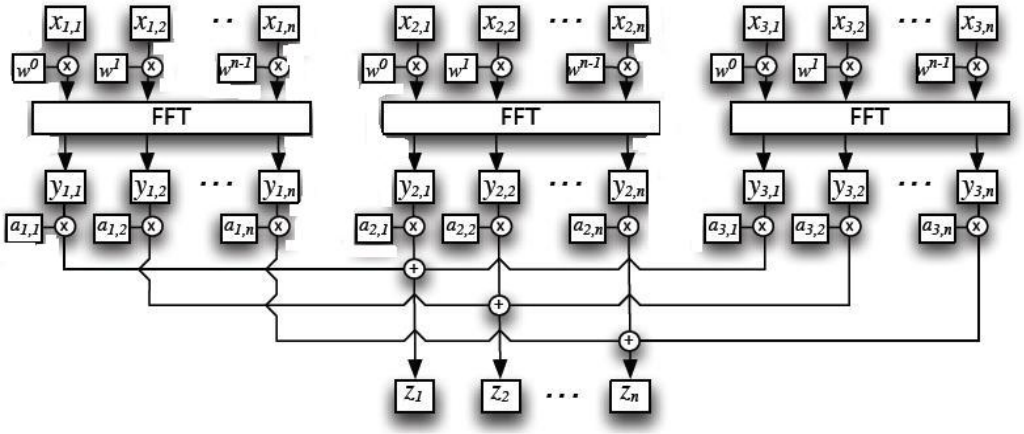

Abbildung 4.1: FFT-Kompressionsfunktion mit $m = 3$

Im Folgenden wird zunächst die Konstruktion der FFT-Kompressionsfunktion vorgestellt.

4.3.1.2 FFT-Algorithmus

Zur Konstruktion von FFT-Kompressionsfunktion werden die folgenden Parameter festgelegt:

- Der Input der Kompressionsfunktion wird als $m \times n$ Matrix $X = (x_{ij})$ dargestellt, wobei n eine Zweierpotenz ist.
- $d = \max(x_{ij})$.
- p eine Primzahl der Form $2 \cdot t \cdot n + 1$, wobei t eine positive ganze Zahl ist. Dieses p stellt sicher, dass ein w aus der multiplikativen Gruppe von $\mathbb{Z}_p$ (bezeichnet mit $\mathbb{Z}_p^*$) existiert, dessen Ordnung $2 \cdot n$ beträgt. Der Beweis dafür ist Folgender:

  Ist β ein n-te Primitivwurzel von $\mathbb{Z}_p^*$, so gilt: $\beta^n = 1$ und $\beta^k \neq 1$, $\forall k < n$. Setzen wir dann $\beta = w^2$.

Algorithmus

Eingabe:

- Schlüssel: Eine $m \times n$ Matrix $A = (a_{ij})$, wobei die Elemente a_{ij} aus $\mathbb{Z}_p$ zufällig und unabhängig ausgewählt werden.
- Eine $m \times n$ Matrix $X = (x_{ij})$, wobei die Elemente $x_{ij} \in D = \{0, 1, \ldots, d\} \subset \mathbb{Z}_p$ mit $m \cdot n \cdot \lfloor \lg(d+1) \rfloor$ Bitlänge.

Ausgabe:

- Ein Wert z mit $n \lceil \log p \rceil$ Bitlänge.

Verlauf:

- $\forall i, j$, sei $x' = w^{j-1} \cdot x_{ij}$.
- $\forall\ i = 1, \ldots, m$, sei $(y_{i,1}, \ldots, y_{i,n})$= mFFT $(x'_{i,1}, \ldots, x'_{i,n})$, wobei die Bezeichnung mFFT für die modulare Fast Fourier Transformation steht.
- $\forall\ j = 1, \ldots, n$, Sei $z_j = a_{1,j}y_{1,j} + \ldots + a_{m,j}y_{m,j}$.
- Gebe $z = (z_1, \ldots, z_n)$ aus.

Die FFT-Kompressionsfunktion (bzw. FFT-Hashfunktion) bildet aus einer Nachricht der Bitlänge $m \cdot n \cdot \lfloor \lg(d+1) \rfloor$ (bzw. beliebiger Länge) einen Hashwert der Bitlänge $n \cdot \lceil \lg p \rceil$.

FFT-Modelle: Die zwei folgenden Tabellen stellen drei konkrete Beispiele von FFT-Hashfunktionen mit deren Eigenschaften (Parametern ,die Bitlänge der Eingaben und Ausgaben) sowie deren Performance im Vergleich mit den Standard-Hashfunktionen SHA 1 und SHA 2 dar.

	n	**m**	**d**	**p**	**Input(bits)**	**Output(bits)**
Bulk	1024	16	15	$\approx 2^{28}$	65536	28672
Mini	128	8	3	257	2048	10252
Nano	64	8	3	257	1024	513

Tabelle 4.2: Die drei FFT-Kompressionsfunktionen

	Bulk	**Mini**	**Nano**	**SHA-1(openssl)**	**SHA-1(sha1sum)**	**SHA-256**
Rate (MB/s)	0,43	0,69	0,67	155	35	42

Tabelle 4.3: Die Performance von Bulk, Mini und Nano im Vergleich mit SHA-1 und SHA-256

Nach Tabelle (4.3) kann man feststellen, dass der FFT-Hash-Algorithmus 60 mal langsamer als die Standard Hash-Algorithmen SHA-1 und SHA-256 ist.

4.3.1.3 Sicherheit des Algorithmus

Es wird im Folgenden eine algebraische Funktion g_A (A ist eine Schlüssel-Matrix) definiert, welche Kollisionsresistenz ist und zum Beweis der Sicherheit von FFT-Kompressionsfunktion f dient.

4.3.1.4 Definition 4.10

Es seien $R = \mathbb{Z}_p[\alpha]/\langle f \rangle$, mit $f = \alpha^n + 1$ und $S = \{x = x_1 + x_2\alpha + \ldots + x_n\alpha^{n-1} \in R : \forall i, x_i \in D\}$ die Menge aller Polynomen aus R mit Koeffizienten aus der Menge D.
Die Funktion g_A ist dann wie folgt definiert:

$$g_A\colon S^m \longrightarrow R$$

mit

$$g_A(x_1, \ldots, x_m) = \sum_{j=1}^{m} a_j.x_j,$$

wobei $A = (a_1, \ldots, a_m) \in R^m$ mit (a_i) zufällig und gleichverteilt aus R gewählt sind.

Im Folgenden wird ein Satz vorgestellt, welcher zeigt, dass das Auffinden einer Kollision von g_A die Lösung von Shortest Vector Problem in jedem Gitter isomorph zu einem ideal in $\mathbb{Z}_p[\alpha]/\langle f \rangle$ impliziert.

4.3.1.5 Satz 4.3

Es seien $p > 6 \cdot d \cdot m \cdot n^{1,5} \cdot \log^2 n$ und $\gamma = 72 \cdot d \cdot m \cdot n \cdot \log^2 n$.
Findet man eine Kollision für die Funktion g_A, so kann man das Shortest Vector Problem SVP_γ in Polynomialer Zeit in allen Gittern isomorph zu einem Ideal aus $\mathbb{Z}_p[\alpha]/\langle f \rangle$ lösen.

Beweis: Der Beweis des obigen Satzes findet man in [7] (Theorem 4) mit $f = \alpha^n - 1$ statt $\alpha^n + 1$ in unserem Fall.

4.3.1.6 Satz 4.4

Die FFT-Kompressionsfunktion f ist Kollisionsresistenz genau dann, wenn die Funktion g_A Kollisionsresistenz ist.

Folgerung
Es seien $p > 6 \cdot d \cdot m \cdot n^{1,5} \cdot \log^2 n$ und $\gamma = 72 \cdot d \cdot m \cdot n \cdot \log^2 n$.
Findet man eine Kollision für FFT-Hashfunktion, so kann man das Shortest Vector Problem SVP_γ

in Polynomialer Zeit in allen Gittern isomorph zu einem Ideal aus $\mathbb{Z}_p[\alpha]/\langle f \rangle$ lösen.

Bemerkung
Für die soeben vorgestellten Beispiele von FFT-Hashfunktion kann man bemerken, dass die Bulk die einzige Funktion ist, deren Sicherheit mittels Satz 1 bewiesen werden kann. Für die restlichen Funktionen (Mini , Nano) ist die Vorausetzung $p > 6 \cdot d \cdot m \cdot n^{1,5} \cdot \log^2 n$ nicht erfüllt. Allerdings bleibt das Auffinden einer Kollision für diese Funktionen schwer.

4.3.1.7 Preimage Attack gegen FFT

Donghoon Chang [17] hat gezeigt, dass eine FFT-Hashfunktion anfällig gegen Einwegeigenschaft ist. Für eine t-Hashwertlänge kann ein Urbild in $2^{t-n.\log_2(d+1)}$ Operationen gefunden werden.
Als Beispiel konnte ein Urbild bei Nano-Funktion mit einer Komplexität von 2^{385}(und nicht 2^{513}) gefunden werden.

Ein anderer möglicher Angriff auf FFT-Hashfunktion ist der sogenannte Gitter-Angriff. Die Idee bei diesem Angriff besteht darin, dass das Finden einer Kollision für FFT-Hashfunktion äquivalent zur Ermittlung eines Vektores x mit $x \cdot A = 0$ ist, wobei $A = (Rot(a_1), \ldots, Rot(a_m))$ eine $m \cdot n \times n$ Matrix darstellt. Ein zweiter möglicher Angriff ist der sogennante Birthday Angriff . In [15] wurde gezeigt, dass das Auffinden einer Kollision für FFT-hashfunktion $2^{\frac{n \cdot \log p}{\log p+1}}$ Operationen benötigt. Im Zusammenhang mit unseren drei Beispielen ist dieser Angriff in der Praxis unmöglich durchführbar.

4.3.2 LASH-Hashfunktion

Die LASH-Hashfunktion wurde auf dem zweiten Hash Workshop 2006 von Bentahar, Page, Silverman, Saarinen und Smart vorgeschlagen [24]. Die Idee zur Konstruktion dieser Hashfunktion ist auf dem Prinzip von Miyaguchi-Preneel-Hashingschema [37] basiert. Der Unterschied ergibt sich dadurch, dass statt Block-Chiffren bei Miyaguchi-Preneel Schema eine Modular Matrix Multiplikation angewendet wird.
LASH-Hashfunktion haben als Vorteile, dass sie zum einen eine einfache Struktur haben und zum anderen zu den kryptographischen beweisbar sicheren Hashfunktionen zählen, deren Sicherheit auf der Lösung eines harten Problems in einem bestimmten Gitter basiert.

4.3.2.1 Eigenschaften von LASH-Hashfunktion

Für die Bezeichnung LASH existieren drei verschiedene Übersetzungen, wobei jede von diesen eine bestimmte Eigenschaft deutlich macht:

- LASH (Linear Algebra based Secure Hash): Diese englische Bezeichnung verdeutlicht die algebraische Struktur dieser Funktion. Zur Ausführung der LASH-Algorithmus wird eine matrizielle Multiplikation auf der Basis von modularer Arithmetik angewendet.

- LASH (Lattice based Secure Hash): Diese englische Begriffe zeigen, dass die Sicherheit der LASH-Hashfunktion von der Lösung eines harten Problems in einem bestimmten Gitter abhängt. Sie ist dann eine beweisbar sichere Hashfunktion.

- LASH (Light-weight Arithmetical Secure Hash): Diese Begriffe weisen auf einen weiteren Vorteil dieser Funktion hin, nähmlich dass sie ein einfaches Design hat und deshalb einfach zu implementieren ist.

4.3.2.2 Konstruktion der LASH-Kompressionsfunktion

Bevor die Konstruktion des LASH-Kompressionsfunktion-Algorithmus vorgestellt wird, werden erstmals die wichtigsten benötigten Komponenten definiert.

- Sei ein Intialwert $y_0 = 54321$.
  Es wird dann eine Folge $(y_i)_{i\geq 1}$ iterativ durch die folgenden Beziehung konstruiert :
  $y_{i+1} = y_i^2 + 2 \mod (2^{31} - 1)$.
  Schliesslich wird eine Zufallsfolge durch die Beziehung $a_i = y_i \mod 256$ erzeugt.
  Konkret hat man als erste Glieder der Folge $(a_i)_{i\geq 1}$, $a_0 = 49, a_1 = 100, a_2 = 135, a_4 = 95$.

- Durch die Pollard-PRNG-Methode wird die Folge (a_i) angewendet, um eine zyklische $m \times n$ Matrix H zu erzeugen. Diese Struktur der Matrix H hat wichtige Vorteile, einerseits ist der LASH-Kompressionsfunktion-Algorithmus damit einfach zu implemetieren und anderseits wird dazu nicht viel Speicherplatz benötigt.
  Diese Matrix sieht wie folgt aus:

$$H = \begin{bmatrix} a_0 & a_{n-1} & a_{n-2} & \dots & a_2 & a_1 \\ a_1 & a_0 & a_{n-1} & \dots & a_3 & a_2 \\ \vdots & & \ddots & & & \vdots \\ a_{m-1} & a_{m-2} & a_{m-3} & \dots & a_{m+1} & a_m \end{bmatrix}$$

Die LASH-Kompressionsfunktion f ist so folgendermassen definiert:

$$\begin{aligned} f : \mathbb{Z}_q^m \times \mathbb{Z}_q^m &\longrightarrow \mathbb{Z}_q^m \\ (r, s) &\longmapsto (r \oplus s) + f_H(r||s) \mod q \end{aligned}$$

,

wobei:

- $\oplus$ bezeichnet man als eine byte-weise XOR.
- $+$ bezeichnet man als eine Byte-weise Addition modulo 256.
- $||$ bezeichnet die Konkatenation von zwei Strings
- Die Funktion f_H ist wie folgt definiert $f_H(r, s) = H \cdot (r||s)^t$ (bei dieser Multiplikation ist (r||s) bitweise dargestellt).
- Die konstante q nimmt man gleich 256.

LASH-Kompressionsfunktion-Algorithmus

Eingabe:

- Eine byte-Folge $r_0, r_1, \ldots r_{m-1}$.
- Eine byte-Folge $s_0, s_1, \ldots s_{m-1}$.

Ausgabe:

- Eine m-Byte-Wertlänge $f(r, s)$.

Verlauf :

for $i = 0, 1, \ldots, m-1$ **do**
  $t_i \longleftarrow r_i \oplus s_i$
end for
for $i = 0, 1, \ldots, n$ **do**
  if $i < 8 \cdot m$ **then**
    $x \longleftarrow \left\lfloor 2^{-(7-(i mod 8))} r_{\lfloor (i/8) \rfloor} \mod 2 \right\rfloor$
  else
    $x \longleftarrow \left\lfloor 2^{-(7-(i mod 8))} r_{\lfloor (i/8) \rfloor - m} \mod 2 \right\rfloor$
  end if
  if $x = 1$ **then**
    for $j = 0, 1, ..., m-1$
      $t_j \longleftarrow t_j + a_{((n+j-i) \mod n)} \mod 256$
    end for
 end if
end for
Gebe $t = (t_0, t_1, \ldots t_{m-1})$ aus.
Setze $f(r, s) = t$.

Die folgende Tabelle veranschaulicht vier konkrete Beispiele von LASH-Kompressionsfunktionen. Die Zahl n (bzw. m) stellt die Bitlänge (bzw.Byte-Länge) der Eingaben (bzw.der Ausgaben) dar. Man kann feststellen, dass m und n die folgende Beziehung $m = n/16$ erfüllen.

Variant	n	m
LASH-160	640	40
LASH-256	1024	64
LASH-384	1536	96
LASH-512	2048	128

Tabelle 4.4: Beispiele von LASH-Kompressionsfunktionen

4.3.2.3 LASH-Algorithmus

Der LASH-Hashfunktionsalgorithmus kann in drei grundlegenden Schritte unterteilt werden:

- Der erste Schritt stellt die Vorbereitungsphase dar. Sei M eine Nachricht mit einer beliebigen Bitlänge l, welche in Blöcke $M_0, M_1, \ldots M_{\lceil l/8m \rceil - 1}$ der Länge $8 \cdot m$ (m Bytes) aufgeteilt wird. Ist die Nachricht M keine Vielfache von $8 \cdot m$, so wird erstmal mit „1“ Bit und den Rest mit Nullen gepaddet.

- In dem zweiten Schritt wird der Vektor r mit Null initialisiert und anhand der Kompressionsfunktion wird iterativ eine m Byte-Wertlänge geliefert.

- Bei dem letzten Schritt werden die vier obersten Bits jedes Byte von der gelieferten Wert aus der zweiten Schritt ausgewählt, welche danach konkateniert werden. Daraus folgt ein Wert mit $m/2$ Byte-Länge, welcher den Hashwert der Nachricht M darstellt.

Der genaue Verlauf der LASH-Hashwertberechnung ist im folgenden Algorithmus wiedergegeben.

Algorithmus

Eingabe:
Eine Nachricht $M = M_0, M_1, \ldots M_{\lceil l/8m \rceil - 1}$.

Ausgabe:
Eine $m/2$ Byte-Hashwert t der Nachricht M.

Verlauf:
for $i = 0, 1, \ldots, m-1$ **do**
  $r_i = 0$
end for
for $i = 0, 1, \ldots, \lceil l/8m \rceil - 1$ **do**
  for $j = 0, 1, \ldots, m-1$ **do**
   $s_j = M_{m \times i + j}$ **end for**
  $r \longleftarrow f(r, s)$
end for
for $i = 0, 1, \ldots, m-1$ **do**
  $s_i \longleftarrow \lfloor l/2^{8i} \rfloor \mod 256$
end for
$r \longleftarrow f(r, s)$
for $j = 0, 1, ..., m/2 - 1$ **do**
  $t_i = 16 \lfloor r_{2i}/16 \rfloor + \lfloor r_{2i+1}/16 \rfloor$
end for
Gebe t aus

4.3.2.4 Sicherheit von LASH-Hashfunktionen

Die LASH-Kompressiosnfunktionen besteht aus einer Kombination von einer XOR-Operation mit einer linearen Funktion f_H, welche die Basis für den Beweis der Sicherheit des LASH-Algorithmus ist.
Wie wir schon bei der Ajtai Konstruktion am Anfang dieses Kapitel gesehen haben, wurde In [31] von Goldreich gezeigt, dass die Sicherheit der Funktion f_H von der Schwierigkeit der Lösung des SVP-Problems in einem gewählten Gitter abhängt. Im Folgenden wird dieser Zusammenhang noch ausführlicher dargestellt.
Seien H eine $m \times n$ Matrix und q eine ganze Zahl (nicht unbedingt Primzahl),
so ergibt sich die folgende Abbildung:

$$f_H : \mathbb{Z}^n \longrightarrow (Z/q\mathbb{Z})^n$$

$$b \longmapsto H.b \mod q$$

Man definiert das Gitter L_H wie folgt :

$$L_H = \{x \in \mathbb{Z}^n : H.x = 0 \mod q\}$$

Da $q \cdot \mathbb{Z}^n \subset L_H \subset \mathbb{Z}^n$, es ist klar, dass n der Dimension von L_H ist.

Definition 4.11
Ein binärer (bzw. ternärer) Vektor in einem Gitter L ist definiert als ein Vektor mit Koordinaten nur aus $\{0, 1\}$ (bzw. $\{-1, 0, 1\}$) sind.
Die Menge aller binären (bzw. ternären) Vektoren wird als $\mathfrak{B}_\mathfrak{n}$ (bzw.$\mathfrak{T}_\mathfrak{n}$) bezeichnet.

Für eine geeignete Wahl der Matrix $H \in M_{m,n}(\mathbb{Z}_q)$ haben Goldreich, Goldwasser und Halevi in [31] so Folgendes gezeigt:

- Wenn die definierte Abbildung f_H Kollisionsresistenz ist, so ist es schwer, einen ternären Vektor aus L_H zu finden.
- Gegeben m, n und q, wobei $m \log q < n < \frac{q}{2 \cdot m^4}$ und $q = o(n^c)$ ($c > 0$). Es gibt eine Äquivalenz zwischen dem Auffinden einer Kollision für die Funktion f_H und der Lösung in Worst Case von shortest vector problem APPR**SVP**in einem Gitter mit dimension m

Satz 4.5

1. Sei ein Vektor $a \in \mathbb{R}^n$, sodass a kein binärer Vektor ist.
   f_H ist eine Einwegfunktion genau dann, wenn man einen Vektor x mit $x \in L_H$ und $x - a \in \mathfrak{B}_\mathfrak{n}$ findet.
2. f_H ist nicht Kollisionsresistenz genau dann, wenn man einen ternären Vektor $x \neq 0$ mit $x \in \mathcal{T}_n \cap L_H$ findet.

Beweis

1. Für einen gegebenen Vektor $b \in (\mathbb{Z}/q\mathbb{Z})^m$ sucht man ein y mit $f_H(y) = b$
   Betrachtet man die Gleichung $H \cdot z \equiv -b \mod q$, so kann man feststellen, dass diese Gleichung nach z lösbar ist (die Anzahl der Unbekannten ist grösser als die Anzahl der Gleichungen, da $n > m$ ist).
   Sei a eine Lösung, so gilt $H \cdot a \equiv -b \mod q$.
   Daraus folgt:

$$\begin{aligned} f_H(y) = b &\Longleftrightarrow \exists y \in \mathfrak{B}_\mathfrak{n} \text{ mit } H \cdot y = b \mod q \\ &\Longleftrightarrow H \cdot (y + a) = 0 \mod q \text{ (da } H \cdot a \equiv -b \mod q) \\ &\Longleftrightarrow \exists x \text{ mit } x - a \in \mathfrak{B}_\mathfrak{n} \text{ (setze } x = y + a) \end{aligned}$$

2. Angenommen f_H ist nicht Kollisionsresistenz, dann existieren zwei Elementen x und y aus $\mathfrak{B}_\mathfrak{n}$, mit $f_H(x) = f_H(y)$, und daraus folgt die Existenz eines ternären Vektors $x - y \in L_H$

■

Zusammengefaßt ist die Funktion f_H Kollisionsresistenz unter der Annahme, dass die folgenden Voraussetzungen erfüllt sind:
$m \log q < n < \frac{q}{2 \cdot m^4}$ und $q = o(n^c)$ ($c > 0$), wobei $q = 256$, $n = 2 \cdot m \cdot \log_2 q$.
Dauraus folgt, dass die LASH-Hashfunktion Kollisionsresistenz ist.

Kapitel 5

Ausblick

Hashfunktionen haben mit ihrer zunehmenden Anwendung mehr und mehr an Bedeutung gewonnen. Ihre Bedeutung ist aufgrund ihrer weiten Verbreitung zur Realisierung von Standards für Sicherheitslösungen im Bereich der Public-Key-Verschlüsselungen und digitalen Signaturen soweit angewachsen, dass man sagen kann, dass sie nicht mehr wegzudenken sind. Angesichts des Umstandes, dass es in den letzten Jahren zu einer dramatischen Zunahme von Angriffen auf kryptographische Hashfunktionen kam, ist die Notwendigkeit und damit das Interesse deutlich gestiegen, die Qualität der eingesetzten Hashfunktionen durch neue Entwicklungen und deren Implementierungen ausreichend schnell zu verbessern.
Die festgestellten Sicherheitsschwächen bei den Standard-Hashfunktionen MD4, MD5 und SHA-1 sind ernst und machten die Bedeutung der Gefahr eines Missbrauchs dieser Sicherheitsschwächen deutlich. Es konnte gezeigt werden, dass die häufig eingesetzte Hashfunktion MD5 innerhalb von wenigen Sekunden gebrochen werden konnte. Selbst der als sicher geltende Standard SHA-1 gilt als theoretisch angreifbar.
Meine Diplomarbeit hat als Schwerpunkt die Darstellung von Konstruktionen von solchen Hashfunktionen, die gegenüber den mit deutlichen Sicherheitsproblemen verbundenen weit verbreiteten Hashfunktionen auf der Basis des Ad-hoc-Design als sicherere Alternative gesehen werden. Die in Kapitel 3 beschriebenen modularen Konstruktionen von solchen Hashfunktionen, deren Sicherheit auf der Schwierigkeit der Lösung von harten zahlentheoretischen Problemen (der Faktorisierung oder des diskreter Logarithmus) basieren, haben zwar als Vorteil, dass sie als relativ sicher gelten, aber es wird die Gefahr gesehen, dass auch sie in der Zukunft als unsicher gelten könnten, wenn der technische Fortschritt in der Entwicklung von immer leistungsfähigeren Computern zu schnell fortschreiten würde (Man denke hierbei an Überlegungen der Konstruktion von Quantencomputern und deren möglicher Einsatz mit Hilfe des Shor-Algorithmus zum Lösen des Faktorisierungsproblems als Gefahr für die Sicherheit).
Als positive Eigenschaften weisen diese modularen Konstruktionen von Hashfunktionen auf, dass sie auf jedes erforderliche Sicherheitsniveau skalierbar sind und einfach einsetzbar sind in Systemen, in denen modulare Arithmetik bereits implementiert ist.
Als noch sicherer gelten die in Kapitel 4 dargestellten Konstruktionen von Hashfunktionen geometrischer oder algebraischen Struktur, deren Sicherheit auf der Schwierigkeit der Lösung von bestimmten Gitterproblemen (Shortest Vector Problem und Closest Vector Problem) basiert, die nach heutiger Ansicht voraussichtlich nicht mit Quantencomputern gelöst werden können.
Vorteilhaft bei diesen Konstruktionen geometrischer oder algebraischer Struktur ist ihre einfa-

che Konstruktion und damit ihre leichte Implementierung. Der entscheidende Nachteil der Hashfunktionen aus Kapitel 3 und 4 besteht darin, dass sie eine schlechte Performance haben und damit Schwierigkeiten haben, sich gegenüber solchen Hashfunktionen durchzusetzen, die auf Adhoc-Design basieren.
Es wird angenommen, dass Hashfunktionen auch weiterhin nur kurz- bzw. mittelfristig eine Sicherheit bieten können und dass eine langfristige Lösung angesichts des Fortschreitens der Entwicklung von Hardware und Software durch sie nicht gegeben werden kann. Trotzdem bleibt das Weiterentwickeln von noch sichereren Konstruktionen von Hashfunktionen auf der Basis von harten zahlentheoretischen bzw. geometrischen Problemen notwendig, um Schritt zu halten mit den Entwicklungen, die das Niveau der Sicherheit im Bereich der Kryptographie ständig schwächen.

Anhang A

Mathematische Grundlagen

Landau-Symbole
Die Landau-Symbole sind zum ersten Mal von Bachmann in 1892 angewendet, um die Größenordnung von Funktionen zu messen.

seien f und g zwei Funktionen, mit f, g: $\mathbb{N} \longrightarrow \mathbb{R}^+$.

- $g = O(f)$ bedeutet, daß die Funktion g höchstens so schnell wie f wächst.
  Formal heißt das: es existiert eine Konstant $c > 0$ und ein $n_0 \in N$, so daß
  $g(n) \leq c \cdot f(n)\ \forall n \geq n_0$.
- $g = \Omega(f)$ bedeutet, daß die Funktion g mindestens so schnell wie f wächst ($f = O(g)$).
  Formal heißt das: es existiert eine Konstant $c > 0$ und ein $n_0 \in N$, so daß
  $g(n) \geq c \cdot f(n)\ \forall n \geq n_0$.
- $g = \theta(f)$ bedeutet, daß die Funktionen f und g genauso so schnell wachsen.
  Formal heißt das: es existiert eine Konstant $c > 0$ und ein $n_0 \in N$, so daß
  $\frac{1}{c} \cdot f(n) \leq g(n) \leq c \cdot f(n)\ \forall n \geq n_0$.
- $g = \omega(f)$ bedeutet, daß die Funktion g schneller als f wächst.
  Formal heißt das: es existiert eine Konstant $c > 0$ und ein $n_0 \in N$, so daß
  $g(n) > c \cdot f(n)\ \forall n \geq n_0$.
- $g = o(f)$ bedeutet, daß die Funktion g langsamer als f wächst.
  Formal heißt das: es existiert eine Konstant $c > 0$ und ein $n_0 \in N$, so daß
  $g(n) < c \cdot f(n)\ \forall n \geq n_0$.
- $g = \tilde{O}(f)$ genau dann, wenn
  $\exists\ a$ und $c > 0$, so daß $g(n) < a \cdot f(n) \cdot \log^c f(n)$.

Typische Komplexitätsklassen
$O(1)$: konstanter Aufwand
$O(\log n)$: logarithmischer Aufwand
$O(n)$: linear Aufwand
$O(n^k)$ für $k \geq 0$: polynmialer Aufwand
$O(2^n)$: exponentieller Aufwand

Vernachlässigbare Funktion
Eine funktion ϵ: $\mathbb{N} \longrightarrow \mathbb{R}^+$ heißt vernachlässigbar („negligible“) und bezeichnet mit $\epsilon(n) = n^{-w(1)}$, wenn $\forall c > 0$, $\exists n_0 \in \mathbb{N}$ mit $|\epsilon(n)| < \frac{1}{n^c}$, $\forall n \geq n_0$.

Algebraische Struktur
Gruppe
Sei G eine Menge und $\circ$ eine Abbildung (Verknüpfung):$G \times G \longrightarrow G$.
G heißt eine *Gruppe* genau dann wenn:

1. $\circ$ assoziativ ist, d.h. $\forall x, y, z \in G : (x \circ y) \circ z = x \circ (y \circ z)$.
2. ein neutrales Element $e \in G$ existiert, d.h. $\forall x \in G : x \circ e = e \circ x = x$.
3. $\forall x \in G, \exists$ ein inverses x^{-1} mit $x \circ x^{-1} = x^{-1} \circ x = e$.

Falls nur die erste Voraussetzung erfüllt ist, heißt G in diesem Fall *Halbgruppe*.
Eine Gruppe $(G, \circ)$ heißt Kommutativ, wenn
$\forall x, y \in G : x \circ y = y \circ x$ gilt.

Gruppenisomorphismus
Zwei Gruppen $(G_1, \circ_1)$ und $(G_2, \circ_2)$ heißen isomorph (bezeichnet mit $G_1 \cong G_2$) genau dann wenn:
es eine bijektive Abbbildung $f : G_1 \longrightarrow G_2$ existiert, für die gilt:
$\forall x, y \in G_1 \ f(x \circ_1 y) = f(x) \circ_2 f(y)$.
Die Abbildung f heißt dann Gruppenisomorphismus.

Ring
seien R eine Menge und $\circ, \star$ zwei Verknüpfungen.
Das Tripel$(R, \circ, \star)$ heißt Ring genau dann wenn:

1. $(R, \circ)$ eine Kommutative Gruppe ist,
2. $(R, \star)$ eine Halbgruppe ist und
3. $\forall x, y, z \in R$ die Distributivgesetze gelten:
   $x \star (y \circ z) = (x \star y) \circ (x \star z)$ und $(x \circ y) \star z) = (x \star z) \circ (x \star z)$

Wenn jede Element aus dem Ring R ein Inverse bezüglich $\star$ besitzt, dann ist R ein Körper.

Ideal
Eine Teilmenge I eines Ringes $(R, +, \cdot)$ heißt ideal von R genau dann, wenn:

1. Die Null Element des Ringes liegt auch in I.
2. $\forall x, y \in I$, sind $x + y \in I$ und $x - y \in I$.
3. $\forall x \in I$ und $r \in R$, ist auch $r \cdot x \in I$.

Polynom
Sei R ein kommutativ Ring.
$P(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_1 \cdot x + a_0 = \sum_{k=1}^{n} a_k \cdot x^k$,wobei $a_i \in R$ heißt ein Polynom über R.
n wird den Grad von dem Polynom P genannt.
$R[x]$ bezeichnt die Menge aller Polynome über R.

Irreduzibles Polynom
Ein Polynom P heißt Irreduzibel über R, wenn man P nicht als Produkt von zwei Polynomen des Grades ≥ 1 darstellen kann.
Wenn $P(x)$ irreduzibel ist, dann ist der Ring $(R[x]/P(x), +, \cdot)$ ein Körper.

Vektorraum
Sei $(V, \circ)$ eine kommutative Gruppe und $(K, +, \cdot)$ ein körper mit 1 als Einselement.
Seien $x, y \in G$ und $\alpha, \beta \in K$.
Falls eine Verknüpfung $\star : K \times V \longrightarrow V$ mit den Eigenschaften:

1. $1 \star x = x$,
2. $\alpha \star (x \circ y) = \alpha \star x \circ \alpha \star y$
3. $(x \circ y) \star \alpha = \alpha \star x \circ \alpha \star y$,
4. $(\alpha \cdot \beta) \star x = \alpha \star (\beta \star x)$, existiert, dann heißt $(V, K, \star)$ ein Vektorraum über dem Körper K.

Die Elemente eines Vektorraumes heißen Vektoren.

Lineare Unabhängigkeit
Sei V ein Vektorraum über K sowie $\alpha_1, \ldots \alpha_n \in K$ und $x_1, \ldots, x_n$. Dann heißt

$$x = \alpha_1 \cdot x_1 + \ldots + \alpha_n \cdot x_n \text{ eine Linearkombination von } x_1, \ldots, x_n.$$

Die Vektoren $x_1, \ldots, x_n$ heißen linear unabhängig, wenn die Linearkombination
$\alpha_1 \cdot x_1 + \ldots + \alpha_n \cdot x_n = 0$ nür für $\alpha_1, \ldots, \alpha_n = 0$ sind.
Eine Menge $B = (b_1, \ldots, b_k)$ heißt eine Basis des Vektorraumes V, wenn die Vektoren $b_1, \ldots, b_k$ linear unabhängig sind und jedes Element aus V als Linearkombination der Vektoren darstellbar ist.
Die Anzahl der Vektoren von B heißt dimension von V und wird mit „dim“ bezeichnet.

Norm
Eine Norm auf einen Vektorraum V ist eine Abbildung $\|.\|$: $V \longrightarrow \mathbb{R}^+$ mit den folgenden Eigenschaften:

1. $\|x\| = 0 \Rightarrow x = 0$.
2. $\|\alpha \cdot x\| = \alpha \cdot \|x\|$
3. $\|x + y\| \leq \|x\| + \|y\|$, wobei $x, y, \in V$ und $\alpha \in K$

Die euklidische Norm ist wie folgt definiert:
$\|x\| := \sqrt{\sum_{i=1}^{n} x_i^2}$, $\forall x \in V$
Die Maximumsnorm ist folgendermaßen definiert:
$\|x\|_\infty = \max_{i=1}^{n} |x_i|$, wobei $x = (x_1, x_2, \dots x_n)$.

Diskrete Fourier-Transformation (DFT)
Seien f eine Funktion mit $f = (f_0, f_2 \dots, f_{n-1})$ und eine Matrix $M = (w_{kl})$ mit $w_{kl} = e^{\frac{2\pi \cdot i \cdot kl}{n}}$.
Man definert die diskrete Fourier Transformation (F) so folgendes:
$F = M \cdot f$, d.h. $F_k = \sum_{l=0}^{n-1} w_{kl} \cdot f^l$.

Fast Fourier-Transformation (FFT)
Die Fast Fourier Transformation ist ein Algorithmus zur schnellen Berechnung der Diskreten Fourier transformation. die Aufwand dieser Berechnung ist $(n \cdot \log n)$.

m Fast Fourier-Transformation (mFFT)
mFFT ist der Fast Fourier Transformation Algoritmus, wobei alle Berechnungen modular einer Zahl p ausgeführt sind.

Literaturverzeichnis

[1] Achim Jung, *The Strength of the CCITT/ISO Hash Function*, Arbeitspapiere der GMD, N 492, Dezember, 1994.

[2] Bart PRENEEL, *Analysis and Design of Cryptographic Hash Functions*, Phd thesis, February 2003, pp.162-163, http://homes.esat.kuleuven.be/~preneel/phd_preneel_feb1993.pdf.

[3] Bart PRENEEL *Analysis and Design of Cryptographic Hash Functions* Phd thesis, February 2003, pp.175-181 http://homes.esat.kuleuven.be/~preneel/phd_preneel_feb1993.pdf.

[4] Benny Chor and Ronald L Rivest. *A knapsack type public key cryptosystem based on arithmetic in finite fields*, Proceedings of CRYPTO 84 on Advances in cryptology, Springer-Verlag, pp.54-65, 1985.

[5] Buchmann, J, *Einführung in die Kryptographie*, Springer, 3. Auflage, 2003.

[6] CCITT: Recommendation X.509 Annex D. *The Directory-Authentication Framework*, Geneva 1989.

[7] Chris Peikert and Alon Rosen, *Efficient Collision-Resistant Hashing from Worst-Case Assumptions on Cyclic Lattices*, TCC, pp.145-166, 2006.

[8] Claudia Eckert, *IT-Sicherheit*, Oldenburg Verlag, 3.Auflage, 2004.

[9] Claus-Peter Schnorr, *FFT-hash,an efficient cryptographic hash function*, In Crypto Rump Session, 1991.

[10] Claus-Peter Schnorr, *FFT-Hash II, Efficient Cryptographic Hashing*, EUROCRYPT, Springer Verlag, pp 45–54, 1992.

[11] C.P. Schnorr and Serge Vaudenay, *Parallel FFT-hashing*, In Fast Software Encryption, pp 149–156, 1993.

[12] Daniele Micciancio, *Generalized compact knapsaks, cyclic lattices, and efficient one-way functions from worst-case complexity assumptions*, (FOCS 2002), pp. 356-365, 2002.

[13] D. Davies and W. L. Price, *The application of digital signatures based on public key cryptosystems*, NPL Report DNACS 39/80, Dezember, 1980.

[14] David Chaum, Eugène van Heijst, and Birgit Potzmann *strong undeniable signatures, unconditionally secure for the signer*, volume 576 of Lecture Notes in Computer Science, Springer-Verlag 1991, pp. 470-484.

[15] D. Wagner. *A generalized birthday problem*, In CRYPTO, pages 288-303, 2002.

[16] D. Davies and W. L. Price, *The application of digital signatures based on public key cryptosystems*, NPL Report DNACS 39/80, December 1980.

[17] Donghoon Chang, *Preimage Attack on Parallel FFT-Hashing*, Cryptology ePrint Archive, Report 2006/413, 2006.

[18] G.R. Blakley and I. Borosh,*Rivest-Shamir-Adleman public-key cryptosystems do not always conceal messages*, Comp and Maths with Applications, Vol. 5, pp.169-178, 1979.

[19] ISO/IEC JTC1/SC27 N804 (ISO/IEC 2nd WD 10118-4):Information Technology- Security Technique, -Hash-Functions,Part 4:*Hash-Functions using modular arithmetic*.94-01-19.

[20] ISO/IEC JTC1/SC27 N989 (ISO/IEC CD 10118-4): Information Technology- Security Techniques, Hash-Functions, *Hash-functions using modular arithmetic*, 94-12-21.

[21] Ivan Bjerre Damgård, *Collision Free Hash Functions and Public Key Signature Schemes*, In Eurocrypt 1987, Band 304 von Lecture Notes in Computer Science, Springer Verlag, pp.203-216, Berlin 1988.

[22] J.A. Gordon, *Strong RSA keys*, Electronic Letters, Vol. 20, No.12, pp.514-516, 1984.

[23] J.C. Pailles and M. Girault, *The security processor CRIPT*, 4th IFIP SEC,Monte-Carlo, pp.127–139, December 1986.

[24] K. Bentahar, D. Page, J.H. Silverman, M.-J. O. Saarinen and N.P. Smart *LASH*, 2nd NIST Cryptographic Hash Workshop, 2006.

[25] Lyubashevsky, Vadim and Micciancio, Daniele and Peikert, Chris and Rosen, Alon *Provably Secure FFT Hashing*, NIST 2nd Cryptogaphic Hash Workshop, August 2006.

[26] M. Girault, *Hash-functions using modulo-n operations*, Advances in Cryptology,Proc. Eurocrypt 87, LNCS 304, D. Chaum and W.L. Price, Eds., Springer-Verlag, 1988, pp.217–226.

[27] Marc Girault, *Hash-Functions using Modulo-N Operations* Advances in Cryptology-Eurocrypt 87, Lecture Notes in Computer Science, V.304, pp.217-226, Springer Verlag 1987.

[28] M.Ajtai,*Generating hard instances of lattice problems (extended abstract)* STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing, pp.99-108, 1996.

[29] M.J. Coster, B.A. LaMacchia, A.M. Odlyzko und C.P. Schnorr, *An Improved Low-Density Subset Sum Algorithm*, volume 547 of LNCS, pp.54-67, Springer-Verlag, 1991.

[30] O. Goldreich and D. Micciancio and S. Safra and J. -P. Seifert *Approximating shortest lattice vectors is not harder than approximating closet lattice vectors*,Inf. Process. Lett. volume 71.pp 55-61, 1999.

[31] Oded Goldreich, Shafi Goldwasser, and Shai Halevi, *Collision-Free Hashing from Lattice Problems*, Cryptology ePrint Archive, Report 1996/009, 1996

[32] Robert R. Jueneman, S.M. Matyas, and C.H. Meyer. *Message authentication with Manipulation Detection Codes*, Proc. 1983 IEEE Symposium on Security and Privacy, pp.33-54, 1984.

[33] Robert R. Jueneman, *Analysis of certain aspects of Output Feedback Mode*, Advances in Cryptology, Proc. Crypto 82, D. Chaum, R.L. Rivest, and A.T. Sherman,Eds, Plenum Press, pp. 99-127, 1983.

[34] R.R. Jueneman, S.M. Matyas, and C.H. Meyer, *Message authentication*, IEEE Communications Mag. Vol.23, No.9, pp.29-40, 1985.

[35] Scott Contini, Arjen K. Lenstra, Ron Steinfeld, *VSH, an Efficient and Provable Collision-Resistant Hash Function*, pp.165-182, EUROCRYPT 2006.

[36] S.F. Mjølsnes, *A hash of some one-way hash functions and birthdays*, preprint, 1989.

[37] S. Miyaguchi, K. Ohta and M. Iwata, *128-bit hash function(N-hash)*, NTT Review, 2, pp.117–127, 1990.

[38] T.Okamoto, K. Tanaka und S.Uchiyama,*Quantum Public Key Cryptosystems*, Proc. Of CRYPTO 2000, volume.1880 of LNCS, pp.147-165, Springer-Verlag, 2000.