

Vorlesungsskript  
Kryptologie  
Wintersemester 2022/23

Prof. Dr. Johannes Köbler  
Humboldt-Universität zu Berlin  
Lehrstuhl Komplexität und Kryptografie

27. Oktober 2022

# Inhaltsverzeichnis

<b>1</b>	<b>Kryptografische Hashverfahren</b>	<b>1</b>
1.1	Einführung . . . . .	1
1.2	Schlüssellose Hashfunktionen (MDCs) . . . . .	3
1.2.1	Vergleich von Sicherheitsanforderungen . . . . .	4
1.2.2	Das Zufallsorakelmodell (ZOM) . . . . .	5
1.2.3	Iterierte Hashfunktionen . . . . .	8
1.2.4	Die Merkle-Damgaard-Konstruktion . . . . .	9
1.2.5	Die MD4-Hashfunktion . . . . .	10
1.2.6	Die MD5-Hashfunktion . . . . .	11
1.2.7	Die SHA-1-Hashfunktion . . . . .	12

# 1 Kryptografische Hashverfahren

## 1.1 Einführung

Durch kryptographische Verfahren lassen sich unter anderem die folgenden **Schutzziele** realisieren.

- *Vertraulichkeit*
  - Geheimhaltung
  - Anonymität (z.B. Mobiltelefon)
  - Unbeobachtbarkeit (von Transaktionen)
- *Integrität*
  - von Nachrichten und Daten
- *Zurechenbarkeit*
  - Authentikation
  - Unabstreitbarkeit
  - Identifizierung
- *Verfügbarkeit*
  - von Daten
  - von Rechenressourcen
  - von Informationsdienstleistungen

Kryptografische Hashverfahren sind ein wirksames Werkzeug zur Sicherstellung der Integrität von Nachrichten oder generell von digitalisierten Daten. Sie nehmen somit beim Schutz der Datenintegrität eine ähnlich herausragende Stellung ein wie sie Kryptosystemen bei der Wahrung der Vertraulichkeit zukommt. Daneben finden kryptografische Hashfunktionen aber auch vielfach als Bausteine von komplexeren Systemen Verwendung. Wie wir noch sehen werden, sind kryptografische Hashfunktionen etwa bei der Erstellung von digitalen Signaturen sehr nützlich. Auf weitere Anwendungsmöglichkeiten werden wir später eingehen.

Vielen Anwendungen von kryptografischen Hashfunktionen  $h$  liegt die Idee zugrunde, dass sie zu einem vorgegebenen Text  $x$  eine zwar kompakte aber dennoch repräsentative Darstellung  $h(x)$  liefern, die unter praktischen Gesichtspunkten als eine eindeutige Identifikationsnummer von  $x$  fungieren kann. Die Berechnungsvorschrift für  $h$  muss somit „charakteristische Merkmale“ von  $x$  in den Hashwert  $h(x)$  einfließen lassen. Da der Fingerabdruck eines Menschen ganz ähnliche Eigenschaften besitzt (was ihn für Kriminalisten bekanntlich so wertvoll macht), wird der Hashwert  $h(x)$  auch oft als ein **digitaler Fingerabdruck** von  $x$  bezeichnet. Gebräuchlich sind auch die Bezeichnungen **kryptografische Prüfsumme** oder *message digest* (englische Bezeichnung für „Nachrichtenextrakt“).

Typische Schutzziele, die sich mittels Hashfunktionen realisieren lassen, sind die Nachrichten- und Teilnehmerauthentikation.

- „Nachrichtenauthentikation“ (message authentication)

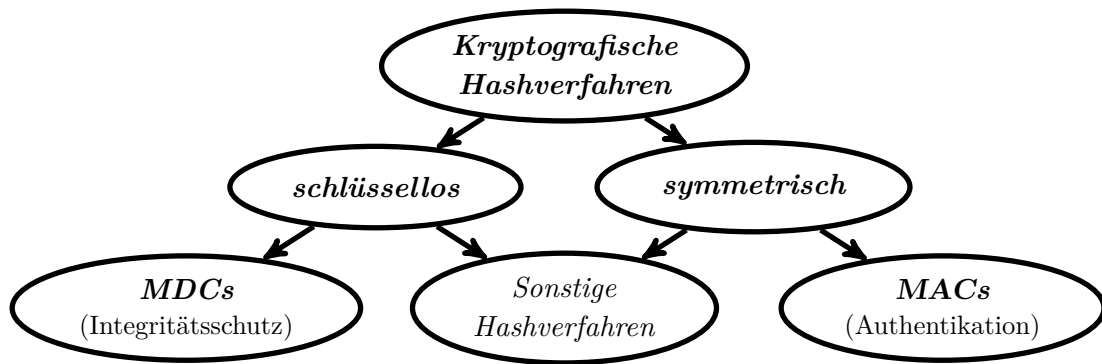


Abbildung 1.1: Eine grobe Einteilung von kryptografischen Hashverfahren.

- Wie lässt sich sicherstellen, dass eine Nachricht (oder eine Datei) während einer (räumlichen oder auch zeitlichen) Übertragung nicht verändert wurde?
- Wie lässt sich der Urheber (oder Absender) einer Nachricht zweifelsfrei feststellen?
- „Teilnehmerauthentikation“ (entity authentication, identification)
  - Wie kann sich eine Person (oder ein Gerät) anderen gegenüber zweifelsfrei ausweisen?

## Klassifikation von Hashverfahren

Kryptografische Hashverfahren lassen sich grob danach klassifizieren, ob der Hashwert lediglich in Abhängigkeit vom Eingabetext berechnet wird oder zusätzlich von einem symmetrischen Schlüssel abhängt (siehe Abbildung 1.1).

Kryptografische Hashfunktionen, bei deren Berechnung keine Schlüssel benutzt werden, dienen vornehmlich der Erkennung von unbefugt vorgenommenen Manipulationen an Dateien oder Nachrichten. Daher werden sie auch als **MDC** bezeichnet (**Manipulation Detection Code** [englisch] = Code zur Erkennung von Manipulationen). Zuweilen wird das Kürzel **MDC** auch als eine Abkürzung für **Modification Detection Code** verwendet. Seltener ist dagegen die Bezeichnung **MIC** (**message integrity codes**). Abbildung 1.2 zeigt eine typische Anwendung von MDCs.

Um die Integrität eines Datensatzes  $x$  sicherzustellen, der über einen ungesi-

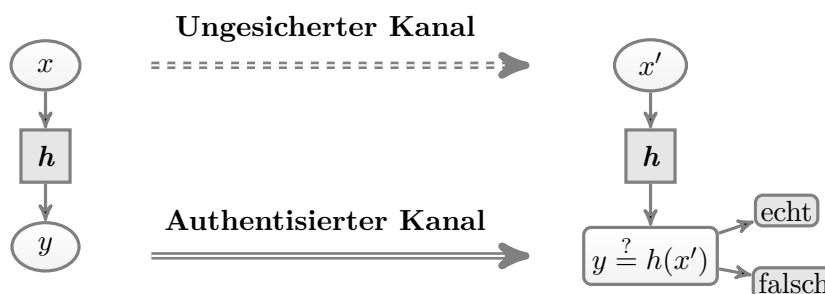


Abbildung 1.2: Einsatz eines MDC  $h$  zur Überprüfung der Integrität eines Datensatzes  $x$ .

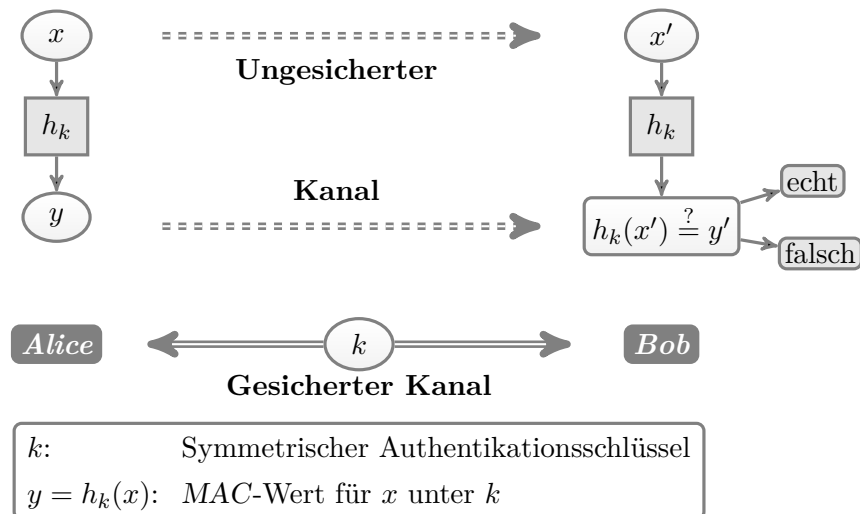


Abbildung 1.3: Verwendung eines MAC zur Nachrichtenauthentikation.

cherten Kanal gesendet (bzw. auf einem vor Manipulationen nicht sicheren Webserver abgelegt) wird, kann man wie folgt verfahren. Man sendet den MDC-Hashwert von  $x$  über einen authentisierten Kanal und prüft, ob der Datensatz nach der Übertragung noch denselben Hashwert liefert.

Kryptografische Hashverfahren mit symmetrischen Schlüsseln finden hauptsächlich bei der Authentifizierung von Nachrichten Verwendung. Diese werden daher auch als **MAC** (**m**essage **a**uthentication **c**ode [englisch] = Code zur Nachrichtenauthentifizierung) oder als **Authentifikationscode** bezeichnet. Daneben gibt es auch Hashverfahren mit asymmetrischen Schlüsseln. Diese werden jedoch der Rubrik der Signaturverfahren zugeordnet, da mit ihnen ausschließlich digitale Signaturen gebildet werden. Abbildung 1.3 zeigt, wie sich Nachrichten mit einem MAC authentisieren lassen. Man beachte, dass nun auch der Hashwert über den unsicheren Kanal gesendet wird.

Möchte Alice eine Nachricht  $x$  an Bob übermitteln, so berechnet sie den zugehörigen MAC-Wert  $y = h_k(x)$  und fügt diesen der Nachricht  $x$  hinzu. Bob überprüft die Echtheit der empfangenen Nachricht  $(x', y')$ , indem er seinerseits den zu  $x'$  gehörigen Hashwert  $h_k(x')$  berechnet und das Ergebnis mit  $y'$  vergleicht. Der geheime Authentifikationsschlüssel  $k$  muss hierbei genau wie bei einem symmetrischen Kryptosystem über einen gesicherten Kanal vereinbart werden.

Indem Alice ihre Nachricht  $x$  um den Hashwert  $y = h_k(x)$  ergänzt, gibt sie Bob nicht nur die Möglichkeit, anhand von  $y$  die empfangene Nachricht auf Manipulationen zu überprüfen. Die Benutzung des geheimen Schlüssels  $k$  erlaubt zudem eine Überprüfung der Herkunft der Nachricht.

## 1.2 Schlüssellose Hashfunktionen (MDCs)

In diesem Abschnitt betrachten wir verschiedene Sicherheitsanforderungen an einzelne Hashfunktionen  $h$ . Dabei nehmen wir an, dass  $h$  öffentlich bekannt ist, d.h.  $h$  ist eine schlüssellose Hashfunktion (MDC).

Sei  $h: X \rightarrow Y$  eine Hashfunktion. Ein Paar  $(x, y) \in X \times Y$  heißt **gültig** für  $h$ , falls  $h(x) = y$  ist. Ein Paar  $(x, x')$  mit  $h(x) = h(x')$  heißt **Kollisionspaar** für  $h$ . Die Anzahl  $\|Y\|$  der Hashwerte bezeichnen wir mit  $m$ . Ist auch der Textraum  $X$  endlich,  $\|X\| = n$ , so heißt  $h$  eine  $(n, m)$ -**Hashfunktion**. In diesem Fall verlangen wir meist, dass  $n \geq 2m$  ist, und wir nennen  $h$  dann eine **Kompressionsfunktion** (*compression function*).

Da  $h$  öffentlich bekannt ist, ist es sehr einfach, für einen vorgegebenen Text  $x$  ein gültiges Paar  $(x, y)$  zu erzeugen. Für bestimmte kryptografische Anwendungen ist es wichtig, dass dies nicht möglich ist, falls der Hashwert  $y$  vorgegeben wird.

### Problem P1: Bestimmung eines Urbilds

**Gegeben:** Eine Hashfkt.  $h: X \rightarrow Y$  und ein Hashwert  $y \in Y$ .

**Gesucht:** Ein Text  $x \in X$  mit  $h(x) = y$ .

Falls es einen immensen Aufwand erfordert, für einen *vorgegebenen* Hashwert  $y$  einen Text  $x$  mit  $h(x) = y$  zu finden, so heißt  $h$  **Einweg-Hashfunktion** (*one-way hash function* bzw. *preimage resistant hash function*). Diese Eigenschaft wird beispielsweise benötigt, wenn die Hashwerte der Benutzerpasswörter in einer öffentlich zugänglichen Datei abgespeichert werden, wie es bei manchen Unix-Systemen der Fall ist.

### Problem P2: Bestimmung eines zweiten Urbilds

**Gegeben:** Eine Hashfkt.  $h: X \rightarrow Y$  und ein Text  $x \in X$ .

**Gesucht:** Ein Text  $x' \in X \setminus \{x\}$  mit  $h(x') = h(x)$ .

Falls sich für einen *vorgegebenen* Text  $x$  nur mit großem Aufwand ein weiterer Text  $x' \neq x$  mit dem gleichen Hashwert  $h(x') = h(x)$  finden lässt, heißt  $h$  **schwach kollisionsresistent** (*weakly collision resistant* bzw. *second preimage resistant*). Diese Eigenschaft wird in der durch Abbildung 1.2 skizzierten Anwendung benötigt. Beim Versuch, eine digitale Signatur zu fälschen (siehe unten), sieht sich der Angreifer dagegen mit folgender Problemstellung konfrontiert.

### Problem P3: Bestimmung einer Kollision

**Gegeben:** Eine Hashfkt.  $h: X \rightarrow Y$ .

**Gesucht:** Texte  $x \neq x' \in X$  mit  $h(x) = h(x')$ .

Falls sich dieses Problem nur mit einem immensen Aufwand lösen lässt, heißt  $h$  (**stark**) **kollisionsresistent** (*collision resistant*).

Obwohl die schwache Kollisionsresistenz eine gewisse Ähnlichkeit mit der Einweg-Eigenschaft aufweist, sind die beiden Eigenschaften im allgemeinen unvergleichbar. So muss eine schwach kollisionsresistente Funktion nicht notwendigerweise eine Einwegfunktion sein, da die Bestimmung eines Urbildes gerade für diejenigen Funktionswerte einfach sein kann, die nur ein einziges Urbild besitzen. Umgekehrt impliziert die Einweg-Eigenschaft auch nicht die schwache Kollisionsresistenz, da die Kenntnis eines Urbildes das Auffinden weiterer Urbilder sehr stark erleichtern kann.

#### 1.2.1 Vergleich von Sicherheitsanforderungen

In diesem Abschnitt zeigen wir, dass stark kollisionsresistente Hashfunktionen sowohl schwach kollisionsresistent als auch Einweghashfunktionen sind.

**Satz 1.** Sei  $h: X \rightarrow Y$  eine  $(n, m)$ -Hashfunktion. Dann ist das Problem P3, ein Kollisionspaar für  $h$  zu bestimmen, auf das Problem P2, ein zweites Urbild zu bestimmen,

---

```

1 wähle zufällig  $x \in X$ 
2  $x' := A(x)$ 
3 if  $x' \neq ?$  then return( $x, x'$ ) else return(?)

```

---

Abbildung 1.4: Reduktion des Kollisionsproblems auf das Problem, ein zweites Urbild zu bestimmen

*reduzierbar. Folglich sind stark kollisionsresistente Hashfunktionen auch schwach kollisionsresistent.*

*Beweis.* Sei  $A$  ein Las-Vegas Algorithmus, der für ein zufällig aus  $X$  gewähltes  $x$  mit Erfolgswahrscheinlichkeit  $\varepsilon$  ein zweites Urbild  $x'$  für  $h$  liefert und andernfalls  $?$  ausgibt. Dann ist klar, dass der in Abbildung 1.4 dargestellte Las-Vegas Algorithmus mit Wahrscheinlichkeit  $\varepsilon$  ein Kollisionspaar findet.  $\square$

Als nächstes zeigen wir, wie sich das Kollisionsproblem auf das Urbildproblem reduzieren lässt.

**Satz 2.** Sei  $h: X \rightarrow Y$  eine  $(n, m)$ -Hashfunktion mit  $n \geq 2m$ . Dann ist das Problem P3, ein Kollisionspaar für  $h$  zu bestimmen, auf das Problem P1, ein Urbild zu bestimmen, reduzierbar.

*Beweis.* Sei  $A$  ein Invertierungsalgorithmus für  $h$ , d.h.  $A$  berechnet für jeden Hashwert  $y$  in  $W(h) = \{h(x) \mid x \in X\}$  ein Urbild  $x$  mit  $h(x) = y$ . Betrachte den in Abbildung 1.5 dargestellten Las-Vegas Algorithmus  $B$ .

Sei  $\mathcal{C} = \{h^{-1}(y) \mid y \in W(h)\}$ . Dann hat  $B$  eine Erfolgswahrscheinlichkeit von

$$\sum_{C \in \mathcal{C}} \frac{\|C\|}{\|X\|} \cdot \frac{\|C\| - 1}{\|C\|} = \frac{1}{n} \sum_{C \in \mathcal{C}} (\|C\| - 1) = (n - m)/n \geq \frac{1}{2}.$$

$\square$

### 1.2.2 Das Zufallsorakelmodell (ZOM)

Das ZOM dient dazu, den Aufwand verschiedener Angriffe auf eine Hashfunktion  $h: X \rightarrow Y$  nach oben abzuschätzen. Sind  $X$  und  $Y$  vorgegeben, so können wir eine Hashfunktion  $h: X \rightarrow Y$  dadurch „konstruieren“, dass wir für jedes  $x \in X$  zufällig ein  $y \in Y$  wählen und  $h(x)$  auf  $y$  setzen. Äquivalent hierzu ist, für  $h$  eine zufällige Funktion aus der Klasse  $F(X, Y)$  aller  $m^n$  Funktionen von  $X$  nach  $Y$  zu wählen. Dieses Verfahren ist auf Grund des hohen Aufwands zwar nicht mehr praktikabel, wenn  $n = \|X\|$  eine bestimmte Größe übersteigt. Es liefert uns aber ein theoretisches Modell für eine Hashfunktion mit „idealen“ kryptografischen Eigenschaften. Offensichtlich besteht für den Angreifer die

---

```

1 wähle zufällig  $x \in X$ 
2  $y := h(x)$ 
3  $x' := A(y)$ 
4 if  $x \neq x'$  then return( $x, x'$ ) else return(?)

```

---

Abbildung 1.5: Reduktion des Kollisionsproblems auf das Urbildproblem

**Prozedur FindPreimage( $h, y, q$ )**


---

```

1 wähle eine beliebige Menge  $X_0 = \{x_1, \dots, x_q\} \subseteq X$ 
2 for each  $x_i \in X_0$  do
3   if  $h(x_i) = y$  then return( $x_i$ )
4 return(?)

```

---

Abbildung 1.6: Bestimmung eines Urbilds für einen Hashwert

einzigste Möglichkeit, Informationen über  $h$  zu erhalten, darin, sich für eine Reihe von Texten die zugehörigen Hashwerte zu besorgen (was der Befragung eines funktionalen Zufallsorakels entspricht).

Eine Zufallsfunktion  $h$  eignet sich deshalb gut als kryptografische Hashfunktion, weil der Hashwert  $h(x)$  für einen Text  $x$  auch dann noch schwer vorhersagbar ist, wenn der Angreifer bereits die Hashwerte einer beliebigen Zahl von anderen Texten kennt.

**Proposition 3.** Sei  $X_0 = \{x_1, \dots, x_k\}$  eine beliebige Menge von  $k$  verschiedenen Texten aus  $X$  und seien  $y_1, \dots, y_k \in Y$ . Dann gilt für eine zufällig aus  $F(X, Y)$  gewählte Funktion  $h$  und für jedes Paar  $(x, y) \in (X - X_0) \times Y$ ,

$$\Pr[h(x) = y \mid h(x_i) = y_i \text{ für } i = 1, \dots, k] = 1/m.$$

Um eine obere Komplexitätsschranke für das Urbildproblem im ZOM zu erhalten, betrachten wir den in Abbildung 1.6 dargestellten Algorithmus. Hier (und bei den beiden folgenden Algorithmen) gibt der Parameter  $q$  die Anzahl der Hashwertberechnungen (also die Anzahl der gestellten Orakelfragen an das Zufallsorakel  $h$ ) an. Der Zeitaufwand der Algorithmen ist dabei proportional zu  $q$ .

**Satz 4.**  $\text{FINDPREIMAGE}(h, y, q)$  gibt im ZOM mit Wahrscheinlichkeit  $\varepsilon = 1 - (1 - 1/m)^q$  ein Urbild von  $y$  aus (unabhängig von der Wahl der Menge  $X_0$ ).

*Beweis.* Sei  $y \in Y$  fest und sei  $X_0 = \{x_1, \dots, x_q\}$ . Für  $i = 1, \dots, q$  bezeichne  $E_i$  das Ereignis “ $h(x_i) = y$ ”. Nach Proposition 3 sind diese Ereignisse stochastisch unabhängig und ihre Wahrscheinlichkeit ist  $\Pr[E_i] = 1/m$  ( $i = 1, \dots, q$ ). Also folgt

$$\Pr[E_1 \cup \dots \cup E_q] = 1 - \Pr[\overline{E_1} \cap \dots \cap \overline{E_q}] = 1 - (1 - 1/m)^q.$$

□

**Prozedur FindSecondPreimage( $h, x, q$ )**


---

```

1  $y := h(x)$ 
2 wähle eine beliebige Menge  $X_0 = \{x_1, \dots, x_{q-1}\} \subseteq X - \{x\}$ 
3 for each  $x_i \in X_0$  do
4   if  $h(x_i) = y$  then return( $x_i$ )
5 return(?)

```

---

Abbildung 1.7: Bestimmung eines 2. Urbilds für einen Hashwert



**Prozedur** Collision( $h, q$ )

---

```

1 wähle eine beliebige Menge  $X_0 = \{x_1, \dots, x_q\} \subseteq X$ 
2 for each  $x_i \in X_0$  do  $y_i := h(x_i)$ 
3 if  $\exists i \neq j : y_i = y_j$  then return( $x_i, x_j$ ) else return(?)

```

---

Abbildung 1.8: Bestimmung eines Kollisionspaares

Der in Abbildung 1.7 dargestellte Algorithmus liefert uns eine obere Schranke für die Komplexität des Problems, ein zweites Urbild für  $h(x)$  zu bestimmen. Die Erfolgswahrscheinlichkeit lässt sich vollkommen analog zum vorherigen Satz bestimmen.

**Satz 5.** *FINDSECONDPREIMAGE( $h, x, q$ ) gibt im ZOM mit Wahrscheinlichkeit  $\varepsilon = 1 - (1 - 1/m)^{q-1}$  ein zweites Urbild  $x_0 \neq x$  von  $y = h(x)$  aus.*

Ist  $q$  vergleichsweise klein, so ist bei beiden bisher betrachteten Angriffen  $\varepsilon \approx q/m$ . Um also auf eine Erfolgswahrscheinlichkeit von  $1/2$  zu kommen, ist  $q \approx m/2$  zu wählen.

Geht es lediglich darum, irgendein Kollisionspaar  $(x, x')$  aufzuspüren, so bietet sich ein sogenannter **Geburtstagsangriff** an. Dieser ist deutlich zeiteffizienter zu realisieren. Wie der Name schon andeutet, basiert dieser Angriff auf dem sogenannten Geburtstagsparadoxon, welches in seiner einfachsten Form folgendes besagt.

**Geburtstagsparadoxon:** Bereits in einer Schulklasse mit 23 Schulkindern haben mit einer Wahrscheinlichkeit größer  $1/2$  mindestens zwei Kinder am gleichen Tag Geburtstag.\*

Tatsächlich zeigt der nächste Satz, dass bei  $q$ -maligem Ziehen (mit Zurücklegen) aus einer Urne mit  $m$  Kugeln mit einer Wahrscheinlichkeit von

$$1 - (m-1)(m-2) \cdots (m-q+1)/m^{q-1}$$

mindestens eine Kugel mehr als einmal gezogen wird. Für  $m = 365$  und  $q = 23$  ergibt dies einen Wert von ungefähr 0,507.

Zur Kollisionsbestimmung verwenden wir den in Abbildung 1.8 dargestellten Algorithmus. Bei einer naiven Implementierung würde zwar der Zeitaufwand für die Auswertung der if-Bedingung quadratisch von  $q$  abhängen. Trägt man aber jeden Text  $x$  unter dem Suchwort  $h(x)$  in eine (herkömmliche) Hashtabelle der Größe  $q$  ein, so wird der Zeitaufwand für die Bearbeitung jedes einzelnen Textes  $x$  im wesentlichen durch die Berechnung von  $h(x)$  bestimmt.

**Satz 6.** *COLLISION( $h, q$ ) gibt im ZOM mit Erfolgswahrscheinlichkeit*

$$\varepsilon = 1 - \frac{(m-1)(m-2) \cdots (m-q+1)}{m^{q-1}}$$

*ein Kollisionspaar  $(x, x')$  für  $h$  aus.*

*Beweis.* Sei  $X_0 = \{x_1, \dots, x_q\}$ . Für  $i = 1, \dots, q$  bezeichne  $E_i$  das Ereignis

$$“h(x_i) \notin \{h(x_1), \dots, h(x_{i-1})\}.”$$

---

\*Da die Häufigkeiten der Geburtstage in Wirklichkeit nicht ganz gleichmäßig über das Jahr verteilt sind, ist die Wahrscheinlichkeit sogar noch etwas höher.

Dann beschreibt  $E_1 \cap \dots \cap E_q$  das Ereignis “COLLISION( $h, q$ ) gibt ? aus” und für  $i = 1, \dots, q$  gilt

$$\Pr[E_i | E_1 \cap \dots \cap E_{i-1}] = \frac{m - i + 1}{m}.$$

Dies führt auf die Erfolgswahrscheinlichkeit

$$\begin{aligned} \varepsilon &= 1 - \Pr[E_1 \cap \dots \cap E_q] \\ &= 1 - \Pr[E_1] \Pr[E_2 | E_1] \dots \Pr[E_q | E_1 \cap \dots \cap E_{q-1}] \\ &= 1 - \left(\frac{m-1}{m}\right) \left(\frac{m-2}{m}\right) \dots \left(\frac{m-q+1}{m}\right). \end{aligned}$$

□

Mit  $1 - x \approx e^{-x}$  folgt

$$\varepsilon = 1 - \prod_{i=1}^{q-1} \left(1 - \frac{i}{m}\right) \approx 1 - \prod_{i=1}^{q-1} e^{-\frac{i}{m}} = 1 - e^{-\frac{1}{m} \sum_{i=1}^{q-1} i} = 1 - e^{-\frac{q(q-1)}{2m}} \approx 1 - e^{-\frac{q^2}{2m}} \approx q^2/2m.$$

Somit erhalten wir die Abschätzung

$$q \approx c_\varepsilon \sqrt{m}$$

mit  $c_\varepsilon = \sqrt{2\varepsilon}$ . Diese Abschätzung ist nur für  $\varepsilon$ -Werte nahe Null hinreichend genau. Eine bessere Abschätzung ergibt sich aus der Approximation  $\varepsilon \approx 1 - e^{-\frac{q^2}{2m}}$ :

$$q \approx c'_\varepsilon \sqrt{m}$$

mit  $c'_\varepsilon = \sqrt{2 \ln \frac{1}{1-\varepsilon}}$ . Für  $\varepsilon = 1/2$  ergibt sich somit  $q \approx \sqrt{(2 \ln 2)m} \approx 1,17\sqrt{m}$ .

Besitzt also eine binäre Hashfunktion  $h: \{0, 1\}^n \rightarrow \{0, 1\}^m$  die Hashwertlänge  $m = 128$  Bit, so müssen im ZOM  $q \approx 1,17 \cdot 2^{64}$  Texte gehasht werden, um mit einer Wahrscheinlichkeit von  $1/2$  eine Kollision zu finden. Um einem Geburtstagsangriff widerstehen zu können, sollte eine Hashfunktion mindestens eine Hashwertlänge von 128 oder besser 160 Bit haben.

### 1.2.3 Iterierte Hashfunktionen

In diesem Abschnitt beschäftigen wir uns mit der Frage, wie sich aus einer kollisionsresistenten Kompressionsfunktion

$$h: \{0, 1\}^{m+t} \rightarrow \{0, 1\}^m$$

eine kollisionsresistente Hashfunktion

$$\hat{h}: \{0, 1\}^* \rightarrow \{0, 1\}^l$$

konstruieren lässt. Hierzu betrachten wir folgende kanonische Konstruktionsmethode.

**Preprocessing:** Transformiere  $x \in \{0, 1\}^*$  mittels einer Funktion

$$y: \{0, 1\}^* \rightarrow \bigcup_{r \geq 1} \{0, 1\}^{rt}$$

zu einem String  $y(x)$  mit der Eigenschaft  $|y(x)| \equiv_t 0$ .

**Processing:** Sei  $IV \in \{0, 1\}^m$  ein öffentlich bekannter Initialisierungsvektor und sei  $y(x) = y_1 \cdots y_r$  mit  $|y_i| = t$  für  $i = 1, \dots, r$ . Berechne eine Folge  $z_0, \dots, z_r$  von Strings  $z_i \in \{0, 1\}^m$  wie folgt:

$$z_i = \begin{cases} IV, & i = 0, \\ h(z_{i-1}y_i), & i = 1, \dots, r. \end{cases}$$

**Optionale Ausgabetransformation:** Berechne den Hashwert  $\hat{h}(x) = g(z_r)$ , wobei  $g: \{0, 1\}^m \rightarrow \{0, 1\}^l$  eine öffentlich bekannte Funktion ist. (Meist wird für  $g$  die Identität verwendet.)

Um  $\hat{h}(x)$  zu berechnen, muss also die Kompressionsfunktion  $h$  genau  $r$ -mal aufgerufen werden. Wir formulieren nun eine für Preprocessing-Funktionen wünschenswerte Eigenschaft.

**Definition 7.** Eine Funktion  $y: \{0, 1\}^* \rightarrow \{0, 1\}^*$  heißt **suffixfrei**, falls es keine Strings  $x \neq \tilde{x}$  und  $z$  in  $\{0, 1\}^*$  mit  $y(\tilde{x}) = zy(x)$  gibt (d.h. kein Funktionswert  $y(x)$  ist Suffix eines Funktionswertes  $y(\tilde{x})$  an einer Stelle  $\tilde{x} \neq x$ ).

Man beachte, dass jede suffixfreie Funktion insbesondere injektiv ist.

**Satz 8.** Falls die Preprocessing-Funktion  $y$  suffixfrei und die Ausgabetransformation  $g$  injektiv ist, so ist mit  $h$  auch  $\hat{h}$  kollisionsresistent.

*Beweis.* Wir nehmen an, dass es gelingt, ein Kollisionspaar  $(x, \tilde{x})$  für  $\hat{h}$  zu finden (d.h.  $\hat{h}(x) = \hat{h}(\tilde{x})$  und  $x \neq \tilde{x}$ ). Sei

$$y(x) = y_1y_2 \dots y_{k-1}y_k \text{ und } y(\tilde{x}) = \tilde{y}_1\tilde{y}_2 \dots \tilde{y}_{l-1}\tilde{y}_l \text{ mit } k \leq l.$$

Da  $y$  suffixfrei ist, muss ein Index  $i \in \{1, \dots, k\}$  mit  $y_i \neq \tilde{y}_{l-k+i}$  existieren. Weiter seien  $z_i$  ( $i = 0, \dots, k$ ) und  $\tilde{z}_j$  ( $j = 0, \dots, l$ ) die in der Processing-Phase berechneten Hashwerte. Da  $g$  injektiv ist, muss mit  $g(z_k) = \hat{h}(x) = \hat{h}(\tilde{x}) = g(\tilde{z}_l)$  auch  $z_k = \tilde{z}_l$  gelten. Sei  $i_{\max}$  der größte Index  $i \in \{1, \dots, k\}$  mit  $z_{i-1}y_i \neq \tilde{z}_{l-k+i-1}\tilde{y}_{l-k+i}$ . Dann bilden  $z_{i_{\max}-1}y_{i_{\max}}$  und  $\tilde{z}_{l-k+i_{\max}-1}\tilde{y}_{l-k+i_{\max}}$  wegen

$$h(z_{i_{\max}-1}y_{i_{\max}}) = z_{i_{\max}} = \tilde{z}_{l-k+i_{\max}} = h(\tilde{z}_{l-k+i_{\max}-1}\tilde{y}_{l-k+i_{\max}})$$

ein Kollisionspaar für  $h$ . □

### 1.2.4 Die Merkle-Damgaard-Konstruktion

Merkle und Damgaard schlugen 1989 folgende konkrete Realisierung ihrer Konstruktion vor. Als Initialisierungsvektors wird der Nullvektor  $IV = 0^m$  benutzt, die optionale Ausgabetransformation entfällt, und für  $y(x)$  wird im Fall  $t \geq 2$  die folgende Funktion verwendet. (Den Fall  $t = 1$  betrachten wir später.)

Für  $x = \varepsilon$  sei  $y(x) = 0^t$  und für  $x \in \{0, 1\}^n$  mit  $n > 0$  sei  $k = \lceil \frac{n}{t-1} \rceil$  und  $x = x_1x_2 \dots x_{k-1}x_k$  mit  $|x_1| = |x_2| = \dots = |x_{k-1}| = t-1$  sowie  $|x_k| = t-1-d$ , wobei

$0 \leq d < t - 1$ . Im Fall  $k = 1$  ist dann  $y(x) = 0x0^d 1 \text{bin}_{t-1}(d)$  und für  $k > 1$  ist  $y(x) = y_1 \cdots y_{k+1}$ , wobei

$$y_i = \begin{cases} 0x_1, & i = 1, \\ 1x_i, & 2 \leq i < k, \\ 1x_k 0^d, & i = k, \\ 1 \text{bin}_{t-1}(d), & i = k + 1, \end{cases} \quad (1.1)$$

und  $\text{bin}_{t-1}(d)$  die durch führende Nullen auf die Länge  $t - 1$  aufgefüllte Binärdarstellung von  $d$  ist.

**Satz 9.** Die durch (1.1) definierte Preprocessing-Funktion  $y$  ist suffixfrei.

*Beweis.* Seien  $x \neq \tilde{x}$  zwei Texte mit  $|x| \leq |\tilde{x}|$ . Wir müssen zeigen, dass  $y(x) = y_1 y_2 \cdots y_{k+1}$  kein Suffix von  $y(\tilde{x}) = \tilde{y}_1 \tilde{y}_2 \cdots \tilde{y}_{l+1}$  ist. Im Fall  $x = \varepsilon$  ist dies klar. Für  $x \neq \varepsilon$  machen wir folgende Fallunterscheidung.

- 1. Fall:**  $|x| \not\equiv_{t-1} |\tilde{x}|$ . Dann folgt  $d \neq \tilde{d}$  und somit  $y_{k+1} \neq \tilde{y}_{l+1}$ .
- 2. Fall:**  $|x| = |\tilde{x}|$ . In diesem Fall ist  $k = l$ . Wegen  $x \neq \tilde{x}$  existiert ein Index  $i \in \{1, \dots, k\}$  mit  $x_i \neq \tilde{x}_i$ . Dies impliziert  $y_i \neq \tilde{y}_i$ , also ist  $y(x)$  kein Suffix von  $y(\tilde{x})$ .
- 3. Fall:**  $|x| \neq |\tilde{x}|$  und  $|x| \equiv_{t-1} |\tilde{x}|$ . In diesem Fall ist  $k < l$ . Da  $y(x)$  mit einer Null beginnt, aber das  $(l - k + 1)$ -te Bit von  $y(\tilde{x})$  eine Eins ist, kann  $y(x)$  kein Suffix von  $y(\tilde{x})$  sein.  $\square$

Nun kommen wir zum Fall  $t = 1$ . Sei  $y$  die durch  $y(x) := 11f(x)$  definierte Funktion, wobei  $f$  wie folgt definiert ist:

$$f(x_1 \dots x_n) = f(x_1) \dots f(x_n) \text{ mit } f(0) = 0 \text{ und } f(1) = 01.$$

Dann ist leicht zu sehen, dass  $y$  suffixfrei ist. Da die Kompressionsfunktion  $h$  bei der Berechnung von  $\hat{h}(x)$  im Fall  $t = 1$  für jedes Bit von  $y(x)$  einmal aufgerufen wird, wird  $h$  genau  $|y(x)| \leq 2(n+1)$ -mal aufgerufen. Im Fall  $t > 1$  werden dagegen nur  $k+1 = \lceil \frac{n}{t-1} \rceil + 1$  Aufrufe benötigt.

### 1.2.5 Die MD4-Hashfunktion

Die MD4-Hashfunktion (*Message Digest*) wurde 1990 von Rivest vorgeschlagen. Die Bitlänge von MD4 beträgt  $l = 128$  Bit. Bei einer Wortlänge von 32 Bit entspricht dies 4 Wörtern. Die im Folgenden vorgestellten Hashfunktionen benutzen u.a. folgende Operationen auf Wörtern.

Operatoren auf $\{0, 1\}^{32}$	
$X \wedge Y$	bitweises „Und“ von $X$ und $Y$
$X \vee Y$	bitweises „Oder“ von $X$ und $Y$
$X \oplus Y$	bitweises „exklusives Oder“ von $X$ und $Y$
$\neg X$	bitweises Komplement von $X$
$X + Y$	Ganzzahl-Addition modulo $2^{32}$
$X \rightarrow s$	Rechtsshift um $s$ Stellen
$X \leftarrow s$	zirkulärer Linksshift um $s$ Stellen

Während die Ganzzahl-Addition bei MD4 und MD5 in **little endian** Architektur (d.h. ein aus 4 Bytes  $a_3a_2a_1a_0$ ,  $0 \leq a_i \leq 255$  zusammengesetztes Wort repräsentiert die Zahl  $a_02^{24} + a_12^{16} + a_22^8 + a_3$ ) ausgeführt wird, verwendet **SHA-1** eine **big endian** Architektur (d.h.  $a_3a_2a_1a_0$ ,  $0 \leq a_i \leq 255$  repräsentiert die Zahl  $a_32^{24} + a_22^{16} + a_12^8 + a_0$ ). Der MD4-Algorithmus benutzt die folgenden Konstanten  $y_j, z_j, s_j$ ,  $j = 0, \dots, 47$

	$y_j$ (in Hexadezimaldarstellung)
$j = 0, \dots, 15$	0
$j = 16, \dots, 31$	5a827999
$j = 32, \dots, 47$	6ed9eba1

	$z_j$
$j = 0, \dots, 15$	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
$j = 16, \dots, 31$	0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15
$j = 32, \dots, 47$	0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15
	$s_j$
$j = 0, \dots, 15$	3, 7, 11, 19, 3, 7, 11, 19, 3, 7, 11, 19, 3, 7, 11, 19
$j = 16, \dots, 31$	3, 5, 9, 13, 3, 5, 9, 13, 3, 5, 9, 13, 3, 5, 9, 13
$j = 32, \dots, 47$	3, 9, 11, 15, 3, 9, 11, 15, 3, 9, 11, 15, 3, 9, 11, 15

und folgende Funktionen  $f_j$ ,  $j = 0, \dots, 47$

$$f_j(X, Y, Z) := \begin{cases} (X \wedge Y) \vee (\neg X \wedge Z), & j = 0, \dots, 15, \\ (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z), & j = 16, \dots, 31, \\ X \oplus Y \oplus Z, & j = 32, \dots, 47. \end{cases}$$

Für MD4 konnten nach ca.  $2^{20}$  Hashwertberechnungen Kollisionen aufgespürt werden. Deshalb gilt MD4 nicht mehr als kollisionsresistent.

#### MD4(x)

---

```

1 input  $x \in \{0, 1\}^*$ ,  $|x| = n$ 
2  $y := x10^k \mathbf{bin}_{64}(n)$ ,  $k \in \{0, 1, \dots, 511\}$  mit  $n + 1 + k + 64 \equiv 0 \pmod{512}$ 
3  $(H_1, H_2, H_3, H_4) := (67452301, efcdab89, 98badcfe, 10325476)$ 
4 sei  $y = M_1 \cdots M_r$ ,  $r = (n + 1 + k + 64)/512$ 
5 for  $i := 1$  to  $r$  do
6   sei  $M_i = X[0] \cdots X[15]$ 
7    $(A, B, C, D) := (H_1, H_2, H_3, H_4)$ 
8   for  $j := 0$  to 47 do
9      $(A, B, C, D) := (D, (A + f_j(B, C, D) + X[z_j] + y_j) \leftarrow s_j, B, C)$ 
10     $(H_1, H_2, H_3, H_4) := (H_1 + A, H_2 + B, H_3 + C, H_4 + D)$ 
11 output  $H_1 H_2 H_3 H_4$ 

```

---

### 1.2.6 Die MD5-Hashfunktion

Der MD5 ist eine 1991 von Rivest präsentierte verbesserte Version von MD4. Die Bitlänge von MD5 beträgt wie bei MD4  $l = 128$  Bit. Bei einer Wortlänge von 32 Bit entspricht dies 4 Wörtern. In MD5 werden teilweise andere Konstanten als in MD4 verwendet.

Zudem besitzt MD5 eine zusätzliche 4. Runde ( $j = 48, \dots, 63$ ), in der die Funktion  $f_j(X, Y, Z) = Y \oplus (X \vee \neg Z)$  verwendet wird. Außerdem wurde die in Runde 2 von MD4 verwendete Funktion durch  $f_j(X, Y, Z) := (X \wedge Z) \vee (Y \wedge \neg Z)$ ,  $j = 16 \dots 31$ , ersetzt. Die  $y$ -Konstanten sind definiert als  $y_j :=$  die ersten 32 Bit der Binärdarstellung von  $\text{abs}(\sin(j + 1))$ ,  $0 \leq j \leq 63$ , und für  $z_j$  und  $s_j$  werden folgende Konstanten benutzt.

	$z_j$
$j = 0, \dots, 15$	$z_j = j : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15$
$j = 16, \dots, 31$	$z_j = (5j + 1) \bmod 16 : 1, 6, 11, 0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12$
$j = 32, \dots, 47$	$z_j = (3j + 5) \bmod 16 : 5, 8, 11, 14, 1, 4, 7, 10, 13, 0, 3, 6, 9, 12, 15, 2$
$j = 48, \dots, 63$	$z_j = 7j \bmod 16 : 0, 7, 14, 5, 12, 3, 10, 1, 8, 15, 6, 13, 4, 11, 2, 9$
	$s_j$
$j = 0, \dots, 15$	7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22
$j = 16, \dots, 31$	5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20
$j = 32, \dots, 47$	4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23
$j = 48, \dots, 63$	6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21

Für MD5 konnten in 2004 ebenfalls Kollisionspaare gefunden werden (für die Kompressionsfunktion von MD5 gelang dies bereits 1996).

#### MD5( $x$ )

---

```

1 input  $x \in \{0, 1\}^*, |x| = n$ 
2  $y := x10^k \mathbf{bin}_{64}(n)$ ,  $k \in \{0, 1, \dots, 511\}$  mit  $n + 1 + k + 64 \equiv 0 \pmod{512}$ 
3  $(H_1, H_2, H_3, H_4) := (67452301, efcdab89, 98badcfe, 10325476)$ 
4 sei  $y = M_1 \cdots M_r$ ,  $r = (n + 1 + k + 64)/512$ 
5 for  $i := 1$  to  $r$  do
6   sei  $M_i = X[0] \cdots X[15]$ 
7    $(A, B, C, D) := (H_1, H_2, H_3, H_4)$ 
8   for  $j := 0$  to  $63$  do
9      $(A, B, C, D) := (D, B + (A + f_j(B, C, D) + X[z_j] + y_j) \leftarrow s_j, B, C)$ 
10     $(H_1, H_2, H_3, H_4) := (H_1 + A, H_2 + B, H_3 + C, H_4 + D)$ 
11 output  $H_1 H_2 H_3 H_4$ 

```

---

### 1.2.7 Die SHA-1-Hashfunktion

Der **Secure Hash Algorithm** (**SHA-1**) ist eine Weiterentwicklung des MD4 bzw. MD5 Algorithmus. Er gilt in den USA als Standard und ist Bestandteil des von der US-Behörde NIST (National Institute of Standards and Technology) im August 1991 veröffentlichten DSS (Digital Signature Standard). Die Bitlänge von **SHA-1** beträgt  $l = 160$  Bit. Bei einer Wortlänge von 32 Bit entspricht dies 5 Wörtern. **SHA-1** unterscheidet sich nur geringfügig von der SHA-0 Hashfunktion, in der eine Schwachstelle dazu führt, dass nach Berechnung von ca.  $2^{61}$  Hashwerten ein Kollisionspaar gefunden werden kann (obwohl bei einem Geburtstagsangriff auf Grund der Hashwertlänge von 160 Bit ca.  $2^{80}$  Berechnungen erforderlich sein müssten). Diese potentielle Schwäche von SHA-0 wurde im **SHA-1** dadurch entfernt, dass **SHA-1** in Zeile 8 einen zirkulären Shift um eine Bitstelle

ausführt. Der **SHA-1**-Algorithmus benutzt die folgenden Konstanten  $K_j$ ,  $j = 0, \dots, 79$

	$K_j$ (in Hexadezimaldarstellung)
$j = 0, \dots, 19$	5a827999
$j = 20, \dots, 39$	6ed9eba1
$j = 40, \dots, 59$	8f1bbcdc
$j = 60, \dots, 79$	ca62c1d6

und folgende Funktionen  $f_j$ ,  $j = 0, \dots, 79$

$$f_j(X, Y, Z) := \begin{cases} (X \wedge Y) \vee (\neg X \wedge Z), & j = 0, \dots, 19, \\ X \oplus Y \oplus Z, & j = 20, \dots, 39, \\ (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z), & j = 40, \dots, 59, \\ X \oplus Y \oplus Z, & j = 60, \dots, 79. \end{cases}$$

---

**SHA-1**( $x$ )

---

```

1 input  $x \in \{0, 1\}^*, |x| = n$ 
2  $y := x10^k \mathbf{bin}_{64}(n)$ ,  $k \in \{0, 1, \dots, 511\}$  mit  $n + 1 + k + 64 \equiv 0 \pmod{512}$ 
3  $(H_0, H_1, H_2, H_3, H_4) := (67452301, efc dab89, 98badcfe, 10325476, c3d2e1f0)$ 
4 sei  $y = M_1 \cdots M_r$ ,  $r = (n + 1 + k + 64)/512$ 
5 for  $i := 1$  to  $r$  do
6   sei  $M_i = X[0] \cdots X[15]$ 
7   for  $t := 16$  to 79 do
8      $X[t] := (X[t - 3] \oplus X[t - 8] \oplus X[t - 14] \oplus X[t - 16]) \leftarrow 1$ 
9      $(A, B, C, D, E) := (H_0, H_1, H_2, H_3, H_4)$ 
10    for  $j := 0$  to 79 do
11       $temp := (A \leftarrow 5) + f_j(B, C, D) + E + X[j] + K_j$ 
12       $(A, B, C, D, E) := (temp, A, B \leftarrow 30, C, D)$ 
13       $(H_0, H_1, H_2, H_3, H_4) := (H_0 + A, H_1 + B, H_2 + C, H_3 + D, H_4 + E)$ 
14 output  $H_0 H_1 H_2 H_3 H_4$ 

```

---