

Vorlesungsskript
Kryptologie
Wintersemester 2022/23

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

9. November 2022

Inhaltsverzeichnis

1	Kryptografische Hashverfahren	1
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
1.2.8	Die SHA-2-Familie	13
1.2.9	Kryptoanalyse von Hashfunktionen	14
1.2.10	Die Sponge-Konstruktion	16
1.2.11	Die SHA3-Familie	19
1.3	Nachrichten-Authentikationscodes (MACs)	21
1.3.1	Angriffe gegen symmetrische Hashfunktionen	22
1.3.2	Informationstheoretische Sicherheit von MACs	22

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)

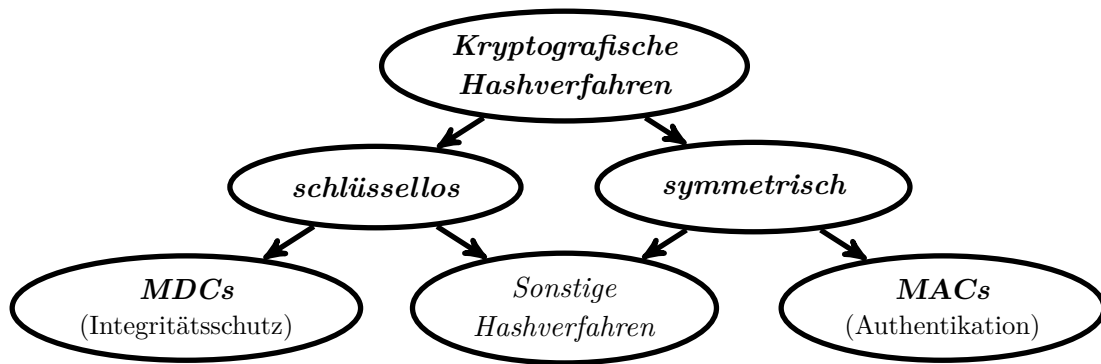


Abbildung 1.1: Eine grobe Einteilung von kryptografischen Hashverfahren.

- 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-

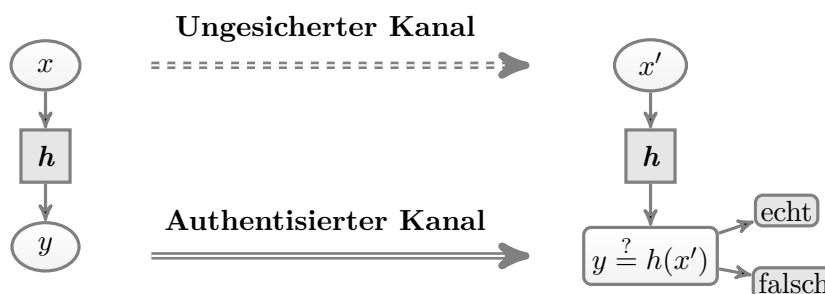


Abbildung 1.2: Einsatz eines MDC h zur Überprüfung der Integrität eines Datensatzes x .

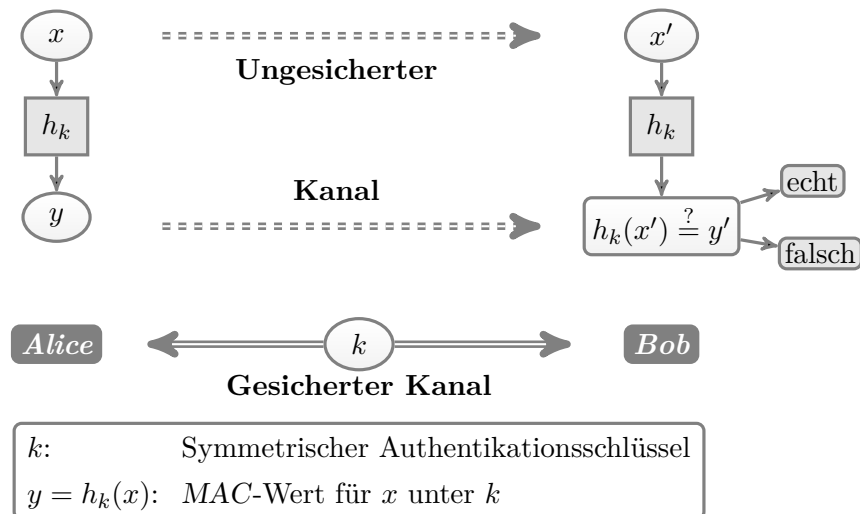


Abbildung 1.3: Verwendung eines MAC zur Nachrichtenauthentikation.

cherten Kanal gesendet (bzw. auf einem vor Manipulationen nicht sicheren Webserver abgelegt) wird, kann man wie folgt vorgehen. 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 ε 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 ε 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 ε -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  $y := x10^k \mathit{bin}_{64}(n)$ ,  $k \in \{0, 1, \dots, 511\}$  mit  $n + 1 + k + 64 \equiv 0 \pmod{512}$ 
3  $(H_0, H_1, H_2, H_3, H_4) := (67452301, efcdab89, 98badcfe, 10325476, c3d2e1f0)$ 
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  $H_0 H_1 H_2 H_3 H_4$ 

```

1.2.8 Die SHA-2-Familie

Im Jahr 2001 veröffentlichte die US-Behörde NIST drei weitere Hashfunktionen der SHA-Familie: SHA-256, SHA-384, and SHA-512. Diese Funktionen werden auch als SHA-2 Hashfunktionen bezeichnet. In 2004 kam noch SHA-224 als vierte Variante hinzu. SHA-256 und SHA-512 haben denselben Aufbau, unterscheiden sich aber in erster Linie in der benutzten Wortlänge: 32 Bit bei SHA-256 und 64 Bit bei SHA-512. Zudem werden unterschiedliche Shift- und Summationskonstanten verwendet und auch die Rundenzahlen differieren. SHA-224 und SHA-384 sind reduzierte Varianten von SHA-256 und SHA-512. Der SHA-256-Algorithmus benutzt die folgenden Konstanten K_j , $j = 0, \dots, 63$ (in

Hexadezimaldarstellung).

428a2f98, 71374491, b5c0fbcf, e9b5dba5, 3956c25b, 59f111f1, 923f82a4, ab1c5ed5, d807aa98, 12835b01, 243185be, 550c7dc3, 72be5d74, 80deb1fe, 9bdc06a7, c19bf174, e49b69c1, efbe4786, 0fc19dc6, 240ca1cc, 2de92c6f, 4a7484aa, 5cb0a9dc, 76f988da, 983e5152, a831c66d, b00327c8, bf597fc7, c6e00bf3, d5a79147, 06ca6351, 14292967, 27b70a85, 2e1b2138, 4d2c6dfc, 53380d13, 650a7354, 766a0abb, 81c2c92e, 92722c85, a2bfe8a1, a81a664b, c24b8b70, c76c51a3, d192e819, d6990624, f40e3585, 106aa070, 19a4c116, 1e376c08, 2748774c, 34b0bcb5, 391c0cb3, 4ed8aa4a, 5b9cca4f, 682e6ff3, 748f82ee, 78a5636f, 84c87814, 8cc70208, 90befffa, a4506ceb, bef9a3f7, c67178f2

Dies sind jeweils die ersten 32 Bit der binären Nachkommastellen der dritten Wurzeln der ersten 64 Primzahlen $2, \dots, 311$. SHA-256 arbeitet wie folgt.

SHA-256(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  $(H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7) := (6a09e667, bb67ae85, 3c6ef372, a54ff53a,$ 
4    $510e527f, 9b05688c, 1f83d9ab, 5be0cd19)$ 
5 sei  $y = M_1 \cdots M_r$ ,  $r = (n + 1 + k + 64)/512$ 
6 for  $i := 1$  to  $r$  do
7   sei  $M_i = X[0] \cdots X[15]$ 
8   for  $t := 16$  to  $63$  do
9      $s0 := (X[t - 15] \hookrightarrow 7) \oplus (X[t - 15] \hookrightarrow 18) \oplus (X[t - 15] \rightarrow 3)$ 
10     $s1 := (X[t - 2] \hookrightarrow 17) \oplus (X[t - 2] \hookrightarrow 19) \oplus (X[t - 2] \rightarrow 10)$ 
11     $X[t] := X[t - 16] + s0 + X[t - 7] + s1$ 
12     $(A, B, C, D, E, F, G, H) := (H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7)$ 
13    for  $j := 0$  to  $63$  do
14       $s0 := (A \hookrightarrow 2) \oplus (A \hookrightarrow 13) \oplus (A \hookrightarrow 22)$ 
15       $maj := (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C)$ 
16       $t2 := s0 + maj$ 
17       $s1 := (E \hookrightarrow 6) \oplus (E \hookrightarrow 11) \oplus (E \hookrightarrow 25)$ 
18       $ch := (E \wedge F) \oplus (\neg E \wedge G)$ 
19       $t1 := H + s1 + ch + K_j + X[j]$ 
20       $(A, B, C, D, E, F, G, H) := (t1 + t2, A, B, C, D + t1, E, F, G)$ 
21       $(H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7)$ 
22       $:= (H_0 + A, H_1 + B, H_2 + C, H_3 + D, H_4 + E, H_5 + F, H_6 + G, H_7 + H)$ 
23 output  $H_0 H_1 H_2 H_3 H_4 H_5 H_6 H_7$ 
```

Die Initialwerte von H_0, \dots, H_7 in den Zeilen 3 und 4 sind jeweils die ersten 32 Bit der binären Nachkommastellen der Wurzeln der Primzahlen 2, 3, 5, 7, 11, 13, 17, 19.

1.2.9 Kryptoanalyse von Hashfunktionen

Bereits 1991 wurden von Den Boer und Bosselaers Schwächen im MD4 aufgedeckt. Im August 2004 erschien ein Bericht [1] mit einer Anleitung, wie sich Kollisionen für MD4 mittels “hand calculation” finden lassen.

In 1993, fanden den Boer und Bosselaers einen Weg, so genannte “Pseudo-Kollisionen” für die MD5 Kompressionsfunktion zu generieren. In 1996, fand Dobbertin ein Kollisionspaar

für die MD5 Kompressionsfunktion.

Im August 2004 wurden schließlich Kollisionen für MD5 von Xiaoyun Wang, Dengguo Feng, Xuejia Lai und Hongbo Yu berechnet. Der benötigte Aufwand wurde mit ca. 1 Stunde auf einem IBM p690 Cluster abgeschätzt.

Im März 2005 veröffentlichten Arjen Lenstra, Xiaoyun Wang und Benne de Weger zwei X.509 Zertifikate mit unterschiedlichen Public-keys, die auf denselben MD5-Hashwert führten. Nur wenige Tage später beschrieb Vlastimil Klima eine Möglichkeit, Kollisionen für MD5 innerhalb weniger Stunden auf einem Notebook zu berechnen. Mittels der so genannten Tunneling-Methode wurde die Rechenzeit vom gleichen Autor im März 2006 auf eine Minute verkürzt.

Auf der CRYPTO 98 stellten Chabaud und Joux einen Angriff auf SHA-0 vor, der ein Kollisionspaar mit nur 2^{61} Hashwertberechnungen (anstelle von 2^{80} bei einem Geburtstagsangriff) aufspürt.

In 2004 fanden Biham und Chen Beinahe-Kollisionen für den SHA-0, bei denen sich die Hashwerte nur an 18 von den 160 Bitpositionen unterschieden. Zudem legten sie volle Kollisionen für den auf 62 Runden reduzierten SHA-0 Algorithmus vor.

Schließlich wurde im August 2004 die Berechnung einer Kollision für den vollen 80-Runden SHA-0 Algorithmus von Joux, Carribault, Lemuet und Jalby bekannt gegeben. Hierzu wurden lediglich 2^{51} Hashwerte berechnet, die ca. 80 000 Stunden CPU-Rechenzeit auf einem 2-Prozessor 256-Itanium Supercomputer benötigten.

Ebenfalls im August 2004 wurde von Wang, Feng, Lai und Yu auf der CRYPTO 2004 eine Angriffsmethode für MD5, SHA-0 und andere Hashfunktionen vorgestellt, mit der sich die Anzahl der Hashwertberechnungen auf 2^{40} senken lässt. Dies wurde im Februar 2005 von Xiaoyun Wang, Yiqun Lisa Yin und Hongbo Yu geringfügig auf 2^{39} Hashwertberechnungen verbessert.

Aufgrund der erfolgreichen Angriffe auf SHA-0 rieten mehrere Experten von einer weiteren Verwendung des **SHA-1** ab. Daraufhin kündigte die amerikanische Behörde NIST an, **SHA-1** in 2010 zugunsten der SHA-2 Varianten abzulösen.

Im Jahr 2005 veröffentlichten Rijmen und Oswald einen Angriff, der mit weniger als 2^{80} Hashwertberechnungen ein Kollisionspaar für den auf 53 Runden reduzierten **SHA-1** Algorithmus findet. Nur wenig später kündigten Xiaoyun Wang, Yiqun Lisa Yin und Hongbo Yu einen Angriff auf den vollen 80-Runden **SHA-1** mit 2^{69} Hashwertberechnungen an. Im August 2005 erfuhr der benötigte Aufwand von Xiaoyun Wang, Andrew Yao und Frances Yao auf der CRYPTO 2005 eine weitere Reduktion auf 2^{63} Berechnungen. In 2008 wurde von Stephane Manuel ein Kollisionsangriff mit einem geschätzten Aufwand von 2^{51} bis 2^{57} Berechnungen veröffentlicht.

Die besten bekannten Angriffe gegen SHA-2 brechen die von 64 auf 41 Runden reduzierte Variante von SHA-256 und die von 80 auf 46 Runden reduzierte Variante von SHA-512.

Im Oktober 2012 wurde der Hash-Algorithmus Keccak als Gewinner des vom NIST ausgeschrieben Wettbewerbs für den SHA-3-Algorithmus ausgewählt. Die Intention dabei war nicht, SHA-2 als Standard durch SHA-3 abzulösen, zumal bisher keine erfolgreichen Angriffe gegen SHA-2 bekannt sind. Vielmehr ging es bei diesem Wettbewerb darum, angesichts der erfolgreichen Angriffe gegen MD5 und SHA-0, die einen ähnlichen Aufbau wie SHA-1 und SHA-2 haben, eine auf einem vollkommen anderen Entwurfsprinzip basierende Alternative zur Verfügung zu stellen.

1.2.10 Die Sponge-Konstruktion

Die Konstruktionsidee hinter dem SHA-3-Gewinner Keccak wird von den Autoren als *Sponge* (Schwamm) bezeichnet. Auf der Basis dieser Entwurfsmethode lassen sich außer Hashfunktionen bspw. auch Pseudozufallsgeneratoren gewinnen. Der Aufbau eines Sponges ähnelt oberflächlich betrachtet der bereits vorgestellten Konstruktion von iterierten Hashfunktionen, weist aber einige Unterschiede auf. So basiert ein Sponge statt auf einer Kompressionsfunktion h auf einer Permutation (oder allgemeiner Transformation) $f : \{0, 1\}^b \rightarrow \{0, 1\}^b$, die wie h iteriert angewendet wird. Mittels der Funktion f wird eine Folge von **Zuständen** $s_i \in \{0, 1\}^b$ generiert, deren Länge b als **Weite** des Sponges bezeichnet wird. Die Zustände $s_i = z_i u_i$ werden in zwei Teilblöcke z_i und u_i der Länge r bzw. c zerlegt, die als **äußerer** bzw. **innerer Zustand** bezeichnet werden.

Wie der Name schon sagt, verbleiben die Bits des inneren Zustands im Sponge, d.h. sie dienen nur zur Berechnung des nächsten Zustands und werden im Gegensatz zu den Bits des äußeren Zustands nicht unmittelbar für die Gewinnung der Ausgabe genutzt. Die Anzahl $c = |z_i|$ der Bits des inneren Zustands wird als **Kapazität** des Sponges bezeichnet und ist sein wichtigster Sicherheitsparameter. Die Anzahl $r = |u_i|$ der Bits des äußeren Zustands heißt **Bitrate**. Da $r + c = b$ gelten muss, lässt sich über die Wahl der Parameter r und c ein Trade-off zwischen der Effizienz und der Sicherheit des Sponges realisieren.

Die Funktion f bildet den Kern der Sponge-Konstruktion. Bevor f iteriert angewendet wird, um die Zustandsfolge s_i zu generieren, findet ein Preprocessing statt. Wir definieren nun mögliche Anforderungen an die Preprocessing-Funktion y .

Definition 10. Sei y eine Funktion der Form $y : \{0, 1\}^* \rightarrow \{0, 1\}^*$. y heißt **Paddingfunktion**, falls für alle $x \in \{0, 1\}^*$ ein $z \in \{0, 1\}^*$ mit $y(x) = xz$ existiert. Eine solche Funktion heißt **Paddingfunktion für eine Bitrate** $r \geq 1$, falls $|y(x)| \equiv_r 0$ für alle $x \in \{0, 1\}^*$ gilt. Die Funktion y heißt **sponge-konform** für Bitrate $r \geq 1$, falls gilt:

- $\forall i = 0, \dots, r-1 \exists z \in \{0, 1\}^* \forall x \in \{0, 1\}^* \text{ mit } |x| \equiv_r i : y(x) = xz$
- $\forall k \geq 0 \forall x \neq x' : y(x) \neq y(x')0^{kr}$.

Es ist leicht zu sehen, dass die Funktion $\text{pad}10^*1_r : \{0, 1\}^* \rightarrow \{0, 1\}^*$ definiert durch

$$\text{pad}10^*1_r(x) = x10^d1 \text{ mit } d = \min\{i \geq 0 \mid |x| + 2 + i \equiv_r 0\}$$

eine sponge-konforme Paddingfunktion für die Bitrate r ist. Ohne die abschließende 1 wäre dies nicht der Fall.

Definition 11. Sei y eine Paddingfunktion für $r \geq 1$ und sei $f : \{0, 1\}^b \rightarrow \{0, 1\}^b$. Für $x \in \{0, 1\}^*$ sei $y(x) = y_1 \dots y_k$ mit $|y_i| = r$ für $i = 1, \dots, k$. Sei $s_0 = 0^b$ und für $i \geq 1$ seien $s_i = z_i u_i \in \{0, 1\}^b$ mit $z_i \in \{0, 1\}^r, u_i \in \{0, 1\}^c$ mit

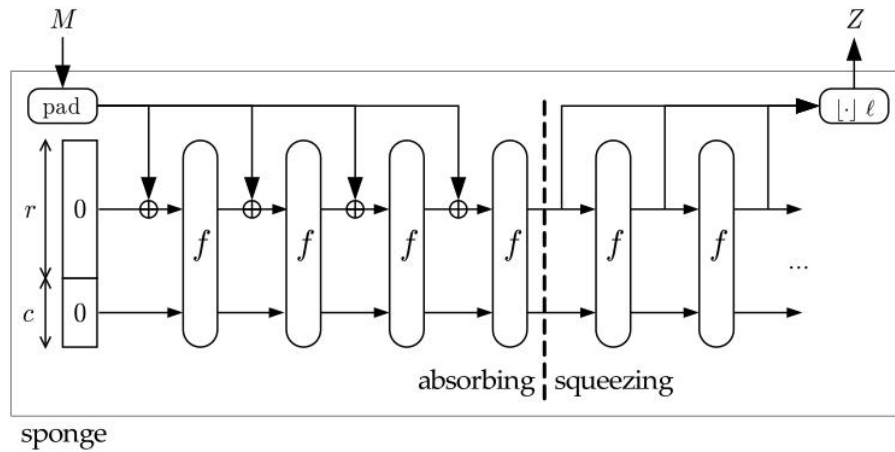
$$s_i = \begin{cases} f(s_{i-1} \oplus y_i 0^c) & 1 \leq i \leq k & (\text{Absorptionsphase}) \\ f(s_{i-1}) & i > k & (\text{Squeezing-Phase}) \end{cases}$$

Dann ist $\text{Sponge}_{f,y,r} : \mathbb{N} \times \{0, 1\}^* \rightarrow \{0, 1\}^*$ die Funktion mit

$$\text{Sponge}_{f,y,r}(l, x) = z_k \dots z_{k+m-1} z'_{k+m},$$

wobei $m = \lfloor \frac{l}{r} \rfloor$ und z'_{k+m} das Präfix von z_{k+m} der Länge $l - mr$ ist. Für die Analyse definieren wir noch die beiden Funktionen

$$\text{Absorb}_{f,r}(y_1 \dots y_k) = s_k \text{ und } \text{Squeeze}_{f,r}(l, s_k) = z_k \dots z_{k+m-1} z'_{k+m}.$$



Den Aufwand, ein Paar $x \neq x'$ mit $\text{Sponge}_{f,y,r}(l, x) = \text{Sponge}_{f,y,r}(l, x')$ zu finden, können wir nach oben durch den Aufwand abschätzen, ein Paar $y \neq y'$ in $\bigcup_{k \geq 1} \{0, 1\}^{kr}$ mit $y = y_1 \dots y_k$, $y' = y'_1 \dots y'_{k'}$ und $\text{Absorb}_{f,r}(y) = \text{Absorb}_{f,r}(y')$ zu finden. Hierbei reicht es, dass die inneren Zustände u_k und $u'_{k'}$ von $s_k = z_k u_k = \text{Absorb}_{f,r}(y)$ und $s'_{k'} = z'_{k'} u'_{k'} = \text{Absorb}_{f,r}(y')$ gleich sind, d.h. $y \neq y'$ ist ein **inneres Kollisionspaar** für die Funktion $\text{Absorb}_{f,r}$. Setzen wir nämlich y_{k+1} und $y'_{k'+1}$ auf die äußeren Zustände z_k von s_k und $z'_{k'}$ von $s'_{k'}$, so folgt für die Texte $x = yy_{k+1}$ und $x' = y'y'_{k'+1}$:

$$\begin{aligned} \text{Absorb}_{f,r}(x) &= f(s_k \oplus (y_{k+1} 0^c)) = f(0^r u_k) = f(0^r u'_{k'}) \\ &= f(s'_{k'} \oplus (y'_{k'+1} 0^c)) = \text{Absorb}_{f,r}(x') \end{aligned}$$

Falls das Suffix z von $y(x) = xz$ nur von $|x| \bmod r$ abhängt, gilt wegen $|x| \equiv_r |x'| \equiv_r 0$ dann auch $y(x') = x'z$ und somit folgt

$$\text{Absorb}_{f,r}(y(x)) = \text{Absorb}_{f,r}(xz) = \text{Absorb}_{f,r}(x'z) = \text{Absorb}_{f,r}(y(x')),$$

was $\text{Sponge}_{f,y,r}(l, x) = \text{Sponge}_{f,y,r}(l, x')$ für alle $l \geq 1$ impliziert.

Um eine solche innere Kollision zu finden, hilft es, sich die 2^c inneren Zustände $u \in \{0, 1\}^c$ als Knoten eines gerichteten Multigraphen G vorzustellen, der für jedes Paar $(xu, x'u')$ mit $f(xu) = x'u'$ eine Kante $(u, u')_{x,x'}$ von u nach u' mit dem Label x, x' enthält. Ziel ist es dann, zwei verschiedene Pfade von 0^c zu demselben Knoten v zu finden, wobei zwei Pfade auch dann verschieden sind, wenn sich die Kanten nur in den Labeln unterscheiden. Wird f durch eine Zufallsfunktion modelliert (ZOM), so lassen bereits berechnete Werte von f keine Rückschlüsse auf die Werte für andere Argumente zu. Anders als beim ZOM für eine Hashfunktion kann es sich dennoch für den Angreifer lohnen, die Argumente von f adaptiv nach einer Strategie \mathcal{S} zu wählen. Der Algorithmus **InnerCollision** fasst dieses Vorgehen zusammen. Der Parameter q beschränkt hierbei die Anzahl der Aufrufe der Transformation f .

Satz 12. Für jede Strategie \mathcal{S} gibt $\text{INNERCOLLISION}(f, r, q, \mathcal{S})$ im ZOM mit Erfolgswahrscheinlichkeit höchstens

$$\varepsilon = 1 - \prod_{i=1}^q \left(1 - \frac{i}{2^c}\right)$$

ein inneres Kollisionspaar (x, x') für die Funktion $\text{Absorb}_{f,r}$ aus. Wählt \mathcal{S} nur von 0^c aus erreichbare Knoten u und kein Argument xu mehrmals, so ist die Erfolgswahrscheinlichkeit exakt ε .

Prozedur InnerCollision(f, r, q, \mathcal{S}) mit $f : \{0, 1\}^b \rightarrow \{0, 1\}^b$

```

1   $c := b - r$ 
2  initialisiere den Multi-Digraphen  $G = (V, E) := (\{0, 1\}^c, \emptyset)$ 
3  for  $i := 1$  to  $q$  do
4    wähle  $u \in V$  und  $v \in \{0, 1\}^r$  nach Strategie  $\mathcal{S}$ 
5     $E := E \cup \{u \xrightarrow{v, \tilde{v}} \tilde{u}\}$ , wobei  $f(vu) = \tilde{v}\tilde{u}$  ist
6  if  $\exists$  2 Pfade  $0^c \xrightarrow{v_1, \tilde{v}_1} u_1 \xrightarrow{v_2, \tilde{v}_2} u_2 \xrightarrow{v_3, \tilde{v}_3} \dots \xrightarrow{v_k, \tilde{v}_k} u_k$  und
7     $0^c \xrightarrow{v'_1, \tilde{v}'_1} u'_1 \xrightarrow{v'_2, \tilde{v}'_2} u'_2 \xrightarrow{v'_3, \tilde{v}'_3} \dots \xrightarrow{v'_{k'}, \tilde{v}'_{k'}} u'_{k'}$  mit  $u_k = u'_{k'}$  then
8     $y := y_1 \dots y_k$  mit  $y_1 = v_1$  und  $y_i = \tilde{v}_{i-1} \oplus v_i$  für  $i = 2, \dots, k$ 
9     $y' := y'_1 \dots y'_{k'}$  mit  $y'_1 = v'_1$  und  $y'_i = \tilde{v}'_{i-1} \oplus v'_i$  für  $i = 2, \dots, k'$ 
10   return( $y, y'$ )
11 else
12   return(?)
```

Abbildung 1.9: Bestimmung eines inneren Kollisionspaares

Beweis. Wir zeigen zuerst, dass $y \neq y'$ gilt, falls die beiden Pfade

$$\begin{aligned}
 P : 0^c &= u_0 \xrightarrow{v_1, \tilde{v}_1} u_1 \xrightarrow{v_2, \tilde{v}_2} u_2 \xrightarrow{v_3, \tilde{v}_3} u_3 \dots \xrightarrow{v_k, \tilde{v}_k} u_k \text{ und} \\
 P' : 0^c &= u'_0 \xrightarrow{v'_1, \tilde{v}'_1} u'_1 \xrightarrow{v'_2, \tilde{v}'_2} u'_2 \xrightarrow{v'_3, \tilde{v}'_3} u'_3 \dots \xrightarrow{v'_{k'}, \tilde{v}'_{k'}} u'_{k'}
 \end{aligned}$$

von 0^c nach $u_k = u'_{k'}$ verschieden sind, wobei wir $k \leq k'$ annehmen. Da f eine Funktion ist, gilt für $i = 1, \dots, k$:

$$u_{i-1}v_i = u'_{i-1}v'_i \Rightarrow \tilde{v}_i u_i = f(v_i u_i) = f(v'_i u'_i) = \tilde{v}'_i u'_i$$

Da P und P' verschieden sind, muss also ein $i \in \{0, \dots, k\}$ ex. mit

$$u_0 v_0 \tilde{v}_0 \dots u_i = u'_0 v'_0 \tilde{v}'_0 \dots u'_i \text{ und } (v_i \neq v'_i \vee i = k < k')$$

Somit folgt $y_i = \tilde{v}_{i-1} \oplus v_i \neq \tilde{v}'_{i-1} \oplus v'_i = y'_i$ oder $i = k < k'$. In beiden Fällen folgt $y = y_1 \dots y_k \neq y'_1 \dots y'_{k'} = y'$.

Sei nun E_i das Ereignis “ G enthält nach dem i -ten Durchlauf der for-Schleife zwei verschiedene Pfade von 0^c zu einem Knoten u .” Da nur durch eine Kante zwischen zwei von 0^c aus erreichbaren Knoten ein zweiter Pfad von 0^c aus geschlossen werden kann und nach $i - 1$ Durchläufen höchstens i von 2^c Knoten erreichbar sind, gilt

$$\Pr[E_i | \bar{E}_1 \cap \dots \cap \bar{E}_{i-1}] \leq \frac{i}{2^c}$$

Wählt \mathcal{S} nur erreichbare Knoten u und kein Argument vu mehrfach, so impliziert das Ereignis $\bar{E}_1 \cap \dots \cap \bar{E}_{i-1}$, dass nach dem i -ten Durchlauf **genau** i Knoten erreichbar sind (sonst gäbe es bereits zwei Pfade von 0^c zu einem Knoten in G) und es gilt sogar Gleichheit. Daher ist die Erfolgswahrscheinlichkeit von INNERCOLLISION(f, r, q, \mathcal{S})

$$\begin{aligned}
 1 - \Pr[\bar{E}_1 \cap \dots \cap \bar{E}_q] &= 1 - \Pr[\bar{E}_1] \Pr[\bar{E}_2 | \bar{E}_1] \dots \Pr[\bar{E}_q | \bar{E}_1 \cap \dots \cap \bar{E}_{q-1}] \\
 &\leq 1 - \left(1 - \frac{1}{2^c}\right) \left(1 - \frac{2}{2^c}\right) \dots \left(1 - \frac{q}{2^c}\right) = \varepsilon
 \end{aligned}$$

□

Mit der Approximation $1 - x \approx e^{-x}$ erhalten wir die Abschätzung

$$\begin{aligned}\varepsilon &= 1 - \prod_{i=1}^q \left(1 - \frac{i}{2^c}\right) \approx 1 - \prod_{i=1}^q e^{-\frac{i}{2^c}} = 1 - e^{-\frac{1}{2^c} \sum_{i=1}^q i} \\ &= 1 - e^{-\frac{q(q+1)}{2 \cdot 2^c}} \approx 1 - e^{-\frac{q^2}{2 \cdot 2^c}} \approx q^2 / 2 \cdot 2^c\end{aligned}$$

Für q ergibt sich daraus die Abschätzung

$$q \approx c_\varepsilon \sqrt{2^c}$$

mit einer von ε abhängigen Konstanten $c_\varepsilon = \sqrt{2\varepsilon}$. Da im Fall $l < c$ ein Geburtstagsangriff gegen die Funktion $x \mapsto \text{Sponge}_{f,y,r}(l, x)$ effizienter ist, erhalten wir die obere Schranke $\min(2^{c/2}, 2^{l/2})$ für den Aufwand eines mit Wahrscheinlichkeit $1/2$ erfolgreichen Angriffs gegen die Funktion $\text{Sponge}_{f,y,r}$ im ZOM.

1.2.11 Die SHA3-Familie

Im Oktober 2012 wurde der Hash-Algorithmus Keccak als Gewinner des vom NIST ausgeschrieben Wettbewerbs für den SHA3-Algorithmus ausgewählt und am 5. August 2015 als Alternative zu SHA-2 standardisiert. Die Intention dabei war nicht, SHA-2 als Standard durch SHA-3 abzulösen, zumal bisher keine erfolgreichen Angriffe gegen SHA-2 bekannt sind. Vielmehr ging es bei diesem Wettbewerb darum, angesichts der erfolgreichen Angriffe gegen MD5, SHA-0 und SHA-1, die einen ähnlichen Aufbau wie SHA-2 haben, eine auf einem vollkommen anderen Entwurfsprinzip basierende Alternative zur Verfügung zu stellen.

Um geeignete Kandidaten für SHA-3 zu finden, veranstaltete das NIST ähnlich wie beim AES von Nov. 2007 bis Okt. 2012 einen Wettbewerb. Die Minimalanforderungen an die Hashfunktionen waren, dass sie Nachrichten bis zu einer Obergrenze von mindestens $2^{64} - 1$ Bit hashen und mindestens die Hash-Längen 224, 256, 384 und 512 Bit unterstützen. Von den eingereichten 64 Hashfunktionen erfüllten 51 die Teilnahmebedingungen und wurden für Runde 1 akzeptiert. In Runde 1 wurde die Sicherheit und Performanz dieser 51 Kandidaten von Kryptologen aus aller Welt in einem offenen Bewertungsprozess analysiert und 14 der 51 Kandidaten erreichten Runde 2.

Nach jeder Runde konnten Veränderungen an den Algorithmen vorgenommen werden, um auf die Ergebnisse der Analysen zu reagieren. Im Fall von Keccak wurde die Rundenzahl der Permutationsfunktion von 18 auf 24 erhöht, um eine größere Sicherheitsreserve zu erreichen. Zudem wurde die Bitrate bei einigen Varianten vergrößert, um die Effizienz zu verbessern.

Im Dezember 2010 wurden die fünf Finalisten BLAKE, Grøstl, JH, Keccak und Skein bekanntgegeben. Der Gewinner Keccak wies eine deutlich bessere Hardware-Performanz als seine Konkurrenten auf. Dagegen hatten BLAKE und Skein die beste Software-Performanz und waren in den meisten Fällen schneller als SHA-2. Keccak erlaubt für die Sponge-Weite die Wahl folgender 7 Werte $b_w \in \{25 \cdot 2^w \mid w = 0, \dots, 6\} = \{25, 50, 100, 200, 400, 800, 1600\}$. Entsprechend stehen 7 Permutationen $f_w : \{0, 1\}^{b_w} \mapsto \{0, 1\}^{b_w}$, $w = 0, \dots, 6$ zur Verfügung. Im SHA3-Standard ist $w = 6$, also $b = b_6 = 1600$ und SHA-3 stellt für $l = 224, 256, 384, 512$ jeweils eine Hashfunktion $\text{SHA3-}l$ mit der festen Hash-Länge l und der Kapazität $c = 2l$ zur Verfügung. Zudem enthält SHA-3 die Hashfunktionen SHAKE128 und SHAKE256 mit flexibler Hash-Länge und den Kapazitäten $c = 256$ bzw. $c = 512$.

Die SHA3-Familie besteht aus folgenden Hashfunktionen, die sich in der Kombination von Bitrate r und Ausgabelänge l unterscheiden:

Name	Hashlänge l	Rate r	Kap. c	Padding
SHA3-224	224	1152	448	0110*1
SHA3-256	256	1088	512	- "-
SHA3-384	384	832	768	- "-
SHA3-512	512	576	1024	- "-
SHAKE128	variabel	1344	256	111110*1
SHAKE256	- "-	1088	512	- "-

Für die Hash-Längen $l \in \{224, 256, 384, 512\}$ gilt also

$$\text{SHA3-}l(x) = \text{Sponge}_{f_6, \text{pad}10^*1_r, r}(l, x01) \text{ mit } c = 2l \text{ und } r = 1600 - 2l$$

Das zusätzliche Padding 01 soll dabei SHA3- l von anderen Anwendungen von Keccak mit denselben Werten b, l, r unterscheiden.

Wir betrachten nun die Funktion $f = f_w$ für $w = 6$ im Detail. Die Funktion $f : \{0, 1\}^{5 \times 5 \times 64} \rightarrow \{0, 1\}^{5 \times 5 \times 64}$ bildet einen $(5 \times 5 \times 64)$ -Block X auf einen ebensolchen Block $X' = f(X)$ ab. Hierzu wird $12 + 2w = 24$ -mal eine Rundenfunktion

$$f^r : \{0, 1\}^{5 \times 5 \times 64} \rightarrow \{0, 1\}^{5 \times 5 \times 64}, \quad r = 0, \dots, 23$$

aufgerufen, die unter Verwendung einer Rundenkonstanten RC_r den Block $X_{r+1} = f^r(X_r)$ für $r = 0, \dots, 23$ berechnet, wobei $X_0 = X$ und $X' = X_{24}$ ist.

Es gilt

$$f^r(X_r) = \iota_r(\chi(\pi(\rho(\theta(X_r))))),$$

wobei θ, ρ, π, χ und ι_r Permutationen auf der Menge $\{0, 1\}^{5 \times 5 \times 64}$ sind.

Zur Beschreibung der Permutationen θ, ρ, π, χ und ι_r bezeichnen wir die einzelnen Bits eines Blocks $Z \in \{0, 1\}^{5 \times 5 \times 64}$ mit $Z[x, y, z]$ für $x, y = 0, \dots, 4$ und $z = 0, \dots, 63$. Die Funktion θ ist so gewählt, dass sich $\theta^{-1}(Z)$ an möglichst vielen Bits ändert, falls eines in Z geflippt wird. Hierzu führt θ auf den **Wörtern** $Z[x, y] = Z[x, y, 0] \dots Z[x, y, 63]$ des Zustandsvektors Z folgende Operationen aus:

- $S(x) := Z[x, 0] \oplus \dots \oplus Z[x, 4]$
- $R(x) := S(x) \leftarrow 1$
- $Z[x, y] := Z[x, y] \oplus S(x - 1) \oplus R(x + 1),$

wobei die Indexoperationen modulo 5 erfolgen. Die Transposition ρ rotiert die Wörter von Z wie folgt:

- $Z[x, y] := Z[x, y] \leftarrow w_{x,y} \bmod 64,$

wobei $w_{0,0} = 0$ und $w_{x,y} = (t + 1)(t + 2)/2$ für $(x, y) = (0, 1) \begin{pmatrix} 3 & 1 \\ 2 & 0 \end{pmatrix}^t$ und $t = 0, \dots, 23$ gilt.

Die Transposition π permutiert die Wörter von Z wie folgt:

- $Z[3x + 2y, x] := Z[x, y]$

Die Funktion χ ist die einzige nichtlineare Funktion und operiert innerhalb der 5-Bit-Spalten $Z[x, z]$ des Zustandsvektors, wobei jedes Bit nur von 2 anderen in dieser Spalte abhängt:

- $Z[x, y] := Z[x, y] \oplus (\neg Z[x, y + 1] \wedge Z[x, y + 2])$

Schlussendlich setzt ι_r das Wort $Z[0, 0]$ auf das Wort

- $Z[0, 0] := Z[0, 0] \oplus RC_r,$

wobei das Bit an Position $2^m - 1$ in RC_r für $m = 0, \dots, 6$ das Bit an Position $7r + m$ eines LFSR mit dem erzeugenden Polynom $x^8 + x^6 + x^5 + x^4 + 1$ ist und alle übrigen Bits in RC_r Null sind.

1.3 Nachrichten-Authentikationscodes (MACs)

Definition 13. Eine **Hashfamilie** $\mathcal{H} = (X, Y, K, H)$ wird durch folgende Komponenten beschrieben:

- X , eine endliche oder unendliche Menge von Texten,
- Y , endliche Menge aller möglichen **Hashwerte**, $\|Y\| \leq \|X\|$,
- K , endlicher **Schlüsselraum** (key space), wobei jeder Schlüssel $k \in K$ eine Hashfunktion $h_k: X \rightarrow Y$ in H spezifiziert, d.h. $H = \{h_k \mid k \in K\}$.

Im folgenden werden wir die Größe $\|X\|$ des Textraumes mit n , die des Hashwertbereiches Y mit m und die des Schlüsselraumes K mit l bezeichnen. Wir nennen dann \mathcal{H} auch eine **(n, m, l) -Hashfamilie** oder einen **(n, m, l) -MAC**.

Damit ein geheimer Schlüssel k für die Authentifizierung mehrerer Nachrichten benutzt werden kann, ohne dass dies einem potentiellen Angreifer zur nichtautorisierten Berechnung von gültigen MAC-Werten verhilft, sollte folgende Bedingung erfüllt sein.

Berechnungsresistenz: Auch wenn eine Reihe von unter einem Schlüssel k generierten Text-Hashwert-Paaren $(x_1, h_k(x_1)), \dots, (x_n, h_k(x_n))$ bekannt ist, erfordert es einen immensen Aufwand, ohne Kenntnis von k ein weiteres Paar (x, y) mit $y = h_k(x)$ zu finden.

Bei Verwendung einer berechnungsresistenten Hashfunktion ist es einem Angreifer nicht möglich, an Bob eine Nachricht x zu schicken, die Bob als von Alice stammend anerkennt.

Verwendung eines MAC zur Versiegelung von Software

Mithilfe einer berechnungsresistenten Hashfunktion kann der Integritätsschutz für mehrere Datensätze auf die Geheimhaltung eines Schlüssels k zurückgeführt werden.

Um die Datensätze x_1, \dots, x_n gegen unbefugt vorgenommene Veränderungen zu schützen, legt man sie zusammen mit ihren MAC-Werten $y_1 = h_k(x_1), \dots, y_n = h_k(x_n)$ auf einem unsicheren Speichermedium ab und bewahrt den geheimen Schlüssel k an einem sicheren Ort auf. Bei einem späteren Zugriff auf einen Datensatz x_i lässt sich dessen Unversehrtheit durch einen Vergleich von y_i mit dem Ergebnis $h_k(x_i)$ einer erneuten MAC-Berechnung überprüfen.

Da auf diese Weise ein wirksamer Schutz der Datensätze gegen Viren und andere Manipulationen erreicht wird, spricht man von einer Versiegelung der gespeicherten Datensätze.

1.3.1 Angriffe gegen symmetrische Hashfunktionen

Ein Angriff gegen einen MAC hat die unbefugte Berechnung von MAC-Werten zum Ziel. Das heißt, der Angreifer versucht, MAC-Werte $h_k(x)$ ohne Kenntnis des geheimen Schlüssels k zu berechnen. Entsprechend der Art des zur Verfügung stehenden Textmaterials lassen sich die Angriffe gegen einen MAC wie folgt klassifizieren.

Impersonation

Der Angreifer kennt nur den benutzten MAC und versucht ein Paar (x, y) mit $h_k(x) = y$ zu generieren, wobei k der (dem Angreifer unbekannte) Schlüssel ist.

Substitution

Der Angreifer versucht in Kenntnis eines Paares $(x, h_k(x))$ ein Paar (x', y') mit $x' \neq x$ und $h_k(x') = y'$ zu generieren.

Angriff bei bekanntem Text (*known-text attack*)

Der Angreifer kennt für eine Reihe von Texten x_1, \dots, x_r (die er nicht selbst wählen konnte) die zugehörigen MAC-Werte $h_k(x_1), \dots, h_k(x_r)$ und versucht, ein Paar (x', y') mit $h_k(x') = y'$ und $x' \notin \{x_1, \dots, x_r\}$ zu generieren.

Angriff bei frei wählbarem Text (*chosen-text attack*)

Der Angreifer kann die Texte x_i selbst wählen.

Angriff bei adaptiv wählbarem Text (*adaptive chosen-text attack*)

Der Angreifer kann die Wahl des Textes x_i von den zuvor erhaltenen MAC-Werten $h_k(x_j)$, $j < i$, abhängig machen.

Auch wenn die Anwender den Schlüssel nach jeder MAC-Berechnung wechseln, sollte \mathcal{H} zumindest einem **Substitutionsangriff** widerstehen, da der Gegner sonst ein Paar (x, y) abfangen und durch ein anderes ersetzen könnte.

1.3.2 Informationstheoretische Sicherheit von MACs

Modell: Schlüssel k und Nachrichten x werden unabhängig gemäß einer Wahrscheinlichkeitsverteilung $p(k, x) = p(k)p(x)$ generiert, welche dem Angreifer bekannt ist. Wir nehmen o.B.d.A. an, dass $p(x) > 0$ und $p(k) > 0$ für alle $x \in X$ und $k \in K$ gilt.

Erfolgswahrscheinlichkeit für Impersonation

Sei α die Wahrscheinlichkeit, mit der sich ein Angreifer bei optimaler Strategie als Alice ausgeben kann, ohne dass Bob dies bemerkt.

Für ein Paar $(x, y) \in X \times Y$ sei $\alpha(x, y)$ die Wahrscheinlichkeit, dass dem Gegner mit der Wahl des Paares (x, y) eine Impersonation gelingt. Dann ist $\alpha(x, y) = p(y|x) = p(x \mapsto y)$ die Wahrscheinlichkeit, dass der zufällig gewählte Schlüssel den Text x auf den MAC-Wert y abbildet:

$$\alpha(x, y) = \sum_{k \in K(x, y)} p(k)$$

wobei $K(x, y) = \{k \in K \mid h_k(x) = y\}$ alle Schlüssel enthält, die x auf y abbilden. Folglich ist

$$\alpha(x) = \max\{\alpha(x, y) \mid y \in Y\}$$

die maximale Wahrscheinlichkeit, mit der eine Impersonation gelingt, falls der Text x gewählt wird. Also ist $\alpha = \max\{\alpha(x) \mid x \in X\}$.

Beispiel 14. Sei $K = \{1, 2, 3\}$, $X = \{a, b, c, d\}$ und $Y = \{0, 1\}$. Wir beschreiben H durch die zugehörige **Authentikationsmatrix**. Die Zeilen und Spalten dieser Matrix werden mit den Schlüsseln $k \in K$ und den Texten $x \in X$ indiziert und ihr Eintrag in Zeile k und Spalte x ist der Wert $h_k(x)$.

		0,1	0,2	0,3	0,4
		a	b	c	d
0,25	1	0	0	0	1
0,30	2	1	1	0	1
0,45	3	0	1	1	0

Die umrahmten Zahlen geben die Wahrscheinlichkeiten $p(x)$ bzw. $p(k)$ an. Dann hat der Angreifer folgende Erfolgsaussichten $\alpha(x)$, falls er an Bob den Text x senden möchte.

x	a	b	c	d
$p(x \mapsto 0)$	0,7	0,25	0,55	0,45
$p(x \mapsto 1)$	0,3	0,75	0,45	0,55
$\alpha(x)$	0,7	0,75	0,55	0,55

Folglich ist $\alpha = 0,75$.

◁

Satz 15. Für alle $x \in X$ ist $\alpha(x) \geq \frac{1}{m}$ und daher gilt $\alpha \geq \frac{1}{m}$.

Beweis. Sei $x \in X$ beliebig. Dann gilt

$$\sum_{y \in Y} p(x \mapsto y) = \sum_{y \in Y} \sum_{k \in K(x,y)} p(k) = \sum_{k \in K} p(k) = 1.$$

Somit existiert für jedes $x \in X$ ein $y \in Y$ mit $p(x \mapsto y) \geq \frac{1}{m}$ und dies impliziert

$$\alpha(x) = \max_{y \in Y} p(x \mapsto y) \geq \frac{1}{m}.$$

□